Polyspace® Code Prover™ Access™

Reference

R2019b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Access™ Reference

Revision History

March 2019	Online only	New for Version 2.0 (Release R2019a)
September 2019	Online Only	Revised for Version 2.1 (Release 2019b)

Contents

1

Run-Time Checks

Absolute address usage

Absolute address is assigned to pointer

Description

This check appears when an absolute address is assigned to a pointer.

By default, this check is green. The software assumes the following about the absolute address:

- The address is valid.
- The type of the pointer to which you assign the address determines the initial value stored in the address.

  If you assign the address to an `int*` pointer, the memory zone that the address points to is initialized with an `int` value. The value can be anything allowed for the data type `int`.

To turn this check orange by default for each absolute address usage, use the command-line option `-no-assumption-on-absolute-addresses`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

"Review and Fix Absolute Address Usage Checks"

Examples

Reading content of absolute address

```
enum typeList {CHAR,INT,LONG};
enum typeList showType(void);
long int returnLong(void);
```

```
void main() {
    int *p = (int *)0x32; //Green absolute address usage
    enum typeList myType = showType();

    char x_char;
    int x_int;
    long int x_long;

    if(myType == CHAR)
        x_char = *p;
    else if(myType == INT)
        x_int = *p;
    else {
        x_long = *p;
        long int x2_long = returnLong();
    }
}
```

In this example, the option `-no-assumption-on-absolute-addresses` is not used. Therefore, the **Absolute address usage** check is green when the pointer `p` is assigned an absolute address.

Following this check, the verification assumes that the address is initialized with an `int` value. If you use `x86_64` for `Target processor type (-target)Target processor type (-target)` (`sizeof(char) < sizeof(int) < sizeof(long int)`), the assumption results in the following:

- In the `if(myType == CHAR)` branch, an orange **Overflow** occurs because `x_char` cannot accommodate all values allowed for an `int` variable.
- In the `else if(myType == INT)` branch, if you place your cursor on `x_int` in your verification results, the tooltip shows that `x_int` potentially has all values allowed for an `int` variable.
- In the `else` branch, if you place your cursor on `x_long`, the tooltip shows that `x_long` potentially has all values allowed for an `int` variable. If you place your cursor on `x2_long`, the tooltip shows that `x2_long` potentially has all values allowed for a `long int` variable. The range of values that `x2_long` can take is wider than the values allowed for an `int` variable in the same target.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Arithmetic on pointers with absolute address

```
void main() {
    int *p = (int *)0x32;
    int x = *p;
    p++;
    x = *p;
}
```

In this example, the option `-no-assumption-on-absolute-addresses` is used. The **Absolute address usage** check is orange when the pointer `p` is assigned an absolute address.

Following this check:

- Polyspace considers that `p` points to a valid memory location. Therefore the **Illegally dereferenced pointer** check on the following line is green.
- In the next two lines, the pointer `p` is incremented and then dereferenced. In this case, an **Illegally dereferenced pointer** check appears on the dereference and not an **Absolute address usage** check even though `p` still points to an absolute address.

Check Information

Group: Static memory
Language: C | C++
Acronym: ABS_ADDR

Invalid result of AUTOSAR runnable implementation

Return value or output arguments violate AUTOSAR specifications

Description

This check evaluates functions implementing AUTOSAR runnables. The check determines if the output arguments and return value from the runnable can violate AUTOSAR specifications at run-time.

Using the information on the **Result Details** pane, determine whether the return value or an argument violates data constraints in the AUTOSAR XML specifications or can be NULL-valued. Look for the **!** icon that indicates a definite error or the **?** icon that indicates a possible error.

For each output argument and the return value, the check looks for these violations:

- *Data constraint violations*:

  Suppose, in this implementation of the runnable `foo`, the return value, which represents an application error, has an enumeration data type with a finite set of values. The analysis checks if the return value can acquire a value outside that set at run time.

  ```
  iOperations_ApplicationError foo(
      Rte_Instance const self,
      app_Array_2_n320to320ConstRef aInput,
      app_Array_2_n320to320Ref aOutput,
      app_Enum001Ref aOut2) {
  ...
  }
  ```

  The check can result in a message such as this. The message indicates that the argument has a value that falls outside the constrained range (in this case, the value 43).

? aReturn may not meet its specification.
Specification: {24U,42U,0U,1U,64U,64U,128U,128U,129U,130U,131U,132U,133U,134U,135U,136U,137U,138U,139U,140U,141U,0U,1U}
Actual value (const unsigned int 8): [0 .. 1] or 24 or 43

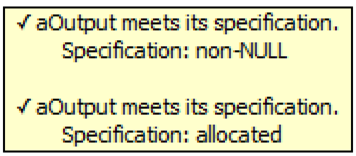
In general, the analysis verifies if each output argument of the runnable and the return value stays within the constrained range allowed by their AUTOSAR data types. You limit values of AUTOSAR data types by referring to data constraints in your ARXML files.

- *NULL or unallocated pointers*:

  Suppose, in this implementation of the runnable `foo`, the first output argument `aOutput` is a pointer. The analysis checks if the pointer is non-NULL and allocated for all possible execution paths upon return from the runnable.

```
iOperations_ApplicationError foo(
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
    app_Enum001Ref aOut2) {
...
}
```

  The check can result in a message such as this.



  In general, the analysis verifies if a pointer output arguments from the runnable are non-NULL and allocated upon return from the runnable.

  By default, the analysis assumes that pointer arguments to runnables and pointers returned from `Rte_` functions are not NULL. To change this assumption, undefine the macro `RTE_PTR2USERCODE_SAFE` using the option `-U` of the `polyspace-autosar` command.

The check first considers the return from the runnable and then the output arguments. If the return from the runnable indicates an error, the check does not look at output arguments on execution paths with the error.

For instance, in this example, the return value is `RTE_E_OK` only if the output argument `aOut2` is not NULL. The check does not consider other execution paths (where the return value is not `RTE_E_OK`). Therefore, it determines that `aOut2` cannot be NULL.

```
// Runnable implementation
iOperations_ApplicationError foo(
```

```
    Rte_Instance const self,
    app_Array_2_n320to320ConstRef aInput,
    app_Array_2_n320to320Ref aOutput,
        app_Enum001Ref aOut2)
{
    iOperations_ApplicationError rc = E_NOT_OK;


    if (aOut2!=NULL_PTR)
    {
        // set invalid value will trigger STD_LIB RED in prove-runnable wrapper
        *aOut2 = 4;
        rc = RTE_E_OK;
    }
    return rc;
}
```

The reason for this behavior is the following: If the return from the runnable indicates an error status on a certain execution path, you can evaluate the error status and take corrective action. Run-time checks are not required for those paths. In certain situations, you might be using one or more output arguments to provide further information on an error status. You might want to check if those output argument can be NULL when the runnable completes execution. If you have this requirement, contact Technical Support.

The check does not flag these situations:

- Output arguments are not written at all within the body of the runnable (or not written along certain execution paths).
- The return value is not initialized within the body of the runnable (or not initialized along certain execution paths).

The analysis checks for conformance with data constraints only when the return value is initialized or output arguments written.

Note If you upload Code Prover results to the Polyspace Access web interface, you can review the results but cannot view details of the AUTOSAR parameter specs within the web interface. To view the parameter specs, open the AUTOSAR specifications (ARXML files) directly or download the results to a Polyspace desktop product and then review them.

Result Information

Group: Other
Language: C
Acronym: AUTOSAR_IMPL

See Also

`Invalid use of AUTOSAR runtime environment function`

Topics

"Interpret Polyspace Code Prover Access Results"

Introduced in R2018a

AUTOSAR runnable not implemented

Function implementing AUTOSAR runnable is not found

Description

This check determines if an AUTOSAR runnable specified in the ARXML specifications is implemented through a function in the source code. The check shows a result only if a function is not found.

You can navigate from the result to the runnable specification through the spec link.

Note If you upload Code Prover results to the Polyspace Access web interface, you can review the results but cannot view details of the AUTOSAR parameter specs within the web interface. To view the parameter specs, open the AUTOSAR specifications (ARXML files) directly or download the results to a Polyspace desktop product and then review them.

Result Information

Group: Other
Language: C
Acronym: AUTOSAR_NOIMPL

See Also

`Invalid result of AUTOSAR runnable implementation`

Introduced in R2018a

Invalid use of AUTOSAR runtime environment function

RTE function argument violates AUTOSAR specifications

Description

This check evaluates calls to functions provided by the AUTOSAR Run-Time Environment (`Rte_` functions). The check determines if the function arguments can violate AUTOSAR XML specifications at run-time.

Using the information on the **Result Details** pane, determine whether an argument violates data constraints in the AUTOSAR XML specifications or can be NULL-valued. Look for the **!** icon that indicates a definite error or the **?** icon that indicates a possible error.

For each function argument, the check looks for these violations:

- *Data constraint violations*:

  Suppose, in this call to `Rte_IWrite_step_out_e4`, the second argument points to a data type that must obey a data constraint. The analysis checks if the constraint can be violated at run time.

  ```
  Rte_IWrite_step_out_e4(self, arg);
  ```

  The check can result in a message such as this. The message indicates that the argument has a value that falls outside the constrained range (in this case, the value 321).

  ```
  ? (*aData)[] may not meet its specification.
      Specification: [-320..320]
      Actual value (const int 32): [-320 .. 321]
  ```

  In general, the analysis verifies if each `Rte_` function argument stays within the constrained range allowed by its AUTOSAR data type. You limit values of AUTOSAR data types by referring to data constraints in your ARXML files. For instance, a constraint specification can look like this (AUTOSAR XML schema version 4.0).

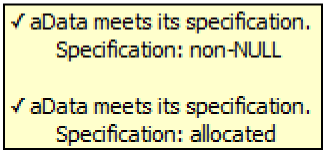
```
<DATA-CONSTR>
   <SHORT-NAME>n320to320</SHORT-NAME>
     <DATA-CONSTR-RULES>
       <DATA-CONSTR-RULE>
         <PHYS-CONSTRS>
           <LOWER-LIMIT INTERVAL-TYPE="CLOSED">-320</LOWER-LIMIT>
           <UPPER-LIMIT INTERVAL-TYPE="CLOSED">320</UPPER-LIMIT>
           <UNIT-REF DEST="UNIT">/jyb/types/units/NoUnit</UNIT-REF>
         </PHYS-CONSTRS>
       </DATA-CONSTR-RULE>
     </DATA-CONSTR-RULES>
</DATA-CONSTR>
...
<APPLICATION-PRIMITIVE-DATA-TYPE>
       <SHORT-NAME>Int_n320to320</SHORT-NAME>
       <CATEGORY>VALUE</CATEGORY>
       <SW-DATA-DEF-PROPS>
         <SW-DATA-DEF-PROPS-VARIANTS>
            <SW-DATA-DEF-PROPS-CONDITIONAL>
                ...
            <DATA-CONSTR-REF DEST="DATA-CONSTR">types/app/constraints/n320to320
            </DATA-CONSTR-REF>
                ...
            </SW-DATA-DEF-PROPS-CONDITIONAL>
        </SW-DATA-DEF-PROPS-VARIANTS>
    </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>
```

- *NULL or unallocated pointers*:

  Suppose, in this call to `Rte_IWrite_step_out_e4`, the second argument is a pointer. The analysis checks if the pointer is non-NULL and allocated for all possible execution paths.

  ```
  Rte_IWrite_step_out_e4(self,arg);
  ```

  The check can result in a message such as this.



In general, the analysis verifies if a pointer argument to an `Rte_` function is non-NULL and allocated.

Note If you upload Code Prover results to the Polyspace Access web interface, you can review the results but cannot view details of the AUTOSAR parameter specs within the web interface. To view the parameter specs, open the AUTOSAR specifications (ARXML files) directly or download the results to a Polyspace desktop product and then review them.

Result Information

Group: Other
Language: C
Acronym: AUTOSAR_USE

See Also

`Invalid result of AUTOSAR runnable implementation`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Introduced in R2018a

Correctness condition

Mismatch occurs during pointer cast or function pointer use

Description

This check determines whether:

- An array is mapped to a larger array through a pointer cast
- A function pointer points to a function with a valid prototype
- A global variable falls outside the range specified through the **Global Assert** mode. See also the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

“Review and Fix Correctness Condition Checks”

Examples

Array is mapped to larger array

```
typedef int smallArray[10];
typedef int largeArray[100];


void main(void) {
    largeArray myLargeArray;
    smallArray *smallArrayPtr = (smallArray*) &myLargeArray;
    largeArray *largeArrayPtr = (largeArray*) smallArrayPtr;
}
```

In this example:

- In the first pointer cast, a pointer of type `largeArray` is cast to a pointer of type `smallArray`. Because the data type `smallArray` represents a smaller array, the **Correctness condition** check is green.
- In the second pointer cast, a pointer of type `smallArray` is cast to a pointer of type `largeArray`. Because the data type `largeArray` represents a larger array, the **Correctness condition** check is red.

Function pointer does not point to function

```
typedef void (*callBack) (float data);
typedef struct {
    char funcName[20];
    callBack func;
} funcStruct;

funcStruct myFuncStruct;

void main(void) {
    float val = 0.f;
    myFuncStruct.func(val);
}
```

In this example, the global variable `myFuncStruct` is not initialized, so the function pointer `myFuncStruct.func` contains `NULL`. When the pointer `myFuncStruct.func` is dereferenced, the **Correctness condition** check produces a red error.

Function pointer points to function through absolute address usage

```
#define MAX_MEMSEG 32764
typedef void (*ptrFunc)(int memseg);
ptrFunc operation = (ptrFunc)(0x003c);

void main(void) {
    for (int i=1; i <= MAX_MEMSEG; i++)
        operation(i);
}
```

In this example, the function pointer `operation` is cast to the contents of a memory location. Polyspace cannot determine whether the location contains a variable or a function code and whether the function is well-typed. Therefore, when the pointer `operation` is dereferenced and used in a function call, the **Correctness condition** check is orange.

After an orange **Correctness condition** check due to absolute address usage, the software assumes that the following variables have the full range of values allowed by their type:

- Variable storing the return value from the function call.

  In the following example, the software assumes that the return value of `operation` is full-range.

  ```
  typedef int (*ptrFunc)(int);
  ptrFunc operation = (ptrFunc)(0x003c);

  int main(void) {
    return operation(0);
  }
  ```

- Variables that can be modified through the function arguments.

  In the following example, the function pointer `operation` takes a pointer argument `ptr` that points to a variable `var`. After the call to `operation`, the software assumes that `var` has full-range value.

  ```
  typedef void (*ptrFunc)(int*);
  ptrFunc operation = (ptrFunc)(0x003c);

  void main(void) {
    int var;
    int *ptr=&var;
    operation(ptr);
  }
  ```

Function pointer points to function with wrong argument type

```
typedef struct {
  double real;
  double imag;
} complex;

typedef int (*typeFuncPtr) (complex*);
```

```
int func(int* x);

void main() {
  typeFuncPtr funcPtr = (typeFuncPtr)&func;
  int arg = 0, result = funcPtr((complex*)&arg);
}
```

In this example, the function pointer `funcPtr` points to a function with argument type `complex*`. However, the pointer is assigned the address of function `func` whose argument type is `int*`. Because of this type mismatch, the **Correctness condition** check is orange.

Function pointer points to function with wrong number of arguments

```
typedef int (*typeFuncPtr) (int, int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg1 = 0, arg2 = 0, result = funcPtr(arg1,arg2);
}
```

In this example, the function pointer `funcPtr` points to a function with two `int` arguments. However, it is assigned the function `func` which has one `int` argument only. Because of this mismatch in number of arguments, the **Correctness condition** check is orange.

Function pointer points to function with wrong return type

```
typedef double (*typeFuncPtr) (int);

int func(int);

void main() {
    typeFuncPtr funcPtr = (typeFuncPtr)&func;
    int arg = 0;
    double result = funcPtr(arg);
}
```

In this example, the function pointer `funcPtr` points to a function with return type `double`. However, it is assigned the function `func` whose return type is `int`. Because of this mismatch in return types, the **Correctness condition** check is orange.

Variable falls outside Global Assert range

```
int glob = 0;
int func();

void main() {
    glob = 5;
    glob = func();
    glob+= 20;
}
```

In this example, a range of `0..10` was specified for the global variable `glob`.

- In the statement `glob=5;`, a green **Correctness condition** check appears on `glob`.
- In the statement `glob=func();`, an orange **Correctness condition** check appears on `glob` because the return value of stubbed function `func` can be outside `0..10`.

  After this statement, Polyspace considers that `glob` has values in `0..10`.
- In the statement `glob+=20;`, a red **Correctness condition** check appears on `glob` because after the addition, `glob` has values in `20..30`.

See also *Constrain Global Variable Range* in the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Other
Language: C | C++
Acronym: COR

See Also

Topics

"Interpret Polyspace Code Prover Access Results"

“Code Prover Analysis Following Red and Orange Checks”

Division by zero

Division by zero occurs

Description

This check determines whether the right operand of a division or modulus operation is zero.

Diagnosing This Check

“Review and Fix Division by Zero Checks”

Examples

Red integer division by zero

```
#include <stdio.h>

void main() {
    int x=2;
    printf("Quotient=%d",100/(x-2));
}
```

In this example, the denominator `x-2` is zero.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

In a complex code, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```
#include <stdio.h>
int input();
void main() {
```

```
    int x=input();
    if(x>0) { //Avoid overflow
        if(x!=2 && x>0)
            printf("Quotient=%d",100/(x-2));
        else
            printf("Zero denominator.");
    }
}
```

Red integer division by zero after `for` loop

```
#include <stdio.h>
void main() {
    int x=-10;
    for (int i=0; i<10; i++)
        x+=3;
    printf("Quotient=%d",100/(x-20));
}
```

In this example, the denominator `x-20` is zero.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

After several iterations of a `for` loop, it is difficult to keep track of values and avoid zero denominators. Therefore, it is good practice to check for zero denominator before every division.

```
#include <stdio.h>
#define MAX 10000
int input();

void main() {
    int x=input();
    for (int i=0; i<10; i++) {
        if(x < MAX) //Avoid overflow
            x+=3;
    }

    if(x>0) { //Avoid overflow
        if(x!=20)
            printf("Quotient=%d",100/(x-20));
        else
```

```
            printf("Zero denominator.");
    }
}
```

Orange integer division by zero inside `for` loop

```
#include<stdio.h>

void main() {
    printf("Sequence of ratios: \n");
    for(int count=-100; count<=100; count++)
        printf(" %.2f ", 1/count);
}
```

In this example, `count` runs from -100 to 100 through zero. When `count` is zero, the **Division by zero** check returns a red error. Because the check returns green in the other `for` loop runs, the `/` symbol is orange.

There is also a red **Non-terminating loop** error on the `for` loop. This red error indicates a definite error in one of the loop runs.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division.

```
#include<stdio.h>

void main() {
    printf("Sequence of ratios: \n");
    for(int count=-100; count<=100; count++) {
        if(count != 0)
            printf(" %.2f ", 1/count);
        else
            printf(" Infinite ");
    }
}
```

Orange float division by zero inside `for` loop

```
#include <stdio.h>
#include <math.h>

#define stepSize 0.1
```

```
void main() {
    float divisor = -1.0;
    int numberOfSteps = (int)((2.0*1.0)/stepSize);

    printf("Divisor running from -1.0 to 1.0\n");
    for(int count = 1; count <= numberOfSteps; count++) {
        divisor+= stepSize;
    divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
        printf(" .2f ", 1.0/divisor);
    }
}
```

In this example, `divisor` runs from –1.0 to 1.0 through 0.0. When `divisor` is 0.0, the **Division by zero** check returns a red error. Because the check returns green in the other `for` loop runs, the `/` symbol is orange.

There is no red **Non-terminating loop** error on the `for` loop. The red error does not appear because Polyspace approximates the values of `divisor` by a broader range. Therefore, Polyspace cannot determine if there is a definite error in one of the loop runs.

Correction — Check for zero denominator

One possible correction is to check for a zero denominator before division. For `float` variables, do not check if the denominator is exactly zero. Instead, check whether the denominator is in a narrow range around zero.

```
#include <stdio.h>
#include <math.h>

#define stepSize 0.1

void main() {
    float divisor = -1.0;
    int numberOfSteps = (int)((2*1.0)/stepSize);

    printf("Divisor running from -1.0 to 1.0\n");;
    for(int count = 1; count <= numberOfSteps; count++) {
        divisor += stepSize;
    divisor = ceil(divisor * 10.) / 10.; // one digit of imprecision
        if(divisor < -0.00001 || divisor > 0.00001)
            printf(" .2f ", 1.0/divisor);
        else
            printf(" Infinite ");
```

```
    }
}
```

Check Information

Group: Numerical
Language: C | C++
Acronym: ZDV

See Also

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Function not called

Function is defined but not called

Description

This check on a function definition determines if the function is called anywhere in the code. This check is disabled if your code does not contain a `main` function.

Use this check to satisfy ISO® 26262 requirements about function coverage. For example, see table 15 of ISO 26262, part 6.

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

"Review and Fix Function Not Called Checks"

Examples

Function not called

```
#define max 100
int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}
```

```
void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            var=0;
    }
}
```

In this example, the function `reset` is defined but not called. Therefore, a gray **Function not called** check appears on the definition of `reset`.

Correction: Call Function

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as instruction `var=0;`. Therefore, replace the instruction with the function call.

```
#define max 100

int var;
int getValue(void);
int getSaturation(void);

void reset() {
    var=0;
}

void main() {
    int saturation = getSaturation(),val;
    for(int index=1; index<=max; index++) {
        val = getValue();
        if(val>0 && val<10)
            var += val;
        if(var > saturation)
            reset();
    }
}
```

Function Called from Another Uncalled Function

```
#define max 100

int var;
int numberOfResets;
int getValue();
int getSaturation();

void updateCounter() {
  numberOfResets++;
}

void reset() {
  updateCounter();
  var=0;
}

void main() {
  int saturation = getSaturation(),val;
  for(int index=1; index<=max; index++) {
    val = getValue();
    if(val>0 && val<10)
      var += val;
    if(var > saturation) {
      numberOfResets++;
      var=0;
    }
  }
}
```

In this example, the function `reset` is defined but not called. Since the function `updateCounter` is called only from `reset`, a gray **Function not called** error appears on the definition of `updateCounter`.

Correction: Call Function

One possible correction is to call the function `reset`. In this example, the function call `reset` serves the same purpose as the instructions in the branch of `if(var > saturation)`. Therefore, replace the instructions with the function call.

```
#define max 100
```

```
int var;
int numberOfResets;
int getValue(void);
int getSaturation(void);

void updateCounter() {
  numberOfResets++;
}

void reset() {
  updateCounter();
  var=0;
}

void main() {
  int saturation = getSaturation(),val;
  for(int index=1; index<=max; index++) {
    val = getValue();
    if(val>0 && val<10)
      var += val;
    if(var > saturation)
      reset();
  }
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: FNC

See Also

Function not reachable

Function is called from unreachable part of code

Description

This check appears on a function definition. The check appears gray if the function is called only from an unreachable part of the code. The unreachable code can occur in one of the following ways:

- The code is reached through a condition that is always false.
- The code follows a `break` or `return` statement.
- The code follows a red check.

If your code does not contain a `main` function, this check is disabled

---

Note This check is not turned on by default. To turn on this check, you must specify the appropriate analysis option. For more information, see `Detect uncalled functions (-uncalled-function-checks)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

"Review and Fix Function Not Reachable Checks"

Examples

Function Call from Unreachable Branch of Condition

```
#include<stdio.h>
#define SIZE 100

void increase(int* arr, int index);
```

```
void printError()
{
  printf("Array index exceeds array size.");
}

void main() {
  int arr[SIZE],i;
  for(i=0; i<SIZE; i++)
    arr[i]=0;

  for(i=0; i<SIZE; i++) {
    if(i<SIZE)
      increase(arr,i);
    else
      printError();
  }
}
```

In this example, in the second `for` loop in `main`, `i` is always less than `SIZE`. Therefore, the `else` branch of the condition `if(i<SIZE)` is never reached. Because the function `printError` is called from the `else` branch alone, there is a gray **Function not reachable** check on the definition of `printError`.

Function Call Following Red Error

```
#include<stdio.h>

int getNum(void);

void printSuccess()
{
  printf("The operation is complete.");
}

void main() {
  int num=getNum(), den=0;
  printf("The ratio is %.2f", num/den);
  printSuccess();
}
```

In this example, the function `printSucess` is called following a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** check on the definition of `printSuccess`.

Function Call from Another Unreachable Function

```
#include<stdio.h>
#define MAX 1000
#define MIN 0

int getNum(void);

void checkUpperBound(double ratio)
{
    if(ratio < MAX)
        printf("The ratio is within bounds.");
}

void checkLowerBound(double ratio)
{
    if(ratio > MIN)
        printf("The ratio is within bounds.");
}

void checkRatio(double ratio)
{
    checkUpperBound(ratio);
    checkLowerBound(ratio);
}

void main() {
    int num=getNum(), den=0;
    double ratio;
    ratio=num/den;
    checkRatio(ratio);
}
```

In this example, the function `checkRatio` follows a red **Division by Zero** error. Therefore, there is a gray **Function not reachable** error on the definition of `checkRatio`. Because `checkUpperBound` and `checkLowerBound` are called only from `checkRatio`, there is also a gray **Function not reachable** check on their definitions.

Function Call from Unreachable Code Using Function Pointer

```
#include<stdio.h>

int getNum(void);
int getChoice(void);

int num, den, choice;
double ratio;

void display(void)
{
    printf("Numerator = %d, Denominator = %d", num, den);
}

void display2(void)
{
    printf("Ratio = %.2f",ratio);
}


void main() {
    void (*fptr)(void);

    choice = getChoice();
    if(choice == 0)
        fptr = &display;
    else
        fptr = &display2;

    num = getNum();
    den = 0;
    ratio = num/den;

    (*fptr)();
}
```

In this example, depending on the value of `choice`, the function pointer `fptr` can point to either `display` or to `display2`. The call through `fptr` follows a red **Division by Zero** error. Because `display` and `display2` are called only through `fptr`, a gray **Function not reachable** check appears on their definitions.

Check Information

Group: Data flow
Language: C | C++
Acronym: FNR

See Also

`Function not called` | `Unreachable code`

Function not returning value

C++ function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on the function definition.

Diagnosing This Check

"Review and Fix Function Not Returning Value Checks"

Examples

Function does not return value for any input

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch<=0)
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

In this example, for all values of `ch`, `reply(ch)` has no return value. Therefore the **Function not returning value** check returns a red error on the definition of `reply()`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch<=0)
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch<10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch < = 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Function not returning value** check returns an orange error on the definition of `reply()`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch<10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

Check Information

Group: C++
Language: C++
Acronym: FRV

See Also

`Return value not initialized`

Topics

"Interpret Polyspace Code Prover Access Results"

Illegally dereferenced pointer

Pointer is dereferenced outside bounds

Description

This check on a pointer dereference determines whether the pointer is NULL or points outside its bounds.

The check message shows you the pointer offset and buffer size in bytes. A pointer points outside its bounds when the sum of the offset and pointer size exceeds the buffer size.

- *Buffer*: When you assign an address to a pointer, a block of memory is allocated to the pointer. You cannot access memory beyond that block using the pointer. The size of this block is the buffer size.

  Sometimes, instead of a definite value, the size can be a range. For instance, if you create a buffer dynamically using `malloc` with an unknown input for the size, Polyspace assumes that the array size can take the full range of values allowed by the input data type.
- *Offset*: You can move a pointer within the allowed memory block by using pointer arithmetic. The difference between the initial location of the pointer and its current location is the offset.

  Sometimes, instead of a definite value, the offset can be a range. For instance, if you access an array in a loop, the offset changes value in each loop iteration and takes a range of values throughout the loop.

For instance, if the pointer points to an array:

- The buffer size is the array size.
- The offset is the difference between the beginning of the array and the current location of the pointer.

Diagnosing This Check

"Review and Fix Illegally Dereferenced Pointer Checks"

Examples

Pointer points outside array bounds

```
#define Size 1024

int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++){
        *p = input();
    }
    *p = input();
}
```

In this example:

- Before the `for` loop, `p` points to the beginning of the array `arr`.
- After the `for` loop, `p` points outside the array.

The **Illegally dereferenced pointer** check on dereference of `p` after the `for` loop produces a red error.

Correction — Remove illegal dereference

One possible correction is to remove the illegal dereference of `p` after the `for` loop.

```
#define Size 1024

int input(void);

void main() {
    int arr[Size];
    int *p = arr;

    for (int index = 0; index < Size ; index++, p++) {
        *p = input();
    }
}
```

Pointer points outside structure field

```
typedef struct S {
    int f1;
    int f2;
    int f3;
} S;

void Initialize(int *ptr) {
    *ptr = 0;
    *(ptr+1) = 0;
    *(ptr+2) = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct.f1);
}
```

In this example, in the body of `Initialize`, `ptr` is an `int` pointer that points to the first field of the structure. When you attempt to access the second field through `ptr`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Avoid memory access outside structure field

One possible correction is to pass a pointer to the entire structure to `Initialize`.

```
typedef struct S {
    int f1;
    int f2;
    int f3;
} S;

void Initialize(S* ptr) {
    ptr->f1 = 0;
    ptr->f2 = 0;
    ptr->f3 = 0;
}

void main(void) {
    S myStruct;
    Initialize(&myStruct);
}
```

NULL pointer is dereferenced

```
#include<stdlib.h>

void main() {
    int *ptr=NULL;
    *ptr=0;
}
```

In this example, `ptr` is assigned the value `NULL`. Therefore when you dereference `ptr`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Avoid NULL pointer dereference

One possible correction is to initialize `ptr` with the address of a variable instead of `NULL`.

```
void main() {
    int var;
    int *ptr=&var;
    *ptr=0;
}
```

Offset on NULL pointer

```
int getOffset(void);

void main() {
    int *ptr = (int*) 0 + getOffset();
    if(ptr != (int*)0)
        *ptr = 0;
}
```

In this example, although an offset is added to `(int*) 0`, Polyspace does not treat the result as a valid address. Therefore when you dereference `ptr`, the **Illegally dereferenced pointer** check produces a red error.

Bit field type is incorrect

```
struct flagCollection {
    unsigned int flag1: 1;
    unsigned int flag2: 1;
    unsigned int flag3: 1;
```

```
    unsigned int flag4: 1;
    unsigned int flag5: 1;
    unsigned int flag6: 1;
    unsigned int flag7: 1;
};

char getFlag(void);

int main()
{
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
    return 0;
}
```

In this example:

- The fields of `flagCollection` have type `unsigned int`. Therefore, a `flagCollection` structure requires 32 bits of memory in a 32-bit architecture even though the fields themselves occupy 7 bits.
- When you cast a `char` address `&myFlag` to a `flagCollection` pointer `myFlagCollection`, you assign only 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` produces a red error.

Correction — Use correct type for bit fields

One possible correction is to use `unsigned char` as field type of `flagCollection` instead of `unsigned int`. In this case:

- The structure `flagCollection` requires 8 bits of memory.
- When you cast the `char` address `&myFlag` to the `flagCollection` pointer `myFlagCollection`, you also assign 8 bits of memory to the pointer. Therefore, the **Illegally dereferenced pointer** check on dereference of `myFlagCollection` is green.

```
struct flagCollection {
    unsigned char flag1: 1;
    unsigned char flag2: 1;
```

```
    unsigned char flag3: 1;
    unsigned char flag4: 1;
    unsigned char flag5: 1;
    unsigned char flag6: 1;
    unsigned char flag7: 1;
};

char getFlag(void);

int main()
{
    unsigned char myFlag = getFlag();
    struct flagCollection* myFlagCollection;
    myFlagCollection = (struct flagCollection *) &myFlag;
    if (myFlagCollection->flag1 == 1)
        return 1;
    return 0;
}
```

Return value of `malloc` is not checked for `NULL`

```
#include <stdlib.h>

void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    *q = 'a';
}
```

In this example, `malloc` can return `NULL` to `p`. Therefore, when you assign `p` to `q` and dereference `q`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Check return value of `malloc` for `NULL`

One possible correction is to check `p` for `NULL` before dereferencing `q`.

```
#include <stdlib.h>
void main(void)
{
    char *p = (char*)malloc(1);
    char *q = p;
    if(p!=NULL) *q = 'a';
}
```

Pointer to union gets insufficient memory from `malloc`

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
    union {
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(char));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

In this example:

- Because the union `myVar` has an `int` variable as a field, it must be assigned 4 bytes in a 32-bit architecture. Therefore, the structure `myType` must be assigned 4+4 = 8 bytes.
- `malloc` returns `sizeof(int) + sizeof(char)`=4+1=5 bytes of memory to `myTypePtr`, a pointer to a `myType` structure. Therefore, when you dereference `myTypePtr`, the **Illegally dereferenced pointer** check returns a red error.

Correction — Assign sufficient memory to pointer

One possible correction is to assign 8 bytes of memory to `myTypePtr` before dereference.

```
#include <stdlib.h>

enum typeName {CHAR,INT};

typedef struct {
    enum typeName myTypeName;
    union {
```

```
        char myChar;
        int myInt;
    } myVar;
} myType;

void main() {
    myType* myTypePtr;
    myTypePtr = (myType*)malloc(sizeof(int) + sizeof(int));
    if(myTypePtr != NULL) {
        myTypePtr->myTypeName = INT;
    }
}
```

Structure is allocated memory partially

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

In this example, `cuboidPtr` obtains sufficient memory to accommodate two of its fields. Because the ANSI® C standards do not allow such partial memory allocations, the **Illegally dereferenced pointer** check on the dereference of `cuboidPtr` produces a red error.

Correction — Allocate full memory

To observe ANSI C standards, `cuboidPtr` must be allocated full memory.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(cuboid));
    if(cuboidPtr!=NULL) {
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Correction — Use Polyspace analysis option

You can allow partial memory allocation for structures, yet not have a red **Illegally dereferenced pointer** error. To allow partial memory allocation, on the **Configuration** pane, under **Check Behavior**, select **Allow incomplete or partial allocation of structures**.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} rectangle;

typedef struct {
    int length;
    int breadth;
    int height;
} cuboid;

void main() {
    cuboid *cuboidPtr = (cuboid*)malloc(sizeof(rectangle));
    if(cuboidPtr!=NULL) {
```

```
        cuboidPtr->length = 10;
        cuboidPtr->breadth = 10;
    }
}
```

Pointer to one field of structure points to another field

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;


void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

In this example, although `squarePtr` is a `char` pointer, it is assigned the address of the integer `mySquare.length`. Because:

- `char` occupies 1 byte,
- `int` occupies 4 bytes in a 32–bit architecture,

`squarePtr` can access the four bytes of `mySquare.length` through pointer arithmetic. But when it accesses the first byte of another field `mySquare.breadth`, the **Illegally dereferenced pointer** check produces a red error.

Correction — Assign address of structure instead of field

One possible correction is to assign `squarePtr` the address of the full structure `mySquare` instead of `mySquare.length`. `squarePtr` can then access all the bytes of `mySquare` through pointer arithmetic.

```
#include <stdlib.h>
typedef struct {
```

```
    int length;
    int breadth;
} square;


void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

Correction — Use Polyspace analysis option

You can use a pointer to navigate across the fields of a structure and not produce a red **Illegally dereferenced pointer** error. To allow such navigation, on the **Configuration** pane, under **Check Behavior**, select **Enable pointer arithmetic across fields**.

This option is not available for C++ projects.

```
#include <stdlib.h>
typedef struct {
    int length;
    int breadth;
} square;


void main() {
    square mySquare;
    char* squarePtr = (char*)&mySquare.length;
//Assign zero to mySquare.length byte by byte
    for(int byteIndex=1; byteIndex<=4; byteIndex++) {
        *squarePtr=0;
        squarePtr++;
    }
//Assign zero to first byte of mySquare.breadth
    *squarePtr=0;
}
```

Function returns pointer to local variable

```
void func2(int *ptr) {
    *ptr = 0;
}

int* func1(void) {
    int ret = 0;
    return &ret ;
}
void main(void) {
    int* ptr = func1() ;
    func2(ptr) ;
}
```

In the following code, `ptr` points to `ret`. Because the scope of `ret` is limited to `func1`, when `ptr` is accessed in `func2`, the access is illegal. The verification produces a red **Illegally dereferenced pointer** check on `*ptr`.

By default, Polyspace Code Prover does not detect functions returning pointers to local variables. To detect such cases, use the option `Detect stack pointer dereference outside scope (-detect-pointer-escape)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Static memory
Language: C | C++
Acronym: IDP

See Also

`Non-initialized pointer`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Incorrect object oriented programming

Dynamic type of `this` pointer is incorrect

Description

This check on a class member function call determines if the call is valid.

A member function call can be invalid for the following reasons:

- You call the member function through a function pointer that points to the function. However, the data types of the arguments or return values of the function and the function pointer do not match.
- You call a pure `virtual` member function from the class constructor or destructor.
- You call a `virtual` member function through an incorrect `this` pointer. The `this` pointer stores the address of the object used to call the function. The `this` pointer can be incorrect because:
  - You obtain an object through a cast from another object. The objects are instances of two unrelated classes.
  - You perform pointer arithmetic on a pointer pointing to an array of objects. However, the pointer arithmetic causes the pointer to go outside the array bounds. When you dereference the pointer, it is not pointing to a valid object.

Diagnosing This Check

“Review and Fix Incorrect Object Oriented Programming Checks”

Examples

Pointer to method has incorrect type

```
#include <iostream>
class myClass {
```

```
public:
  void method() {}
};

void main() {
  myClass Obj;
  int (myClass::*methodPtr) (void) = (int (myClass::*) (void))
&myClass::method;
  int res = (Obj.*methodPtr)();
  std::cout << "Result = " << res;
}
```

In this example, the pointer `methodPtr` has return type `int` but points to `myClass:method` that has return type `void`. Therefore, when `methodPtr` is dereferenced, the **Incorrect object oriented programming** check produces a red error.

Pointer to method contains NULL when dereferenced

```
#include <iostream>
class myClass {
public:
  void method() {}
};

void main() {
  myClass Obj;
  void (myClass::*methodPtr) (void) =  &myClass::method;
  methodPtr = 0;
  (Obj.*methodPtr)();
}
```

In this example, `methodPtr` has value `NULL` when it is dereferenced.

Pure virtual function is called in base class constructor

```
class Shape {
public:
  Shape(Shape *myShape) {
    myShape->setShapeDimensions(0.0);
  }
  virtual void setShapeDimensions(double) = 0;
};
```

```
class Square: public Shape {
  double side;
public:
  Square():Shape(this) {
  }
  void setShapeDimensions(double);
};

void Square::setShapeDimensions(double val) {
  side=val;
}

void main() {
  Square sq;
  sq.setShapeDimensions(1.0);
}
```

In this example, the derived class constructor `Square::Square` calls the base class constructor `Shape::Shape()` with its `this` pointer. The base class constructor then calls the pure virtual function `Shape::setShapeDimensions` through the `this` pointer. Since the call to a pure virtual function from a constructor is undefined, the **Incorrect object oriented programming** check produces a red error.

Incorrect `this` Pointer: Cast Between Pointers to Unrelated Objects

```
#include <new>

class Foo {
public:
  void funcFoo() {}
};


class Bar {
public:
  virtual void funcBar() {}
};

void main() {
  Foo *FooPtr = new Foo;
  Bar *BarPtr = (Bar*)(void*)FooPtr;
```

```
  BarPtr->funcBar();
}
```

In this example, the classes `Foo` and `Bar` are not related. When a `Foo*` pointer is cast to a `Bar*` pointer and the `Bar*` pointer is used to call a `virtual` member function of class `Bar`, the **Incorrect object oriented programming** check produces a red error.

Incorrect this Pointer: Pointer Out of Bounds

```
#include <new>
class Foo {
public:
    virtual void func() {}
};

void main() {
    Foo *FooPtr = new Foo[4];
    for(int i=0; i<=4; i++)
        FooPtr++;
    FooPtr->func();
    delete [] FooPtr;
}
```

In this example, the pointer `FooPtr` points outside the allocated bounds when it is used to call the `virtual` member function `func()`. It does not point to a valid object. Therefore, the **Incorrect object oriented programming** check produces a red error.

Incorrect this Pointer: Non-initialized Object

```
class Foo {
public:
  virtual int func() {
    return 1;
  }
};

class Ref {
public:
  Ref(Foo* foo) {
    foo->func();
  }
};
```

```
class Bar {
private:
  Ref m_ref;
  Foo m_Foo;
public:
  Bar() : m_ref(&m_Foo) {}
};
```

In this example, the constructor `Bar::Bar()` calls the constructor `Ref::Ref()` with the address of `m_Foo` before `m_Foo` is initialized. When the `virtual` member function `func` is called through a pointer pointing to `&m_Foo`, the **Incorrect object oriented programming** check produces a red error.

To reproduce the results, analyze only the class `Bar` using the option `Class (-class-analyzer)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Incorrect `this` Pointer: Cast from Base to Derived Class Pointer

```
#include <new>

class Foo {
public:
  virtual void funcFoo() {}
};


class Bar: public Foo {
public:
  void funcFoo() {}
};

void main() {
  Foo *FooPtr = new Foo;
  Bar *BarPtr = (Bar*)(void*)FooPtr;
  BarPtr->funcFoo();
}
```

In this example, you might intend to call the derived class version of `funcFoo` but depending on your compiler, you call the base class version or encounter a segmentation fault.

The pointer FooPtr points to a Foo object. The cast incorrectly attempts to convert the Foo* pointer FooPtr to a Bar* pointer BarPtr. BarPtr still points to the base Foo object and cannot access Bar::funcFoo.

Correction – Make Base Class Pointer Point Directly to Derived Class Object

C++ polymorphism allows defining a pointer that can traverse the class hierarchy to point to the most derived member function. To implement polymorphism correctly, start from the base class pointer and make it point to a derived class object.

```
#include <new>

class Foo {
public:
  virtual void funcFoo() {}
};


class Bar: public Foo {
public:
  void funcFoo() {}
};

void main() {
  Foo *FooPtr = new Bar;
  FooPtr->funcFoo();
}
```

Check Information

Group: C++
Language: C++
Acronym: OOP

See Also

Base class destructor not virtual | Incompatible types prevent overriding | Missing virtual inheritance | Partial override of overloaded virtual functions

Topics

“Interpret Polyspace Code Prover Access Results”
“Code Prover Analysis Following Red and Orange Checks”

Invalid C++ specific operations

C++ specific invalid operations occur

Description

These checks on C++ code operations determine whether the operations are valid. The checks look for a range of invalid behaviors:

- Array size is not strictly positive.
- `typeid` operator dereferences a `NULL` pointer.
- `dynamic_cast` operator performs an invalid cast.
- (C++11 and beyond) The number of array initializer clauses exceeds the number of array elements to initialize.
- (C++11 and beyond) The pointer argument to a placement new operator does not point to enough memory.

Diagnosing This Check

"Review and Fix Invalid C++ Specific Operations Checks"

Examples

Array size Not Strictly Positive

```
class License {
protected:
  int numberOfUsers;
  char (*userList)[20];
  int *licenseList;
public:
  License(int numberOfLicenses);
  void initializeList();
  char* getUser(int);
```

```
  int getLicense(int);
};

License::License(int numberOfLicenses) : numberOfUsers(numberOfLicenses) {
  userList = new char [numberOfUsers][20];
  licenseList = new int [numberOfUsers];
  initializeList();
}

int getNumberOfLicenses();
int getIndexForSearch();

void main() {
  int n = getNumberOfLicenses();
  if(n >= 0 && n <= 100) {
    License myFirm(n);
    int index = getIndexForSearch();
    myFirm.getUser(index);
    myFirm.getLicense(index);
  }
}
```

In this example, the argument `n` to the constructor `License::License` falls into two categories:

- `n = 0`: When the `new` operator uses this argument, the **Invalid C++ specific operations** produce an error.
- `n > 0`: When the `new` operator uses this argument, the **Invalid C++ specific operations** is green.

Combining the two categories of arguments, the **Invalid C++ specific operations** produce an orange error on the `new` operator.

`typeid` Operator Dereferencing `NULL` Pointer

To see this issue, enable the option `Consider environment pointers as unsafe (-stubbed-pointers-are-unsafe)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

```
#include <iostream>
#include <typeinfo>
#define PI 3.142
```

```
class Shape {
public:
  Shape();
  virtual void setVal(double) = 0;
  virtual double area() = 0;
};

class Circle: public Shape {
  double radius;
public:
  Circle(double radiusVal):Shape() {
    setVal(radiusVal);
  }

  void setVal(double radiusVal) {
     radius = radiusVal;
  }

  double area() {
    return (PI * radius * radius);
  }
};


Shape* getShapePtr();

void main() {
  Shape* shapePtr = getShapePtr();
  double val;

  if(typeid(*shapePtr)==typeid(Circle)) {
    std::cout<<"Enter radius:";
    std::cin>>val;
    shapePtr->setVal(val);
    std::cout<<"Area of circle = "<<shapePtr->area();
  }
  else {
    std::cout<<"Shape is not a circle.";
  }

}
```

In this example, the `Shape*` pointer `shapePtr` returned by `getShapePtr()` function can be `NULL`. Because a possibly `NULL`-valued `shapePtr` is used with the `typeid` operator, the **Invalid C++ specific operations** check is orange.

Incorrect `dynamic_cast` on Pointers

```
class Base {
public :
  virtual void func() ;
};

class Derived : public Base  {
};

Base* returnObj(int flag) {
  if(flag==0)
    return new Derived;
  else
    return new Base;
}


int main() {

    Base * ptrBase;
    Derived * ptrDerived;

    ptrBase = returnObj(0) ;
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Correct dynamic cast
    assert(ptrDerived != 0); //Returned pointer is not null

    ptrBase = returnObj(1);
    ptrDerived = dynamic_cast<Derived*>(ptrBase); //Incorrect dynamic cast
    // Verification continues despite red
    assert(ptrDerived == 0); //Returned pointer is null
}
```

In this example, the **Invalid C++ specific operations** on the `dynamic_cast` operator are:

- Green, when the pointer `ptrBase` that the operator casts to `Derived` is already pointing to a `Derived` object.

- Red, when the pointer `ptrBase` that the operator casts to `Derived` is pointing to a `Base` object.

  Red checks typically stop the verification in the same scope as the check. However, after red **Invalid C++ specific operations** on `dynamic_cast` operation involving pointers, the verification continues. The software assumes that the `dynamic_cast` operator returns a `NULL` pointer.

Incorrect dynamic_cast on References

```
class Base {
public :
  virtual void func() ;
};

class Derived : public Base  {
};

Base& returnObj(int flag) {
  if(flag==0)
    return *(new Derived);
  else
    return *(new Base);
}


int main() {
  Base & refBase1 = returnObj(0);
  Derived & refDerived1 = dynamic_cast<Derived&>(refBase1); //Correct dynamic cast;

  Base & refBase2 = returnObj(1);
  Derived & refDerived2 = dynamic_cast<Derived&>(refBase2); //Incorrect dynamic cast
  // Analysis stops
  assert(1);
}
```

In this example, the **Invalid C++ specific operations** on the `dynamic_cast` operator are:

- Green, when the reference `refBase1` that the operator casts to `Derived&` is already referring to a `Derived` object.
- Red, when the reference `refBase2` that the operator casts to `Derived&` is referring to a `Base` object.

After red **Invalid C++ specific operations** on `dynamic_cast` operation involving pointers, the software does not verify the code in the same scope as the check. For instance, the software does not perform the **User assertion** check on the `assert` statement.

(C++11 and Beyond) Excess Initializer Clauses in Array Initialization

```
#include <stdio.h>

int* arr_const;

void allocate_consts(int size) {
    if(size>1)
      arr_const = new int[size]{0,1,2};
    else if(size==1)
      arr_const = new int[size]{0,1};
    else
       printf("Nonpositive array size!");
}

int main() {
    allocate_consts(3);
    allocate_consts(1);
    return 0;
}
```

In this example, the **Invalid C++ specific operations** check determines if the number of initializer clauses match the number of elements to initialize.

In the first call to `allocate_consts`, the initialization list has three elements to initialize an array of size three. The **Invalid C++ specific operations** check on the `new` operator is green. In the second call, the initialization list has two elements but initializes an array of size one. The check on the `new` operator is red.

(C++11 and Beyond) Pointer Argument to Placement `new` Operator Does Not Point to Enough Memory

```
#include <new>
```

```
class aClass {
  virtual void func();
};

void allocateNObjects(unsigned int n) {
    char* location = new char[sizeof(aClass)];
    aClass* objectLocation = new(location) aClass[n];
}
```

In this example, memory equal to the size of one `aClass` object is associated with the pointer `location`. However, depending on the function argument `n` more than one object can be allocated when using the placement `new` operator. The pointer `location` might not have enough memory for the objects allocated.

Check Information

Group: C++
Language: C++
Acronym: CPP

See Also

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

External Websites

C++ Reference: `dynamic_cast` conversion

Invalid operation on floats

Result of floating-point operation is NaN for non-NaN operands

Description

This check determines if the result of a floating-point operation is NaN. The check is performed only if you enable a verification mode that incorporates NaNs and specify that the verification must highlight operations that result in NaN.

If you specify that the verification must produce a warning for NaN, the check is:

- Red, if the operation produces NaN on all execution paths that the software considers, and the operands are not NaN.
- Orange, if the operation produces NaN on some of the execution paths when the operands are not NaN.
- Green, if the operation does not produce NaN unless the operands are NaN.

If you specify that the verification must forbid NaN, the check color depends on the result of the operation only. The color does not depend on the operands.

The check also highlights conversions from floating-point variables to integers where the floating-point variable can be NaN. In this case, the check is always performed when you incorporate NaNs in the verification and does not allow NaNs as input to the conversion.

To enable this verification mode, use these options:

- `Consider non finite floats (-allow-non-finite-floats)`
- `NaNs (-check-nan)`: Use argument `warn-first` or `forbid`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

NaN Detected with Red Check

Results in `forbid` mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

In this example, both the operands of the `-` operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the `-` operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on `y` in the `return` statement.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    double y=x-x;
    return y;
}
```

In this example, both the operands of the `-` operation are not NaN but the result is NaN. The **Invalid operation on floats** check on the `-` operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on `y` in the `return` statement. If you place your cursor on `y` in the verification result, you see that it has the value NaN.

NaN Detected with Orange Check

Results in `forbid` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
    return ret;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. In the `forbid` mode, following the check, the verification terminates the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it does not have the value `NaN`.

Results in `warn-first` mode:

```
double func(double arg1, double arg2) {
    double ret=arg1-arg2;
    return ret;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can be both infinite, for instance, and the result of `arg1-arg2` can be NaN. The orange checks in `warn-first` mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution path that results in NaN. If you place your cursor on `ret` in the `return` statement, it continues to have the value `NaN` along with other possible values.

Orange Check Despite NaN Being the Only Result

```
double func(double arg1, double arg2) {
    double z=arg1-arg2;
    return z;
}

void caller() {
    double x=1.0/0.0;
    double y=x-x;
    func(x,x);
    func(y,y);
}
```

In this example, in `func`, the result of the `-` operation is always NaN but the **Invalid operation on floats** check is orange instead of red.

- In the first call to `func`, both the operands `arg1` and `arg2` are not NaN, but the result is NaN. So, the check is red.
- In the second call to `func`, both the operands `arg1` and `arg2` are NaN, and therefore the result is NaN. So, the check is green, indicating that the result is not NaN unless the operands are NaN.

Combining the colors for the two calls to `func`, the check is orange.

In the example, the option `-check-nan warn-first` was used.

NaN in Conversion from Floating Point to Integers

```
void func() {
    double x= 1.0/0.0;
    double y= x-x;
    int z = y;
}
```

In this example, the **Invalid operation on floats** check detects the assignment of NaN to an integer variable `z`.

The check is enabled if you specify that non-finite floats must be considered in the verification. The check blocks further verification on the same execution path irrespective of whether you allow, forbid or ask for warnings on non-finite floats.

Result Information

Group: Numerical
Language: C | C++
Acronym: INVALID_FLOAT_OP

See Also

`Overflow` | `Subnormal float`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Introduced in R2016a

Invalid shift operations

Shift operations are invalid

Description

This check on shift operations on a variable `var` determines:

- Whether the shift amount is larger than the range allowed by the type of `var`.
- If the shift is a left shift, whether `var` is negative.

Diagnosing This Check

"Review and Fix Invalid Shift Operations Checks"

Examples

Shift amount outside bounds

```
#include <stdlib.h>
#define shiftAmount 32
enum shiftType {
 SIGNED_LEFT,
 SIGNED_RIGHT,
 UNSIGNED_LEFT,
 UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
  enum shiftType myShiftType = getShiftType();
  int signedInteger = 1;
  unsigned int unsignedInteger = 1;
  switch(myShiftType) {
  case SIGNED_LEFT:
```

```
    signedInteger = signedInteger << shiftAmount;
    break;
  case SIGNED_RIGHT:
    signedInteger = signedInteger >> shiftAmount;
    break;
  case UNSIGNED_LEFT:
    unsignedInteger = unsignedInteger << shiftAmount;
    break;
  case UNSIGNED_RIGHT:
    unsignedInteger = unsignedInteger >> shiftAmount;
    break;
  }
}
```

In this example, the shift amount `shiftAmount` is outside the allowed range for both signed and unsigned `int`. Therefore the **Invalid shift operations** check produces a red error.

Correction — Keep shift amount within bounds

One possible correction is to keep the shift amount in the range 0..31 for unsigned integers and 0...30 for signed integers. This correction works if the size of `int` is 32 on the target processor.

```
#include <stdlib.h>
#define shiftAmountSigned 30
#define shiftAmount 31
enum shiftType {
 SIGNED_LEFT,
 SIGNED_RIGHT,
 UNSIGNED_LEFT,
 UNSIGNED_RIGHT
};

enum shiftType getShiftType();

void main() {
  enum shiftType myShiftType = getShiftType();
  int signedInteger = 1;
  unsigned int unsignedInteger = 1;
  switch(myShiftType) {

  case SIGNED_LEFT:
    signedInteger = signedInteger << shiftAmountSigned;
```

```
    break;

  case SIGNED_RIGHT:
    signedInteger = signedInteger >> shiftAmountSigned;
    break;

  case UNSIGNED_LEFT:
    unsignedInteger = unsignedInteger << shiftAmount;
     break;

  case UNSIGNED_RIGHT:
    unsignedInteger = unsignedInteger >> shiftAmount;
    break;
  }
}
```

Left operand of left shift is negative

```
void main(void) {
  int x = -200;
  int y;
  y = x << 1;
}
```

In this example, the left operand of the left shift operation is negative.

Correction — Use Polyspace analysis option

You can use left shifts on negative numbers and not produce a red **Invalid shift operations** error. To allow such left shifts, on the **Configuration** pane, under **Check Behavior**, select **Allow negative operand for left shifts**.

```
void main(void) {
  int x = -200;
  int y;
  y = x << 1;
}
```

Left operand of left shift may be negative

```
short getVal();
```

```
int foo(void) {
  long lvar;
  short svar1, svar2;

  lvar = 0;
  svar1 = getVal();
  svar2 = getVal();

  lvar =  (svar1 - svar2) << 10;
  if (svar1 < svar2)  {
    return 1;
  } else {
    return 0;
  }
}

int main(void) {
  return foo();
}
```

In this example, if `svar1 < svar2`, the left operand of << can be negative. Therefore the **Shift operations** check on << is orange. Following an orange check, execution paths containing the error get truncated. Therefore, following the orange **Invalid shift operations** check, Polyspace assumes that `svar1 >= svar2`. The branch of the statement, `if(svar1 < svar2)`, is unreachable.

Check Information

Group: Numerical
Language: C | C++
Acronym: SHF

See Also

Topics

“Interpret Polyspace Code Prover Access Results”
“Code Prover Analysis Following Red and Orange Checks”

Invalid use of standard library routine

Standard library function is called with invalid arguments

Description

This check on a standard library function call determines whether the function is called with valid arguments.

The check works differently for memory routines, floating point routines or string routines because their arguments can be invalid in different ways. For more information on each type of routines, see the following examples.

Diagnosing This Check

"Review and Fix Invalid Use of Standard Library Routine Checks"

Examples

Invalid use of standard library float routine

```
#include <assert.h>
#include <math.h>

#define LARGE_EXP 710

enum operation {
    ASIN,
    ACOS,
    TAN,
    SQRT,
    LOG,
    EXP,
};

enum operation getOperation(void);
```

```
double getVal(void);

void main() {
    enum operation myOperation = getOperation();
    double myVal=getVal(), res;
    switch(myOperation) {
    case ASIN:
        assert( myVal <- 1.0 || myVal > 1.0);
        res = asin(myVal);
        break;
    case ACOS:
        assert( myVal < -1.0 || myVal > 1.0);
        res = acos(myVal);
        break;
    case SQRT:
        assert( myVal < 0.0);
        res = sqrt(myVal);
        break;
    case LOG:
        assert(myVal <= 0.0);
        res = log(myVal);
        break;
    case EXP:
        assert(myVal > LARGE_EXP);
        res = exp(myVal);
        break;
    }
}
```

In this example, following each `assert` statement, Polyspace considers that `myVal` contains only those values that make the `assert` condition true. For example, following `assert(myVal < 1.0);`, Polyspace considers that `myVal` contains values less than 1.0.

When `myVal` is used as argument in a standard library function, its values are invalid for the function. Therefore, the **Invalid use of standard library routine** check produces a red error.

To learn more about the specifications of this check for floating point routines, see “Invalid Use of Standard Library Floating Point Routines”.

Invalid use of standard library memory routine

```
#include <string.h>
#include <stdio.h>
```

```
char* Copy_First_Six_Letters(void) {
  char str1[10],str2[5];
  printf("Enter string:\n");
  scanf("%s",str1);
  memcpy(str2,str1,6);
  return str2;
}

int main(void) {
  (void*)Copy_First_Six_Letters();
  return 0;
}
```

In this example, the size of string `str2` is 5, but 6 characters of string `str1` are copied into `str2` using the `memcpy` function. Therefore, the **Invalid use of standard library routine** check on the call to `memcpy` produces a red error.

Correction — Call function with valid arguments

One possible correction is to adjust the size of `str2` so that it accommodates the characters copied with the `memcpy` function.

```
#include <string.h>
#include <stdio.h>

char* Copy_First_Six_Letters(void) {
  char str1[10],str2[6];
  printf("Enter string:\n");
  scanf("%s",str1);
  memcpy(str2,str1,6);
  return str2;
}

int main(void) {
  (void*)Copy_First_Six_Letters();
  return 0;
}
```

Invalid use of standard library string routine

```
#include <stdio.h>
#include <string.h>
```

```
char* Copy_String(void)
{
  char *res;
  char gbuffer[5],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  return(res);
}

int main(void) {
  (void*)Copy_String();
}
```

In this example, the string `text` is larger in size than `gbuffer`. Therefore, when you copy `text` into `gbuffer`. the **Invalid use of standard library routine** check on the call to `strcpy` produces a red error.

Correction — Call function with valid arguments

One possible correction is to declare the destination string `gbuffer` with equal or larger size than the source string `text`.

```
#include <stdio.h>
#include <string.h>

char* Copy_String(void)
{
  char *res;
  char gbuffer[20],text[20]="ABCDEFGHIJKL";
  res=strcpy(gbuffer,text);
  return(res);
}

int main(void) {
  (void*)Copy_String();
}
```

Check Information

Group: Other
Language: C | C++
Acronym: STD_LIB

See Also

Topics

“Interpret Polyspace Code Prover Access Results”
“Code Prover Analysis Following Red and Orange Checks”
“Using memset and memcpy”

Non-initialized local variable

Local variable is not initialized before being read

Description

This check occurs for every local variable read. It determines whether the variable being read is initialized.

Diagnosing This Check

"Review and Fix Non-initialized Local Variable Checks"

Examples

Non-initialized variable used on right side of assignment operator

```
#include <stdio.h>

void main(void) {
   int sum;
   for(int i=1;i <= 10; i++)
      sum+=i;
   printf("The sum of the first 10 natural numbers is %d.", sum);
 }
```

The statement `sum+=i;` is the shorthand for `sum=sum+i;`. Because `sum` is used on the right side of an expression before being initialized, the **Non-initialized local variable** check returns a red error.

Correction — Initialize variable before using on right side of assignment

One possible correction is to initialize `sum` before the `for` loop.

```
#include <stdio.h>

void main(void) {
   int sum=0;
   for(int i=1;i <= 10; i++)
      sum+=i;
   printf("The sum of the first 10 natural numbers is %d.", sum);
 }
```

Non-initialized variable used with relational operator

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count,sum=0,term;

    while( count <= 10  && sum <1000) {
       count++;
       term = getTerm();
       if(term > 0 && term <= 1000) sum += term;
      }

    printf("The sum of 10 terms is %d.", sum);
 }
```

In this example, the variable `count` is not initialized before the comparison `count <= 10`. Therefore, the **Non-initialized local variable** check returns a red error.

Correction — Initialize variable before using with relational operator

One possible correction is to initialize `count` before the comparison `count <= 10`.

```
#include <stdio.h>

int getTerm();

void main(void) {
    int count=1,sum=0,term;

    while( count <= 10  && sum <1000) {
       count++;
       term = getTerm();
```

```
       if(term > 0 && term <= 1000) sum+= term;
      }

    printf("The sum of 10 terms is %d.", sum);
 }
```

Non-initialized variable passed to function

```
#include <stdio.h>

int getShift();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}

void main(void) {
    int initVal;
    printf("The result of a shift is %d",shift(initVal));
}
```

In this example, `initVal` is not initialized when it is passed to `shift()`. Therefore, the **Non-initialized local variable** check returns a red error. Because of the red error, Polyspace does not verify the operations in `shift()`.

Correction — Initialize variable before passing to function

One possible correction is to initialize `initVal` before passing to `shift()`. `initVal` can be initialized through an input function. To avoid an overflow, the value returned from the input function must be within bounds.

```
#include <stdio.h>

int getShift();
int getInit();
int shift(int var) {
    int shiftVal = getShift();
    if(shiftVal > 0 && shiftVal < 1000)
        return(var+shiftVal);
    return 1000;
}
```

```
void main(void) {
   int initVal=getInit();
   if(initVal >0 && initVal < 1000)
     printf("The result of a shift is %d",shift(initVal));
   else
     printf("Value must be between 0 and 1000.");
 }
```

Non-initialized array element

```
#include <stdio.h>
#define arrSize 19

void main(void)
{
  int arr[arrSize],indexFront, indexBack;
  for(indexFront = 0,indexBack = arrSize - 1;
    indexFront < arrSize/2;
    indexFront++, indexBack--) {
    arr[indexFront] = indexFront;
    arr[indexBack] = arrSize - indexBack - 1;
  }
  printf("The array elements are: \n");
  for(indexFront = 0; indexFront < arrSize; indexFront++)
    printf("Element[%d]: %d", indexFront, arr[indexFront]);
 }
```

In this example, in the first `for` loop:

- `indexFront` runs from 0 to 8.
- `indexBack` runs from 18 to 10.

Therefore, `arr[9]` is not initialized. In the second `for` loop, when `arr[9]` is passed to `printf`, the **Non-initialized local variable** check returns an error. The error is orange because the check returns an error only in one of the loop runs.

Due to the orange error in one of the loop runs, a red **Non-terminating loop** error appears on the second `for` loop.

Correction — Initialize variable before passing to function

One possible correction is to keep the first `for` loop intact and initialize `arr[9]` outside the `for` loop.

```
#include <stdio.h>
#define arrSize 19

void main(void)
{
  int arr[arrSize],indexFront, indexBack;
  for(indexFront = 0,indexBack = arrSize - 1;
    indexFront < arrSize/2;
    indexFront++, indexBack--) {
    arr[indexFront] = indexFront;
    arr[indexBack] = arrSize - indexBack - 1;
  }
  arr[indexFront] = indexFront;
  printf("The array elements are: \n");
  for(indexFront = 0; indexFront < arrSize; indexFront++)
    printf("Element[%d]: %d", indexFront, arr[indexFront]);
}
```

Non-initialized structure

```
typedef struct S {
   int integerField;
   char characterField;
}S;

void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
  S myStruct;
  operateOnStructure(myStruct);
  operateOnStructureField(myStruct.integerField);
}
```

In this example, the structure `myStruct` is not initialized. Therefore, when the structure `myStruct` is passed to the function `operateOnStructure`, a **Non-initialized local variable** check on the structure appears red.

Correction— Initialize structure

One possible correction is to initialize the structure `myStruct` before passing it to a function.

```
typedef struct S {
   int integerField;
   char characterField;
}S;

void operateOnStructure(S);
void operateOnStructureField(int);

void main() {
  S myStruct = {0,' '};
  operateOnStructure(myStruct);
  operateOnStructureField(myStruct.integerField);
}
```

Partially initialized structure — All used fields initialized

```
typedef struct S {
   int integerField;
   char characterField;
   double doubleField;
}S;

int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);

void printFields(S s) {
 printIntegerField(s.integerField);
 printCharacterField(s.characterField);
}

void main() {
  S myStruct;

  myStruct.integerField = getIntegerField();
  myStruct.characterField = getCharacterField();
  printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is green because:

- The fields `integerField` and `characterField` that are used are both initialized.
- Although the field `doubleField` is not initialized, there is no read or write operation on the field `doubleField` in the code.

To determine which fields are checked for initialization:

1. Select the check on the **Results List** pane or **Source** pane.
2. View the message on the **Result Details** pane.

Partially initialized structure — Some used fields initialized

```
typedef struct S {
   int integerField;
   char characterField;
   double doubleField;
}S;

int getIntegerField(void);
char getCharacterField(void);

void printIntegerField(int);
void printCharacterField(char);
void printDoubleField(double);

void printFields(S s) {
 printIntegerField(s.integerField);
 printCharacterField(s.characterField);
 printDoubleField(s.doubleField);
}

void main() {
  S myStruct;

  myStruct.integerField = getIntegerField();
  myStruct.characterField = getCharacterField();
  printFields(myStruct);
}
```

In this example, the **Non-initialized local variable** check on `myStruct` is orange because:

- The fields `integerField` and `characterField` that are used are both initialized.

- The field `doubleField` is not initialized and there is a read operation on `doubleField` in the code.

To determine which fields are checked for initialization:

1. Select the check on the **Results List** pane or **Source** pane.
2. View the message on the **Result Details** pane.

Potential Initialization in Stubbed Functions

```
int var;

struct s {
    int data;
    int pos;
};

void init_struct(struct s*);
void init_int(int*);

void main (void) {
    struct s s_obj;
    int loc_var;

    init_int(&loc_var);
    var =  loc_var;

    s_obj.pos = 1;
    init_struct(&s_obj);
    var =  s_obj.data;
    var = s_obj.pos;

}
```

In this example, the definition of functions `init_int` and `init_struct` are not provided. The verification uses stubs for these functions. The verification makes these assumptions for stubbed functions:

- If a variable is uninitialized or partially initialized, the function stubs can leave the variables uninitialized or partially initialized. The orange **Non-initialized local variable** checks on `loc_var` and `s_obj.data` indicate this potentially non-initialized state.

- If a variable is previously initialized, the function stubs can write a different value to the variable but cannot uninitialize the variable. The green **Non-initialized local variable** check on `s_obj.pos` indicate this initialized state. If you place your cursor on `pos` in `var = s_obj.pos`, you see that `pos` can take any value allowed for an `int` variable.

Check Information

Group: Data flow
Language: C | C++
Acronym: NIVL

See Also

`Non-initialized pointer` | `Non-initialized variable`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Non-initialized pointer

Pointer is not initialized before being read

Description

This check occurs for every pointer read. It determines whether the pointer being read is initialized.

Diagnosing This Check

"Review and Fix Non-initialized Pointer Checks"

Examples

Non-initialized pointer passed to function

```
int assignValueToAddress(int *ptr) {
  *ptr = 0;
}

void main() {
 int* newPtr;
 assignValueToAddress(newPtr);
}
```

In this example, `newPtr` is not initialized before it is passed to `assignValueToAddress()`.

Correction — Initialize pointer before passing to function

One possible correction is to assign `newPtr` an address before passing to `assignValueToAddress()`.

```
int assignValueToAddress(int *ptr) {
```

```
  *ptr = 0;
}

void main() {
 int val;
 int* newPtr = &val;
 assignValueToAddress(newPtr);
}
```

Non-initialized pointer to structure

```
#include <stdlib.h>
#define stackSize 25

typedef struct stackElement {
  int value;
  int *prev;
}stackElement;

int input();

void main() {
 stackElement *stackTop;

 for (int count = 0; count < stackSize; count++) {
    if(stackTop!=NULL) {
       stackTop -> value = input();
       stackTop -> prev = (int*)stackTop;
    }
    stackTop = (stackElement*)malloc(sizeof(stackElement));
 }
}
```

In this example, in the first run of the `for` loop, `stackTop` is not initialized and does not point to a valid address. Therefore, the **Non-initialized pointer** check on `stackTop!=NULL` returns a red error.

Correction — Initialize pointer before dereference

One possible correction is to initialize `stackTop` through `malloc()` before the check `stackTop!=NULL`.

```
#include <stdlib.h>
#define stackSize 25
```

```
typedef struct stackElement {
  int value;
  int *prev;
}stackElement;

int input();

void main() {
 stackElement *stackTop;

 for (int count = 0; count < stackSize; count++) {
    stackTop = (stackElement*)malloc(sizeof(stackElement));
    if(stackTop!=NULL) {
       stackTop->value = input();
       stackTop->prev = (int*)stackTop;
    }
 }
}
```

Non-initialized `char*` pointer used to store string

```
#include <stdio.h>

void main() {
 char *str;
 scanf("%s",str);
}
```

In this example, `str` does not point to a valid address. Therefore, when the `scanf` function reads a string from the standard input to `str`, the **Non-initialized pointer** check returns a red error.

Correction — Use `char` array instead of `char*` pointer

One possible correction is to declare `str` as a `char` array. This declaration assigns an address to the `char*` pointer associated with the array name `str`. You can then use the pointer as input to `scanf`.

```
#include <stdio.h>

void main() {
 char str[10];
```

```
 scanf("%s",str);
}
```

Non-initialized array of char* pointers used to store variable-size strings

```
#include <stdio.h>

void assignDataBaseElement(char** str) {
 scanf("%s",*str);
}

void main() {
 char *dataBase[20];

 for(int count = 1; count < 20 ; count++) {
    assignDataBaseElement(&dataBase[count]);
    printf("Database element %d : %s",count,dataBase[count]);
 }
}
```

In this example, `dataBase` is an array of `char*` pointers. In each run of the `for` loop, an element of `dataBase` is passed via pointers to the function `assignDataBaseElement()`. The element passed is not initialized and does not contain a valid address. Therefore, when the element is used to store a string from standard input, the **Non-initialized pointer** check returns a red error.

Correction — Initialize char* pointers through calloc

One possible correction is to initialize each element of `dataBase` through the `calloc()` function before passing it to `assignDataBaseElement()`. The initialization through `calloc()` allows the `char` pointers in `dataBase` to point to strings of varying size.

```
#include <stdio.h>
#include <stdlib.h>

void assignDataBaseElement(char** str) {
 scanf("%s",*str);
}
int inputSize();

void main() {
 char *dataBase[20];
```

```
 for(int count = 1; count < 20 ; count++) {
    dataBase[count] = (char*)calloc(inputSize(),sizeof(char));
    assignDataBaseElement(&dataBase[count]);
    printf("Database element %d : %s",count,dataBase[count]);
 }
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: NIP

See Also

`Non-initialized local variable` | `Non-initialized variable`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Non-initialized variable

Variable other than local variable is not initialized before being read

Description

This check occurs when you read variables that are not local (global or static variables). It determines whether the variable being read is initialized.

Note By default, Polyspace considers that global variables are initialized. The verification checks global variables only if you prevent this default initialization.

Diagnosing This Check

"Review and Fix Non-initialized Variable Checks"

Examples

Non-initialized global variable

```
int globVar;
int getVal();

void main() {
 int val = getVal();
 if(val>=0 && val<= 100)
   globVar += val;
}
```

In this example, `globVar` does not have an initial value when incremented. Therefore, the **Non-initialized variable** check produces a red error.

The example uses the option to prevent default initialization of global variables.

Correction — Initialize global variable before use

One possible correction is to initialize the global variable `globVar` before use.

```
int globVar;
int getVal();

void main() {
 int val = getVal();
 globVar = 0;
 if(val>=0 && val<= 100)
   globVar += val;
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: NIV

See Also

`Non-initialized local variable` | `Non-initialized pointer`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Non-terminating call

Called function does not return to calling context

Description

This check on a function call appears when the following conditions hold:

- The called function does not return to its calling context. The call leads to a definite run-time error or a process termination function like `exit()` in the function body.
- There are other calls to the same function that do not lead to a definite error or process termination function in the function body.

When only a fraction of calls to a function lead to a definite error, this check helps identify those function calls. In the function body, even though a definite error occurs, the error appears in orange instead of red because the verification results in a function body are aggregated over all function calls. To indicate that a definite error has occurred, a red **Non-terminating call** check is shown *on the function call* instead.

Otherwise, if all the calls to a function lead to a definite error or process termination function in the function body, the **Non-terminating call** error is not displayed. The error appears in red in the function body and a dashed red underline appears on the function calls. However, following the function call, like other red errors, Polyspace does not analyze the remaining code in the same scope as the function call.

You can navigate directly from the function call to the operation causing the run-time error in the function body.

- To find the source of error, on the **Source** pane, place your cursor on the loop keyword and view the tooltip.
- Navigate to the source of error in the function body. Right-click the function call and select **Go to Cause** if the option exists.

  If the error is the result of multiple causes, the option takes you to the first cause in the function body. Multiple causes can occur, for instance, when some values of a function argument trigger one specific error and other values trigger other errors.

Diagnosing This Check

“Review and Fix Non-Terminating Call Checks”

Examples

Dashed red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
  return(num/den);
}

void main() {
  int i,j;
  i=2;
  j=0;
  printf("%.2f",ratio(i,j));
}
```

In this example, a red **Division by zero** error appears in the body of `ratio`. This **Division by zero** error in the body of `ratio` causes a dashed red underline on the call to `ratio`.

Red underline on function call

```
#include<stdio.h>
double ratio(int num, int den) {
  return(num/den);
}

int inputCh();

void main() {
  int i,j,ch=inputCh();
  i=2;

  if(ch==1)  {
    j=0;
    printf("%.2f",ratio(i,j));
```

```
  }
  else {
    j=2;
    printf("%.2f",ratio(i,j));
  }
}
```

In this example, there are two calls to `ratio`. In the first call, a **Division by zero** error occurs in the body of `ratio`. In the second call, Polyspace does not find errors. Therefore, combining the two calls, an orange **Division by zero** check appears in the body of `ratio`. A red **Non-terminating call** check on the first call indicates the error.

Red underline on call through function pointer

```
typedef void (*f)(void);
// function pointer type

void f1(void) {
  int x;
  x++;
}

void f2(void) { }
void f3(void) { }

f fptr_array[3] = {f1,f2,f3};
unsigned char getIndex(void);

void main(void) {
 unsigned char index = getIndex() % 3;
 // Index is between 0 and 2

 fptr_array[index]();
 fptr_array[index]();
}
```

In this example, because `index` can lie between 0 and 2, the first `fptr_array[index]()` can call `f1`, `f2` or `f3`. If `index` is zero, the statement calls `f1`. `f1` contains a red **Non-initialized local variable** error, therefore, a dashed red error appears on the function call. Unlike other red errors, the verification continues.

After this statement, the software considers that `index` is either 1 or 2. An error does not occur on the second `fptr_array[index]()`.

Check Information

Group: Control flow
Language: C | C++
Acronym: NTC

See Also

`Non-terminating loop`

Topics

"Identify Function Call with Run-Time Error"
"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Non-terminating loop

Loop does not terminate or contains an error

Description

This check on a loop determines if the loop has one of the following issues:

- The loop definitely does not terminate.

  The check appears only if Polyspace cannot detect an exit path from the loop. For example, if the loop appears in a function and the loop termination condition is met for some function inputs, the check does not appear, even though the condition might not be met for some other inputs.
- The loop contains a definite error in one its iterations.

  Even though a definite error occurs in one loop iteration, because the verification results in a loop body are aggregated over all loop iterations, the error shows as an orange check in the loop body. To indicate that a definite failure has occurred, a red **Non-terminating loop** check is shown on the loop command.

Unlike other checks, this check appears only when a definite error occurs. In your verification results, the check is always red. If the error occurs only in some cases, for instance, if the loop bound is variable and causes an issue only for some values, the check does not appear. Instead, the loop command is shown in dashed red with more information in the tooltip.

The check also does not appear if both conditions are true:

- The loop has a trivial predicate such as `for(;;)` or `while(1)`.
- The loop has an empty body, or a body without an exit statement such as `break`, `goto`, `return` or an exception.

Instead, the loop statement is underlined with red dashes. If you place your cursor on the loop statement, you see that the verification considers the loop as intentional. If you deliberately introduce infinite loops, for instance, to emulate cyclic tasks, you do not have to justify red checks.

Using this check, you can identify the operation in the loop that causes the run-time error.

- To find the source of error, on the **Source** pane, place your cursor on the function call and view the tooltip.
- For loops with fewer iterations, you can navigate to the source of error in the loop body. Select the loop to see the full history of the result. Alternatively, right-click the loop keyword and select **Go to Cause** if the option exists.

Diagnosing This Check

"Review and Fix Non-Terminating Loop Checks"

Examples

Loop does not terminate

```
#include<stdio.h>

void main() {
  int i=0;
  while(i<10) {
    printf("%d",i);
  }
}
```

In this example, in the `while` loop, `i` does not increase. Therefore, the test `i<10` never fails.

Correction — Ensure Loop Termination

One possible correction is to update `i` such that the test `i<10` fails after some loop iterations and the loop terminates.

```
#include<stdio.h>

void main() {
  int i=0;
  while(i < 10) {
    printf("%d",i);
    i++;
  }
}
```

Loop contains an out of bounds array index error

```
void main() {
  int arr[20];
  for(int i=0; i<=20; i++) {
    arr[i]=0;
  }
}
```

In this example, the last run of the `for` loop contains an **Out of bounds array index** error. Therefore, the **Non-terminating loop** check on the `for` loop is red. A tooltip appears on the `for` loop stating the maximum number of iterations including the one containing the run-time error.

Correction — Avoid loop iteration containing error

One possible correction is to reduce the number of loop iterations so that the **Out of bounds array index** error does not occur.

```
void main() {
  int arr[20];
  for(int i=0; i<20; i++) {
    arr[i]=0;
  }
}
```

Loop contains an error in function call

```
int arr[4];

void assignValue(int index) {
  arr[index] = 0;
}

void main() {
  for(int i=0;i<=4;i++)
    assignValue(i);
}
```

In this example, the call to function `assignValue` in the last `for` loop iteration contains an error. Therefore, although an error does not show in the `for` loop body, a red **Non-terminating loop** appears on the loop itself.

Correction — Avoid loop iteration containing error

One possible correction is to reduce the number of loop iterations so the error in the call to `assignValue` does not occur.

```
int arr[4];

void assignValue(int index) {
  arr[index] = 0;
}

void main() {
 for(int i=0;i<4;i++)
   assignValue(i);
}
```

Loop contains an overflow error

```
#define MAX 1024
void main() {
 int i=0,val=1;
 while(i<MAX) {
   val*=2;
   i++;
  }
}
```

In this example, an **Overflow** error occurs in iteration number 31. Therefore, the **Non-terminating loop** check on the `while` loop is red. A tooltip appears on the `while` loop stating the maximum number of iterations including the one containing the run-time error.

Correction — Reduce loop iterations

One possible correction is to reduce the number of loop iterations so that the overflow does not occur.

```
#define MAX 30
void main() {
 int i=0,val=1;
 while(i<MAX) {
   val*=2;
   i++;
```

```
  }
}
```

Check Information

Group: Control flow
Language: C | C++
Acronym: NTL

See Also

`Non-terminating call`

Topics

"Identify Loop Operation with Run-Time Error"
"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Null this-pointer calling method

`this` pointer is null during member function call

Description

This check on a `this` pointer dereference determines whether the pointer is `NULL`.

Diagnosing This Check

"Review and Fix Null This-pointer Calling Method Checks"

Examples

Pointer to object is NULL during member function call

```
#include <stdlib.h>
class Company {
 public:
  Company(int initialNumber):numberOfClients(initialNumber) {}
  void addNewClient() {
    numberOfClients++;
  }
 protected:
  int numberOfClients;
};

void main() {
 Company* myCompany = NULL;
 myCompany->addNewClient();
}
```

In this example, the pointer `myCompany` is initialized to `NULL`. Therefore when the pointer is used to call the member function `addNewClient`, the **Null this-pointer calling method** produces a red error.

Correction — Initialize pointer with valid address

One possible correction is to initialize `myCompany` with a valid memory address using the `new` operator.

```
#include <stdlib.h>
class Company {
 public:
  Company(int initialNumber):numberOfClients(initialNumber) {}
  void addNewClient() {
    numberOfClients++;
  }
 protected:
  int numberOfClients;
};

void main() {
 Company* myCompany = new Company(0);
 myCompany->addNewClient();
}
```

Check Information

Group: C++
Language: C++
Acronym: NNT

See Also

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Out of bounds array index

Array is accessed outside range

Description

This check on an array element access determines whether the element is outside the array range.

Diagnosing This Check

"Review and Fix Out of Bounds Array Index Checks"

Examples

Array index is equal to array size

```
#include <stdio.h>

void fibonacci(void)
{
  int i;
  int fib[10];

  for (i = 0; i < 10; i++)
  {
    if (i < 2)
      fib[i] = 1;
    else
      fib[i] = fib[i-1] + fib[i-2];
    }

  printf("The 10-th Fibonacci number is %i .\n", fib[i]);
}

int main(void) {
```

```
  fibonacci();
}
```

In this example, the array `fib` is assigned a size of 10. An array index for `fib` has allowed values of [0,1,2,...,9]. The variable `i` has a value 10 when it comes out of the `for`-loop. Therefore, when the `printf` statement attempts to access `fib[10]` through `i`, the **Out of bounds array index** check produces a red error.

The check also produces a red error if `printf` uses `*(fib+i)` instead of `fib[i]`.

Correction — Keep array index less than array size

One possible correction is to print `fib[i-1]` instead of `fib[i]` after the `for`-loop.

```
#include <stdio.h>

void fibonacci(void)
{
  int i;
  int fib[10];

  for (i = 0; i < 10; i++)
  {
    if (i < 2)
      fib[i] = 1;
    else
      fib[i] = fib[i-1] + fib[i-2];
    }

  printf("The 10-th Fibonacci number is %i .\n", fib[i-1]);
}

int main(void) {
  fibonacci();
}
```

Check Information

Group: Static memory
Language: C | C++
Acronym: OBAI

See Also

`Illegally dereferenced pointer`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Overflow

Arithmetic operation causes overflow

Description

This check on an arithmetic operation determines whether the result overflows. The result of this check depends on whether you allow nonfinite float results such as infinity and NaN.

The result of the check also depends on the float rounding mode you specify. By default, the rounding mode is `to-nearest`. See `Float rounding mode (-float-rounding-mode)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Nonfinite Floats Not Allowed

By default, nonfinite floats are not allowed. When the result of an operation falls outside the allowed range, an overflow occurs. The check is:

- Red, if the result of the operation falls outside the allowed range.
- Orange, if the result of the operation falls outside the allowed range on some of the execution paths.
- Green, if the result of the operation does not fall outside the allowed range.

To fine tune the behavior of the overflow check, use these options and specify argument `forbid`, `allow`, or `warn-with-wrap-around`:

- `Overflow mode for unsigned integer (-unsigned-integer-overflows)`
- `Overflow mode for signed integer (-signed-integer-overflows)`

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

The operand data types determine the allowed range for the arithmetic operation. If the operation involves two operands, the verification uses the ANSI C conversion rules to determine a common data type. This common data type determines the allowed range.

Nonfinite Floats Allowed

If you enable a verification mode that incorporates infinities and specify that the verification must warn about operations that produce infinities, the check is:

- Red, if the operation produces infinity on all execution paths that the software considers, and the operands themselves are not infinite.
- Orange, if the operation produces infinity on some of the execution paths when the operands themselves are not infinite.
- Green, if the operation does not produce infinity unless the operands themselves are infinite.

If you specify that the verification must forbid operations that produce infinities, the check color depends on the result of the operation only. The color does not depend on the operands.

To enable this verification mode, use these options:

- `Consider non finite floats (-allow-non-finite-floats)`
- `Infinities (-check-infinite)`: Use argument `warn` or `forbid`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

"Review and Fix Overflow Checks"

Examples

Integer Overflow

```
void main() {
  int i=1;
  i = i << 30; //i = 2^30
  i = 2*i-2;
}
```

In this example, the operation `2*i` results in a value 2^{31}. The **Overflow** check on the multiplication produces a red error because the maximum value that the type `int` can hold on a 32–bit target is 2^{31}`-1`.

Overflow Due to Left Shift on Signed Integers

```
void main(void)
 {
   unsigned int i;

   i = 1090654225 << 1;
 }
```

In this example, an **Overflow** error occurs due to integer promotion.

Float Overflow

```
#include <float.h>

void main() {
 float val = FLT_MAX;
 val = val * 2 + 1.0;
}
```

In this example, `FLT_MAX` is the maximum value that `float` can represent on a 32-bit target. Therefore, the operation `val * 2` results in an **Overflow** error.

Overflow on Casts from Negative Floats to Unsigned Integers

```
void func(void) {
   float fVal = -2.0f;
   unsigned int iVal = (unsigned int)fVal;
}
```

In this example, a red **Overflow** check appears on the cast from `float` to `unsigned int`. According to the C99 Standard (footnote to paragraph 6.3.1.4), the range of values that can be converted from floating-point values to unsigned integers while keeping the code portable is `(-1, MAX + 1)`. For floating-point values outside this range, the conversion to unsigned integers is not well-defined. Here, `MAX` is the maximum number that can be stored by the unsigned integer type.

Even if a run-time error does not occur when you execute the code on your target, the cast might fail on another target.

Correction — Cast to Signed Integer First

One possible solution is to cast the floating-point value to a signed integer first. The signed integer can then be cast to an unsigned integer type. For these casts, the conversion rules are well-defined.

```
void func(void) {
   float fVal = -2.0f;
   int iValTemp = (int)fVal;
   unsigned int iVal = (unsigned int)iValTemp;
}
```

Negative Overflow

```
#define FLT_MAX 3.40282347e+38F

void float_negative_overflow() {
   float min_float = -FLT_MAX;
   min_float = -min_float * min_float;
}
```

In `float_negative_overflow`, `min_float` contains the most negative number that the type `float` can represent. Because the operation `-min_float * min_float` produces a number that is more negative than this number, the type `float` cannot represent it. The **Overflow** check produces a red error.

Overflows on Unsigned Bit Fields

```
#include <stdio.h>

struct
{
  unsigned int dayOfWeek : 2;
} Week;

void main()
{
  Week.dayOfWeek = 2;
  Week.dayOfWeek = 3;
```

```
  Week.dayOfWeek = 4;
}
```

In this example, `dayOfWeek` occupies 2 bits. It can take values in `[0,3]` because it is an unsigned integer. When you assign 4 to `dayOfWeek`, the **Overflow** check is red.

To detect overflows on signed and unsigned integers, on the **Configuration** pane, under **Check Behavior**, select `forbid` or `warn-with-wrap-around` for **Overflow mode for signed integer** and **Overflow mode for unsigned integer** .

Overflows on Signed and enum Bit Fields

```
enum tBit {
  ZERO = 0x00,
  ONE = 0x01 ,
  TWO = 0x02
};

struct twoBit
{
  enum tBit myBit:2;
} myBitField;

void main()
{
  myBitField.myBit = ZERO;
  myBitField.myBit = ONE;
  myBitField.myBit = TWO;
}
```

In this example, being an `enum` variable, `myBit` is implemented through a signed integer according to the ANSI C90 standard. `myBit` occupies 2 bits. It can take values in `[-2,1]` because it is a signed integer. When you assign 2 to `myBit`, the **Overflow** check is red.

To detect overflows on signed integers alone, on the **Configuration** pane, under **Check Behavior**, select `forbid` or `warn-with-wrap-around` for **Overflow mode for signed integer** and `allow` for **Overflow mode for unsigned integer**.

Nonfinite Floats: Infinity Detected with Red Check

Results in `forbid` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the `/` operation is not infinite but the result is infinity. The **Overflow** check on the `-` operation is red. In the `forbid` mode, the verification stops after the red check. For instance, a **Non-initialized local variable** check does not appear on `x` in the `return` statement. If you do not turn on the option **Allow non finite floats**, a **Division by zero** check appears because infinities are not allowed.

Results in `warn-first` mode:

```
double func(void) {
    double x=1.0/0.0;
    return x;
}
```

In this example, both the operands of the `/` operation are not infinite but the result is infinity. The **Overflow** check on the `-` operation is red. The red checks in `warn-first` mode are different from red checks for other check types. The verification does not stop after the red check. For instance, a green **Non-initialized local variable** check appears on `x` in the `return` statement. In the verification result, if you place your cursor on `x`, you see that it has the value `Inf`.

Nonfinite Floats: Infinity Detected with Orange Check

Results in `forbid` mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible `double` values. For instance, `arg1` can be nonzero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. Therefore, an orange **Overflow** check appears on the division operation. Following the check, the verification terminates the execution thread that results in infinity. The verification assumes that `arg2` cannot be zero following the orange check. The **Overflow** check on the second division operation `ratio2=arg1/arg2` is green.

Results in `warn-first` mode:

```
void func(double arg1, double arg2) {
    double ratio1=arg1/arg2;
    double ratio2=arg1/arg2;
}
```

In this example, the values of `arg1` and `arg2` are unknown to the verification. The verification assumes that `arg1` and `arg2` can have all possible `double` values. For instance, `arg1` can be non-zero and `arg2` can be zero and the result of `ratio1=arg1/arg2` can be infinity. An orange **Overflow** check appears on the division operation. The orange checks in `warn-first` mode are different from orange checks for other check types. Following the check, the verification does not terminate the execution thread that results in infinity. The verification retains the zero value of `arg2` following the orange check. Therefore, the **Overflow** check on the second division operation `ratio2=arg1/arg2` is also orange.

Check Information

Group: Numerical
Language: C | C++
Acronym: OVFL

See Also

`Invalid operation on floats` | `Subnormal float`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Return value not initialized

C function does not return value when expected

Description

This check determines whether a function with a return type other than `void` returns a value. This check appears on every function call.

Diagnosing This Check

"Review and Fix Return Value Not Initialized Checks"

Examples

Function does not return value for given input

```
#include <stdio.h>
int input(void);
int inputRep(void);

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch <= 0)
    ans = reply(0);
  else
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

In this example, for the function call `reply(0)`, there is no return value. Therefore the **Return value not initialized** check returns a red error. The second call `reply(ch)` always returns a value. Therefore, the check on this call is green.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep();

int reply(int msg) {
  int rep = inputRep();
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch <= 0)
    ans = reply(0);
  else
    ans = reply(ch);
  printf("The answer is %d.",ans);
}
```

Function does not return value for some inputs

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
}

void main(void) {
  int ch = input(), ans;
  if (ch < 10)
    ans = reply(ch);
  else
    ans = reply(10);
```

```
  printf("The answer is %d.",ans);
}
```

In this example, in the first branch of the `if` statement, the value of `ch` can be divided into two ranges:

- `ch < = 0`: For the function call `reply(ch)`, there is no return value.
- `ch > 0` and `ch < 10`: For the function call `reply(ch)`, there is a return value.

Therefore the **Return value not initialized** check returns an orange error on `reply(ch)`.

Correction — Return value for all inputs

One possible correction is to return a value for all inputs to `reply()`.

```
#include <stdio.h>
int input();
int inputRep(int);

int reply(int msg) {
  int rep = inputRep(msg);
  if (msg > 0) return rep;
  return 0;
}

void main(void) {
  int ch = input(), ans;
  if (ch < 10)
    ans = reply(ch);
  else
    ans = reply(10);
  printf("The answer is %d.",ans);
}
```

Check Information

Group: Data flow
Language: C
Acronym: `IRV`

See Also

Topics

“Interpret Polyspace Code Prover Access Results”
“Code Prover Analysis Following Red and Orange Checks”

Subnormal float

Floating-point operation has subnormal results

Description

This check determines if a floating-point operation produces a subnormal result.

Subnormal numbers have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the significand. The presence of subnormal numbers indicates loss of significant digits. This loss can accumulate over subsequent operations and eventually result in unexpected values. Subnormal numbers can also slow down the execution on targets without hardware support.

By default, the results of the check do not appear in your verification results. To see the results of the check, change the default value of the option `Subnormal detection mode (-check-subnormal)`. The results of the check vary based on the detection mode that you specify. In all modes other than `allow`, to identify the subnormal results, look for red or orange **Subnormal float** checks on operations. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Mode	Check Colors	Behavior Following Check
`forbid`: This mode detects the occurrence of a subnormal value. This mode stops the execution path with the subnormal result and prevents subnormal values from propagating further. Therefore, in practice, you see only the first occurrence of the subnormal value.	The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands. For instance, if `x` is unknown, `x * 2` can be subnormal because `x` can be subnormal. The result of the check is orange.	Blocking check. If the check is red, the verification stops. If the check is orange, the verification removes the execution paths containing the subnormal result from consideration. For instance, the tooltip on the result does not show the subnormal values.
`warn-all`: This mode highlights all occurrences of subnormal values. Even if a subnormal result comes from previous subnormal values, the result is highlighted.	The color of the check depends only on the result of the operation. The check flags an operation that has subnormal results even if those results come only from subnormal operands. For instance, if `x` is unknown, `x * 2` can be subnormal because `x` can be subnormal. The result of the check is orange.	Non-blocking check. The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.

<table>
<tr><th>Mode</th><th>Check Colors</th><th>Behavior Following Check</th></tr>
<tr><td><code>warn-first</code>:

This mode highlights the first occurrence of a subnormal value. If a subnormal value propagates to further subnormal results, those subsequent results are not highlighted.</td><td>The check color depends on the result of the operation and the operand values. The check does not flag a subnormal result if it comes only from subnormal operands.

In this mode, the check is:

• Red, if the operation produces subnormal results on all execution paths that the software considers, and the operands are not subnormal.
• Orange, if the operation produces subnormal results on some of the execution paths when the operands are not subnormal.

For instance, if <code>x</code> is unknown, <code>x * 0.5</code> can be subnormal even if <code>x</code> is not subnormal.
• Green, if the operation does not produce subnormal results unless the operands are subnormal.

For instance, even if <code>x</code> is unknown, <code>x * 2</code> cannot be subnormal unless <code>x</code> is subnormal.</td><td>Non-blocking check.

The verification continues even if the check is red. If the check is orange, the verification does not remove the execution paths containing the subnormal result from consideration.</td></tr>
</table>

If you choose to check for subnormals, you can also identify from the tooltips whether a variable range excludes subnormal values. For instance, if the tooltips show `[-1.0 ..`

-1.1754E-38] or [-0.0..0.0] or [1.1754E-38..1.0], you can interpret that the variable does not have subnormal values.

Examples

Subnormal Results Detected with Red Checks

In the following examples, DBL_MIN is the minimum normal value that can be represented using the type double.

Results in forbid mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, the first **Subnormal float** check is red because the result of DBL_MIN/4.0 is subnormal. The red check stops the verification. The following operation, val * 2.0, is not verified for run-time errors.

Results in warn-all mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
    double val2 = val * 2.0;
}
```

In this example, both **Subnormal float** checks are red because both operations have subnormal results.

Results in warn-first mode:

```
#include <float.h>

void func(){
    double val = DBL_MIN/4.0;
```

```
    double val2 = val * 2.0;
}
```

In this example, `DBL_MIN` is not subnormal but the result of `DBL_MIN/4.0` is subnormal. The first **Subnormal float** check is red. The second **Subnormal float** check is green. The reason is that `val * 2.0` is subnormal only because `val` is subnormal. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Subnormal Results Detected with Orange Checks

In the following examples, `arg1` and `arg2` are unknown. The verification assumes that they can take all values allowed for the type `double`.

Results in `forbid` mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
      double val2 = difference2 * 2;
}
```

In this example, `difference1` can be subnormal if `arg1` and `arg2` are sufficiently close. The first **Subnormal float** check is orange. Following this check, the verification excludes from consideration the following:

- The close values of `arg1` and `arg2` that led to the subnormal value of `difference1`.

  In the subsequent operation `arg1 - arg2`, the **Subnormal float** check is green and `difference2` is not subnormal. The result of the check on `difference2 * 2` is green for the same reason.
- The subnormal value of `difference1`.

  In the subsequent operation `difference1 * 2`, the **Subnormal float** check is green.

Results in `warn-all` mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
```

```
      double val2 = difference2 * 2;
}
```

In this example, the four operations can have subnormal results. The four **Subnormal float** checks are orange.

Results in `warn-first` mode:

```
void func (double arg1, double arg2) {
      double difference1 = arg1 - arg2;
      double difference2 = arg1 - arg2;
      double val1 = difference1 * 2;
      double val2 = difference2 * 2;
}
```

In this example, if `arg1` and `arg2` are sufficiently close, `difference1` and `difference2` can be subnormal. The first two **Subnormal float** checks are orange. `val1` and `val2` cannot be subnormal unless `difference1` and `difference2` are also subnormal. The last two **Subnormal float** checks are green. Through red/orange checks, you see only the first instance where a subnormal value appears. You do not see red/orange checks from those subnormal values propagating to subsequent operations.

Conversion of Floating Point Literals

```
void main() {
    float d = 1e-38;
    float e = 1e-38 - 1e-39;
}
```

In this example, the two red checks appear in both `warn-first` and `warn-all` mode (the `forbid` mode prevents analysis after the first red check).

Literal constants such as `1e-38` have the data type `double`. If you assign a literal constant to a variable with narrower type `float`, the constant might not be representable in this type. This issue is indicated with the red checks. The checks flag the conversion from `double` to `float` during assignment.

Result Information

Group: Numerical

Language: C | C++
Acronym: SUBNORMAL

See Also

`Invalid operation on floats` | `Overflow`

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"
"Order of Code Prover Run-Time Checks"

Introduced in R2016b

Uncaught exception

Exception propagates uncaught to the `main` or another entry-point function

Description

This check looks for the following issues:

- An uncaught exception propagates to the `main` or another entry-point function.
- An exception is thrown in the constructor of a global variable and not caught.
- An exception is thrown in a destructor call or `delete` expression.
- An exception is thrown before a previous throw expression is handled by a `catch` statement, for instance, when constructing a `catch` statement parameters.
- A `noexcept` specification is violated. For instance, a function declared with `noexcept(true)` is not supposed to throw any exceptions but an exception is thrown in the function body.
- The data type of the exception that is actually thrown is not in the list of exception types that a function is declared to throw.

In these situations, according to the C++ standard, the `std::terminate` function is called and can cause unexpected results.

Note The **Uncaught exception** check on functions from the Standard Template Library such as `operator new` is green, even though Polyspace stubs these functions and does not check if a function throws an exception. To prevent the stubbing, use the option `No STL stubs (-no-stl-stubs)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Diagnosing This Check

"Review and Fix Uncaught Exception Checks"

Examples

Exception in call to function

```
#include <vector>
using namespace std;

class error {};

class initialVector {
private:
  int sizeVector;
  vector<int> table;
public:
  initialVector(int size) {
    sizeVector = size;
    table.resize(sizeVector);
    Initialize();
  }
  void Initialize();
  int getValue(int number) throw(error);
};

void initialVector::Initialize() {
  for(int i=0; i<table.size(); i++)
    table[i]=0;
}

int initialVector::getValue(int index) throw(error) {
  if(index >= 0 && index < sizeVector)
    return table[index];
  else throw error();
}

void main() {
    initialVector *vectorPtr = new initialVector(5);
    vectorPtr->getValue(5);
}
```

In this example, the call to method `initialVector::getValue` throws an exception. This exception propagates uncaught to the `main` function resulting in a red **Uncaught exception** check.

Exception handled through `try`/`catch` construct

```
class error {
public:
  error() {  }
  error(const error&) { }
};


void funcNegative() {
  try {
    throw error() ;
  } catch (error NegativeError) {
  }
}

void funcPositive() {
  try {
  }
  catch (error PositiveError) {
  /* Gray code */
  }
}

int input();
void main()
{
    int val=input();
    if(val < 0)
        /* Green check */
        funcNegative();
    else
        /* Green check */
        funcPositive();
}
```

In this example:

- The call to `funcNegative` throws an exception. However, the exception is placed inside a `try` block and is caught by the corresponding handler (`catch` clause). The **Uncaught exception** check on the `main` function appears green because the exception does not propagate to the `main`.
- The call to `funcPositive` does not throw an exception in the `try` block. Therefore, the `catch` block following the `try` block appears gray.

Exception in call to destructor

```
class error {
};

class X
{
public:
  X() {
    ;
  }
  ~X() {
    throw error();
  }
};

int main() {
  try {
    X * px = new X ;
    delete px;
  } catch (error) {
    assert(1) ;
  }
}
```

In this example, the `delete` operator calls the destructor `X::~X()`. The destructor throws an exception that appears as a red error on the destructor body and dashed red on the `delete` operator. The exception does not propagate to the `catch` block. The code following the exception is not verified. This behavior enforces the requirement that a destructor must not throw an exception.

The black `assert` statement suggests that the exception has not propagated to the `catch` block.

Exception in infinite loop

```
#include<stdio.h>
#define SIZE 100

int arr[SIZE];
int getIndex();

int runningSum() {
```

```
  int index, sum=0;
  while(1) {
    index=getIndex();
    if(index < 0 || index >= SIZE)
      throw int(1);
    sum+=arr[index];
  }
}

void main() {
    printf("The sum of elements is: %d",runningSum());
}
```

In this example, the `runningSum` function throws an exception only if `index` is outside the range `[0,SIZE]`. Typically, an error that occurs due to instructions in an `if` statement is orange, not red. The error is orange because an alternate execution path that does not involve the `if` statement does not produce an error. Here, because the loop is infinite, there is no alternate execution path that goes outside the loop. The only way to go outside the loop is through the exception in the `if` statement. Therefore, the **Uncaught exception** error is red.

Type mismatch between `throw` declaration and usage

```
#include <string>
using namespace std;

class negativeBalance {
public:
  negativeBalance(const string & s): errorMessage(s) {}
  ~negativeBalance() {}
private:
    string errorMessage;
};

class Account {
public:
  Account(long initVal):balance(initVal) {}
  ~Account() {}
  void debitAccount(long debitAmount) throw (int, char);
private:
  long balance;
};
```

```
void Account::debitAccount(long debitAmount) throw (int, char) {
    if((balance - debitAmount) < 0 )
        throw negativeBalance("Negative balance");
    else
        balance -= debitAmount;
}

void main() {
    Account *myAccount = new Account(1000);
    try {
        myAccount->debitAccount(2000);
    }
    catch(negativeBalance &) {
    }
    delete myAccount;
}
```

In this example, the types associated with the `throw` statement in the `Account::debitAccount` method are `int` and `char`. However, the method throws an exception with type `negativeBalance`. Therefore, the **Uncaught exception** check produces a red error on `throw`.

Rethrow outside `catch` block

```
#include <string>

void f() { throw; }        //rethrow not allowed - an error is raised here
void main() {
    try {
        throw std::string("hello");
     }
    catch (std::string& exc) {
        f();
    }
}
```

In this example, an exception is rethrown in the function `f()` outside a `catch` block. A rethrow occurs when you call `throw` by itself without an exception argument. A rethrow is typically used *inside* a `catch` block to propagate an exception to an outer `try-catch` sequence. Polyspace Code Prover does not support a rethrow *outside* a `catch` block and produces a red **Uncaught exception** error.

Check Information

Group: C++
Language: C++
Acronym: EXC

See Also

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

Unreachable code

Code cannot be reached during execution

Description

Unreachable code uses statement coverage to determine whether a section of code can be reached during execution. Statement coverage checks whether a program statement is executed. If a statement has test conditions, and at least one of them occurs, the statement is executed and reachable. The test conditions that do not occur are not considered dead code unless they have a corresponding code branch. If all the test conditions do not occur, the statement is not executed and each test condition is an instance of unreachable code. For example, in the `switch` statements of this code, `case 3` never occurs:

```
void test1 (int a) {
    int tmp = 0;
    if ((a!=3)) {
        switch (a){
            case 1:
                tmp++;
                break;
            default:
                tmp = 1;
                break;
/* case 3 falls through to
   case 2, no dead code */
            case 3:
            case 2:
                tmp = 100;
                break;
        }
    }
}

void test2 (int a) {
    int tmp = 0;
    if ((a!=3)) {
        switch (a){
            case 1:
                tmp++;
                break;
            default:
                tmp = 1;
                break;
// Dead code on case 3
            case 3:
                break;
            case 2:
                tmp = 100;
                break;
        }
    }
}
```

In `test1()`, `case 3` falls through to `case 2` and the check shows no dead code. In `test2()`, the check shows dead code for `case 3` because the `break` statement on the next line is not executed.

Other examples of unreachable code include:

- If a test condition always evaluates to false, the corresponding code branch cannot be reached. On the **Source** pane, the opening brace of the branch is gray.
- If a test condition always evaluates to true, the condition is redundant. On the **Source** pane, the condition keyword, such as `if`, appears gray.
- The code follows a `break` or `return` statement.

If an opening brace of a code block appears gray on the **Source** pane, to highlight the entire block, double-click the brace.

The check operates on code inside a function. The checks **Function not called** and **Function not reachable** determine if the function itself is not called or called from unreachable code.

Diagnosing This Check

"Review and Fix Unreachable Code Checks"

Examples

Test in `if` Statement Always False

```
#define True 1
#define False 0

typedef enum {
  Intermediate, End, Wait, Init
} enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval)  {
  if (stateval == Init)
    return True;
  return False;
}
```

```
int checkWait (enumState stateval)  {
  if (stateval == Wait)
    return True;
  return False;
}

void main()  {
  enumState myState = input(),refState = inputRef() ;
  if(checkInit(myState)){
    if(checkWait(myState)) {
      operation(myState,checkInit(refState));
    } else {
      operation(myState,checkWait(refState));
    }
  }
}
```

In this example, the `main` enters the branch of `if(checkInit(myState))` only if `myState = Init`. Therefore, inside that branch, Polyspace considers that `myState` has value `Init`. `checkWait(myState)` always returns `False` and the first branch of `if(checkWait(myState))` is unreachable.

Correction — Remove Redundant Test

One possible correction is to remove the redundant test `if(checkWait(myState))`.

```
#define True 1
#define False 0

typedef enum {
  Intermediate, End, Wait, Init
} enumState;

enumState input();
enumState inputRef();
void operation(enumState, int);

int checkInit (enumState stateval)  {
  if (stateval == Init)
    return True;
  return False;
}
```

```
int checkWait (enumState stateval)  {
  if (stateval == Wait) return True;
  return False;
}

void main()  {
  enumState myState = input(),refState = inputRef() ;
  if(checkInit(myState))
    operation(myState,checkWait(refState));
}
```

Test in `if` Statement Always True

```
#include <stdlib.h>
#include <time.h>

int roll() {
  return(rand()%6+1);
}

void operation(int);

void main()   {
    srand(time(NULL));
    int die = roll();
    if(die >= 1 && die <= 6)
      /*Unreachable code*/
      operation(die);
  }
```

In this example, `roll()` returns a value between 1 and 6. Therefore the `if` test in `main` always evaluates to true and is redundant. If there is a corresponding `else` branch, the gray error appears on the `else` statement. Without an `else` branch, the gray error appears on the `if` keyword to indicate the redundant condition.

Correction — Remove Redundant Test

One possible correction is to remove the condition `if(die >= 1 && die <=6)`.

```
#include <stdlib.h>
#include <time.h>

int roll() {
  return(rand()%6+1);
```

```
}

void operation(int);

void main()   {
  srand(time(NULL));
  int die = roll();
  operation(die);
}
```

Test in `if` Statement Unreachable

```
#include <stdlib.h>
#include <time.h>
#define True 1
#define False 0

int roll1() {
  return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main()   {
  srand(time(NULL));
  int die1 = roll1(),die2=roll2();
  if((die1>=1 && die1<=6) ||
     (die2>=1 && die2 <=6))
  /*Unreachable code*/
    operation(die1,die2);
}
```

In this example, `roll1()` returns a value between 1 and 6. Therefore, the first part of the `if` test, `if((die1>=1) && (die1<=6))` is always true. Because the two parts of the `if` test are combined with `||`, the `if` test is always true irrespective of the second part. Therefore, the second part of the `if` test is unreachable.

Correction — Combine Tests with &&

One possible correction is to combine the two parts of the `if` test with && instead of `||`.

```
#include <stdlib.h>
#include <time.h>
```

```
#define True 1
#define False 0

int roll1() {
  return(rand()%6+1);
}

int roll2();
void operation(int,int);

void main()   {
  srand(time(NULL));
  int die1 = roll1(),die2=roll2();
  if((die1>=1 && die1<=6) &&
     (die2>=1 && die2<=6))
    operation(die1,die2);
}
```

Check Information

Group: Data flow
Language: C | C++
Acronym: `UNR`

See Also

`Function not called` | `Function not reachable`

Topics

"Interpret Polyspace Code Prover Access Results"

User assertion

`assert` statement fails

Description

This check determines whether the argument to an `assert` macro is true.

The argument to the `assert` macro must be true when the macro executes. Otherwise the program aborts and prints an error message. Polyspace models this behavior by treating a failed `assert` statement as a run-time error. This check allows you to detect failed `assert` statements before program execution.

Diagnosing This Check

"Review and Fix User Assertion Checks"

Examples

Red `assert` on array index

```
#include<stdio.h>
#define size 20

int getArrayElement();

void initialize(int* array) {
 for(int i=0;i<size;i++)
   array[i] = getArrayElement();
}

void printElement(int* array,int index) {
 assert(index < size);
 printf("%d", array[index]);
}
```

```
int getIndex() {
 int i = size;
 return i;
}

void main() {
 int array[size];
 int index;

 initialize(array);
 index = getIndex();
 printElement(array,index);

}
```

In this example, the `assert` statement in `printElement` causes program abort if `index >= size`. The `assert` statement makes sure that the array index is not outside array bounds. If the code does not contain exceptional situations, the `assert` statement must be green. In this example, `getIndex` returns an index equal to `size`. Therefore the `assert` statement appears red.

Correction — Correct cause of `assert` failure

When an `assert` statement is red, investigate the cause of the exceptional situation. In this example, one possible correction is to force `getIndex` to return an index equal to `size-1`.

```
#include<stdio.h>
#define size 20

int getArrayElement();

void initialize(int* array) {
 for(int i=0;i<size;i++)
  array[i] = getArrayElement();
}

void printElement(int* array,int index) {
 assert(index < size);
 printf("%d", array[index]);
}

int getIndex() {
 int i = size;
```

```
 return (i-1);
}

void main() {
 int array[size];
 int index;

 initialize(array);
 index = getIndex();
 printElement(array,index);

}
```

Orange `assert` on `malloc` return value

```
#include <stdlib.h>

void initialize(int*);
int getNumberOfElements();

void main() {
 int numberOfElements, *myArray;

 numberOfElements = getNumberOfElements();

 myArray = (int*)malloc(numberOfElements);
 assert(myArray!=NULL);

 initialize(myArray);
}
```

In this example, `malloc` can return `NULL` to `myArray`. Therefore, `myArray` can have two possible values:

- `myArray == NULL`: The `assert` condition is false.
- `myArray != NULL`: The `assert` condition is true.

Combining these two cases, the **User assertion** check on the `assert` statement is orange. After the orange `assert`, Polyspace considers that `myArray` is not equal to `NULL`.

Correction — Check return value for `NULL`

One possible correction is to write a customized function `myMalloc` where you always check the return value of `malloc` for `NULL`.

```
#include <stdio.h>
#include <stdlib.h>

void initialize(int*);
int getNumberOfElements();

void myMalloc(int **ptr, int num) {
 *ptr = (int*)malloc(num);
 if(*ptr==NULL) {
    printf("Memory allocation error");
    exit(1);
  }
}

void main() {
 int numberOfElements, *myArray=NULL;

 numberOfElements = getNumberOfElements();

 myMalloc(&myArray,numberOfElements);
 assert(myArray!=NULL);

 initialize(myArray);
}
```

Imposing constraint through orange assert

```
#include<stdio.h>
#include<math.h>

float getNumber();
void squareRootOfDifference(float firstNumber, float secondNumber) {
   assert(firstNumber > secondNumber);
   if(firstNumber > 0 && secondNumber > 0)
   printf("Square root = %.2f",sqrt(firstNumber-secondNumber));
}

void main() {
   double firstNumber = getNumber(), secondNumber = getNumber();
   squareRootOfDifference(firstNumber,secondNumber);
}
```

In this example, the `assert` statement in `squareRootOfDifference()` causes program abort if `firstNumber` is less than `secondNumber`. Because Polyspace does not have enough information about `firstNumber` and `secondNumber`, the `assert` is orange.

Following the `assert`, all execution paths that cause assertion failure terminate. Therefore, following the `assert`, Polyspace considers that `firstNumber >= secondNumber`. The **Invalid use of standard library routine** check on `sqrt` is green.

Use `assert` statements to help Polyspace determine:

- Relationships between variables
- Constraints on variable ranges

Check Information

Group: Other
Language: C | C++
Acronym: ASRT

See Also

Topics

"Interpret Polyspace Code Prover Access Results"
"Code Prover Analysis Following Red and Orange Checks"

MISRA C 2012

MISRA C:2012 Rule 1.1

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits

Description

Rule Definition

The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

Polyspace Implementation

The rule violation can come from multiple causes. Standard compilation error messages do not lead to a violation of this MISRA® rule.

Tip To mass-justify all results that come from the same cause, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Message in Report

- Too many nesting levels of #includes: N1. The limit is N0.

  Note: The rule checker considers a brace as an additional level. For instance, the `if` branch in this code is counted as two levels of nesting.

  ```
  if(flag) {
  }
  ```

  The metric `Number of Call Levels` counts this as one level of nesting.
- Integer constant is too large.

- ANSI C does not allow '#XX'.
- Text following preprocessing directive violates ANSI standard.
- Too many macro definitions: N1. The limit is N0.
- Array of zero size should not be used.
- Integer constant does not fit within long int.
- Integer constant does not fit within unsigned long int.
- Too many nesting levels for control flow: N1. The limit is N0.
- Assembly language should not be used.
- Too many enumeration constants: N1. The limit is N0.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 1.2`

Introduced in R2014b

MISRA C:2012 Rule 1.2

Language extensions should not be used

Description

Rule Definition

Language extensions should not be used.

Rationale

If a program uses language extensions, its portability is reduced. Even if you document the language extensions, the documentation might not describe the behavior in all circumstances.

Polyspace Implementation

All the supported extensions lead to a violation of this MISRA rule.

Message in Report

- ANSI C90 forbids hexadecimal floating-point constants.
- ANSI C90 forbids universal character names.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids case ranges.
- ANSI C90/C99 forbids local label declaration.
- ANSI C90 forbids mixed declarations and code.
- ANSI C90/C99 forbids typeof operator.
- ANSI C90/C99 forbids casts to union.
- ANSI C90 forbids compound literals.
- ANSI C90/C99 forbids statements and declarations in expressions.

- ANSI C90 forbids __func__ predefined identifier.
- ANSI C90 forbids keyword '_Bool'.
- ANSI C90 forbids 'long long int' type.
- ANSI C90 forbids long long integer constants.
- ANSI C90 forbids 'long double' type.
- ANSI C90/C99 forbids 'short long int' type.
- ANSI C90 forbids _Pragma preprocessing operator.
- ANSI C90 does not allow macros with variable arguments list.
- ANSI C90 forbids designated initializer.

  Keyword 'inline' should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard C Environment
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 1.1`

Introduced in R2014b

MISRA C:2012 Rule 1.3

There shall be no occurrence of undefined or critical unspecified behaviour

Description

Rule Definition

There shall be no occurrence of undefined or critical unspecified behaviour.

Message in Report

There shall be no occurrence of undefined or critical unspecified behavior

- 'defined' without an identifier.
- macro 'XX' used with too few arguments.
- macro 'XX used with too many arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard C Environment
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.1`

Introduced in R2014b

MISRA C:2012 Rule 10.1

Operands shall not be of an inappropriate essential type

Description

Rule Definition

Operands shall not be of an inappropriate essential type.

Rationale

What Are Essential Types?

An essential type category defines the essential type of an object or expression.

Essential type category	Standard types
Essentially Boolean	`bool` or `_Bool` (defined in `stdbool.h`) You can also define types that are essentially Boolean using the option `-boolean-types`.
Essentially character	`char`
Essentially enum	named `enum`
Essentially signed	signed `char`, signed `short`, signed `int`, signed `long`, signed `long long`
Essentially unsigned	unsigned `char`, unsigned `short`, unsigned `int`, unsigned `long`, unsigned `long long`
Essentially floating	`float`, `double`, `long double`

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Amplification and Rationale

For operands of some operators, you cannot use certain essential types. In the table below, each row represents an operator/operand combination. If the essential type column is not empty for that row, there is a MISRA restriction when using that type as the operand. The number in the table corresponds to the rationale list after the table.

Operation		**Essential type category of arithmetic operand**					
Operator	**Operand**	**Boolean**	**character**	**enum**	**signed**	**unsigned**	**floating**
[ ]	integer	3	4				1
+ (unary)		3	4	5			
- (unary)		3	4	5		8	
+ -	either	3		5			
* /	either	3	4	5			
%	either	3	4	5			1
< > <= >=	either	3					
== !=	either						
! && ||	any		2	2	2	2	2
<< >>	left	3	4	5,6	6		1
<< >>	right	3	4	7	7		1
~ & | ^	any	3	4	5,6	6		1
?:	1st		2	2	2	2	2
?:	2nd and 3rd						

1 An expression of essentially floating type for these operands is a constraint violation.

2 When an operand is interpreted as a Boolean value, use an expression of essentially Boolean type.

3 When an operand is interpreted as a numeric value, do not use an operand of essentially Boolean type.

4 When an operand is interpreted as a numeric value, do not use an operand of essentially character type. The numeric values of character data are implementation-defined.

5 In an arithmetic operation, do not use an operand of essentially enum type. An enum object uses an implementation-defined integer type. An operation involving an enum object can therefore yield a result with an unexpected type.

6 Perform only shift and bitwise operations on operands of essentially unsigned type. When you use shift and bitwise operations on essentially signed types, the resulting numeric value is implementation-defined.

7 To avoid undefined behavior on negative shifts, use an essentially unsigned right-hand operand.

8 For the unary minus operator, do not use an operand of essentially unsigned type. The implemented size of int determines the signedness of the result.

Message in Report

The *operand_name* operand of the *operator_name* operator is of an inappropriate essential type category *category_name*.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Violation of Rule 10.1, Rationale 2: Inappropriate Operand Types for Operators That Take Essentially Boolean Operands

```
typedef unsigned char boolean;

extern float f32a;
extern char cha;
extern signed char s8a;
extern unsigned char u8a;
enum enuma { a1, a2, a3 } ena;

extern boolean bla, blb, rbla;

void foo(void) {

  rbla = cha && bla;        /* Non-compliant: cha is essentially char  */
  enb = ena ? a1 : a2;      /* Non-compliant: ena is essentially enum  */
  rbla = s8a && bla;        /* Non-compliant: s8a is essentially signed char  */
```

```
  ena = u8a ? a1 : a2;      /* Non-compliant: u8a is essentially unsigned char  */
  rbla = f32a && bla;       /* Non-compliant: f32a is essentially float */

  rbla = bla && blb;        /* Compliant */
  ru8a = bla ? u8a : u8b;   /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because:

- The operator && expects only essentially Boolean operands. However, at least one of the operands used has a different type.
- The first operand of `?:` is expected to be essentially Boolean. However, a different operand type is used.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Violation of Rule 10.1, Rationale 3: Inappropriate Boolean Operands

```
typedef unsigned char boolean;


enum enuma { a1, a2, a3 } ena;
enum { K1 = 1, K2 = 2 };    /* Essentially signed */
extern char cha, chb;
extern boolean bla, blb, rbla;
extern signed char rs8a, s8a;


void foo(void) {

  rbla = bla * blb;      /* Non-compliant - Boolean used as a numeric value */
  rbla = bla > blb;      /* Non-compliant - Boolean used as a numeric value */

  rbla = bla && blb;     /* Compliant */
  rbla = cha > chb;      /* Compliant */
  rbla = ena > a1;       /* Compliant */
```

```
  rbla = u8a > 0U;       /* Compliant */
  rs8a = K1 * s8a;       /* Compliant - K1 obtained from anonymous enum */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators * and > do not expect essentially Boolean operands. However, the operands used here are essentially Boolean.

Note For Polyspace to detect the rule violation, you must define the type name `boolean` as an effective Boolean type. For more information, see `Effective boolean types (-boolean-types)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Violation of Rule 10.1, Rationale 4: Inappropriate Character Operands

```
extern char rcha, cha, chb;
extern unsigned char ru8a, u8a;

void foo(void) {

  rcha = cha & chb;      /* Non-compliant - char type used as a numeric value */
  rcha = cha << 1;       /* Non-compliant - char type used as a numeric value */

  ru8a = u8a & 2U;       /* Compliant */
  ru8a = u8a << 2U;      /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the operators & and << do not expect essentially character operands. However, at least one of the operands used here has essentially character type.

Violation of Rule 10.1, Rationale 5: Inappropriate Enum Operands

```
typedef unsigned char boolean;

enum enuma { a1, a2, a3 } rena, ena, enb;
```

```
void foo(void) {

  ena--;             /* Non-Compliant - arithmetic operation with enum type*/
  rena = ena * a1;   /* Non-Compliant - arithmetic operation with enum type*/
  ena += a1;         /* Non-Compliant - arithmetic operation with enum type*/

}
```

In the noncompliant examples, rule 10.1 is violated because the arithmetic operators `--`, `*` and `+=` do not expect essentially enum operands. However, at least one of the operands used here has essentially enum type.

Violation of Rule 10.1, Rationale 6: Inappropriate Signed Operand for Bitwise Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = s8a & 2;       /* Non-compliant - bitwise operation on signed type */
  ru8a = 2 << 3U;       /* Non-compliant - shift operation on signed type */

  ru8a = u8a << 2U;     /* Compliant */

}
```

In the noncompliant examples, rule 10.1 is violated because the `&` and `<<` operations must not be performed on essentially signed operands. However, the operands used here are signed.

Violation of Rule 10.1, Rationale 7: Inappropriate Signed Right Operand for Shift Operations

```
extern signed char s8a;
extern unsigned char ru8a, u8a;

void foo(void) {

  ru8a = u8a << s8a;    /* Non-compliant - shift magnitude uses signed type */
  ru8a = u8a << -1;     /* Non-compliant - shift magnitude uses signed type */
```

```
  ru8a = u8a << 2U;     /* Compliant */
  ru8a = u8a << 1;      /* Compliant - exception */

}
```

In the noncompliant examples, rule 10.1 is violated because the operation `<<` does not expect an essentially signed right operand. However, the right operands used here are signed.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.2`

MISRA C:2012 Rule 10.2

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations

Description

Rule Definition

Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

Rationale

Essentially character type expressions are `char` variables. Do not use character data arithmetically because the data does not represent numeric values.

Message in Report

- The *`operand_name`* operand of the `+` operator applied to an expression of essentially character type shall have essentially signed or unsigned type.
- The right operand of the `-` operator applied to an expression of essentially character type shall have essentially signed or unsigned or character type.
- The left operand of the `-` operator shall have essentially character type if the right operand has essentially character type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model

Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.1`

MISRA C:2012 Rule 10.3

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category

Description

Rule Definition

The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

Message in Report

- The expression is assigned to an object with a different essential type category.
- The expression is assigned to an object with a narrower essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.4` | `MISRA C:2012 Rule 10.5` | `MISRA C:2012 Rule 10.6`

Topics

"Justify Coding Rule Violations Using Code Prover Checks"

MISRA C:2012 Rule 10.4

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category

Description

Rule Definition

Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

Rationale

The use of implicit conversions between types can lead to unintended results, including possible loss of value, sign, or precision.

Polyspace Implementation

Polyspace does not produce a violation of this rule:

- If one of the operands is the constant zero.
- If one of the operands is a signed constant and the other operand is unsigned, and the signed constant has the same representation as its unsigned equivalent.

  For instance, the statement `u8b = u8a + 3;`, where `u8a` and `u8b` are `unsigned char` variables, does not violate the rule because the constants `3` and `3U` have the same representation.

Message in Report

Operands of *`operator_name`* operator shall have the same essential type category.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7`

MISRA C:2012 Rule 10.5

The value of an expression should not be cast to an inappropriate essential type

Description

Rule Definition

The value of an expression should not be cast to an inappropriate essential type.

Rationale

Converting Between Variable Types

		From					
		Boolean	**character**	**enum**	**signed**	**unsigned**	**floating**
To	**Boolean**		Avoid	Avoid	Avoid	Avoid	Avoid
	character	Avoid					Avoid
	enum	Avoid	Avoid	Avoid	Avoid	Avoid	Avoid
	signed	Avoid					
	unsigned	Avoid					
	floating	Avoid	Avoid				

Some inappropriate explicit casts are:

- In C99, the result of a cast of assignment to `_Bool` is always 0 or 1. This result is not necessarily the case when casting to another type which is defined as essentially Boolean.
- A cast to an essential enum type may result in a value that does not lie within the set of enumeration constants for that type.
- A cast from essential Boolean to any other type is unlikely to be meaningful.
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Some acceptable explicit casts are:

- To change the type in which a subsequent arithmetic operation is performed.
- To truncate a value deliberately.
- To make a type conversion explicit in the interests of clarity.

Message in Report

The value of an expression should not be cast to an inappropriate essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.8`

MISRA C:2012 Rule 10.6

The value of a composite expression shall not be assigned to an object with wider essential type

Description

Rule Definition

The value of a composite expression shall not be assigned to an object with wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

If you assign the result of a composite expression to a larger type, the implicit conversion can result in loss of value, sign, precision, or layout.

Message in Report

The composite expression is assigned to an object with a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.3` | `MISRA C:2012 Rule 10.7`

MISRA C:2012 Rule 10.7

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type

Description

Rule Definition

If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed, then the other operand shall not have wider essential type.

Rationale

A *composite expression* is a nonconstant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

Restricting implicit conversion on composite expressions mean that sequences of arithmetic operations within expressions must use the same essential type. This restriction reduces confusion and avoids loss of value, sign, precision, or layout. However, this rule does not imply that all operands in an expression are of the same essential type.

Message in Report

- The right operand shall not have wider essential type than the left operand which is a composite expression.

- The left operand shall not have wider essential type than the right operand which is a composite expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 10.8

The value of a composite expression shall not be cast to a different essential type category or a wider essential type

Description

Rule Definition

The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Rationale

A *composite expression* is a non-constant expression using a composite operator. In the Essential Type Model, composite operators are:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary `+`, binary `-`)
- Bitwise (&, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?`, `:`)

Casting to a wider type is not permitted because the result may vary between implementations. Consider this expression:

```
(uint32_t) (u16a +u16b);
```

On a 16-bit machine the addition is performed in 16 bits. The result is wrapped before it is cast to 32 bits. On a 32-bit machine, the addition takes place in 32 bits and preserves high-order bits that are lost on a 16-bit machine. Casting to a narrower type with the same essential type category is acceptable as the explicit truncation of the results always leads to the same loss of information.

For information on essential types, see `MISRA C:2012 Rule 10.1`.

Polyspace Implementation

The rule checker raises a defect only if the result of a composite expression is cast to a different or wider essential type.

For instance, in this example, a violation is shown in the first assignment to `i` but not the second. In the first assignment, a composite expression `i+1` is directly cast from a signed to an unsigned type. In the second assignment, the composite expression is first cast to the same type and then the result is cast to a different type.

```
typedef int int32_T;
typedef unsigned char uint8_T;
...
...
int32_T i;
i = (uint8_T)(i+1); /* Noncompliant */
i = (uint8_T)((int32_T)(i+1)); /* Compliant */
```

Message in Report

- The value of a composite expression shall not be cast to a different essential type category.
- The value of a composite expression shall not be cast to a wider essential type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casting to Different or Wider Essential Type

```
extern unsigned short ru16a, u16a, u16b;
extern unsigned int   u32a, ru32a;
extern signed int     s32a, s32b;

void foo(void)
{
```

```
  ru16a  = (unsigned short) (u32a + u32a);/* Compliant                       */
  ru16a += (unsigned short) s32a;    /* Compliant - s32a is not composite    */
  ru32a  = (unsigned int) (u16a + u16b);  /* Noncompliant - wider essential type */
}
```

In this example, rule 10.8 is violated in the following cases:

- `s32a` and `s32b` are essentially `signed` variables. However, the result `( s32a + s32b )` is cast to an essentially `unsigned` type.
- `u16a` and `u16b` are essentially `unsigned short` variables. However, the result `( s32a + s32b )` is cast to a wider essential type, `unsigned int`.

Check Information

Group: The Essential Type Model
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.5`

MISRA C:2012 Rule 11.1

Conversions shall not be performed between a pointer to a function and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to a function and any other type.

Rationale

The rule forbids the following two conversions:

- Conversion from a function pointer to any other type. This conversion causes undefined behavior.
- Conversion from a function pointer to another function pointer, if the function pointers have different argument and return types.

  The conversion is forbidden because calling a function through a pointer with incompatible type results in undefined behavior.

Polyspace Implementation

Polyspace considers both explicit and implicit casts when checking this rule. However, casts from `NULL` or `(void*)0` do not violate this rule.

Message in Report

Conversions shall not be performed between a pointer to a function and any other type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Cast between two function pointers

```
typedef void (*fp16) (short n);
typedef void (*fp32) (int n);

#include <stdlib.h>                     /* To obtain macro  NULL */

void func(void) {   /* Exception 1 - Can convert a null pointer
                     * constant into a pointer to a function */
  fp16 fp1 = NULL;                 /* Compliant - exception  */
  fp16 fp2 = (fp16) fp1;           /* Compliant */
  fp32 fp3 = (fp32) fp1;           /* Non-compliant */
  if (fp2 != NULL) {}              /* Compliant - exception  */
  fp16 fp4 = (fp16) 0x8000;        /* Non-compliant - integer to
                                    * function pointer */}
```

In this example, the rule is violated when:

- The pointer `fp1` of type `fp16` is cast to type `fp32`. The function pointer types `fp16` and `fp32` have different argument types.
- An integer is cast to type `fp16`.

The rule is not violated when function pointers `fp1` and `fp2` are cast to `NULL`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 11.2

Conversions shall not be performed between a pointer to an incomplete type and any other type

Description

Rule Definition

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Rationale

An incomplete type is a type that does not contain sufficient information to determine its size. For example, the statement `struct s;` describes an incomplete type because the fields of `s` are not defined. The size of a variable of type `s` cannot be determined.

Conversions to or from a pointer to an incomplete type result in undefined behavior. Typically, a pointer to an incomplete type is used to hide the full representation of an object. This encapsulation is broken if another pointer is implicitly or explicitly cast to such a pointer.

Message in Report

Conversions shall not be performed between a pointer to an incomplete type and any other type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casts from incomplete type

```
struct s *sp;
struct t *tp;
short  *ip;
struct ct *ctp1;
struct ct *ctp2;


void foo(void) {

    ip = (short *) sp;            /* Non-compliant */
    sp = (struct s *) 1234;       /* Non-compliant */
    tp = (struct t *) sp;         /* Non-compliant */
    ctp1 = (struct ct *) ctp2;    /* Compliant */

    /* You can convert a null pointer constant to
     * a pointer to an incomplete type */
    sp = NULL;                    /* Compliant - exception  */

    /* A pointer to an incomplete type may be converted into void */
    struct s *f(void);
    (void) f();                   /* Compliant - exception  */

}
```

In this example, types `s`, `t` and `ct` are incomplete. The rule is violated when:

- The variable `sp` with an incomplete type is cast to a basic type.
- The variable `sp` with an incomplete type is cast to a different incomplete type `t`.

The rule is not violated when:

- The variable `ctp2` with an incomplete type is cast to the same incomplete type.
- The `NULL` pointer is cast to the variable `sp` with an incomplete type.
- The return value of `f` with incomplete type is cast to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.5`

Introduced in R2014b

MISRA C:2012 Rule 11.3

A cast shall not be performed between a pointer to object type and a pointer to a different object type

Description

Rule Definition

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Rationale

If a pointer to an object is cast into a pointer to a different object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.

Even if the conversion produces a pointer that is correctly aligned, the behavior can be undefined if the pointer is used to access an object.

Exception: You can convert a pointer to object type into a pointer to one of the following types:

- `char`
- `signed char`
- `unsigned char`

Message in Report

A cast shall not be performed between a pointer to object type and a pointer to a different object type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Noncompliant: Cast to Pointer Pointing to Object of Wider Type

```
signed   char *p1;
unsigned int *p2;

void foo(void){
  p2 = ( unsigned int * ) p1;     /* Non-compliant */
}
```

In this example, `p1` can point to a `signed char` object. However, `p1` is cast to a pointer that points to an object of wider type, `unsigned int`.

Noncompliant: Cast to Pointer Pointing to Object of Narrower Type

```
extern unsigned int read_value ( void );
extern void display ( unsigned int n );

void foo ( void ){
  unsigned int u = read_value ( );
  unsigned short *hi_p = ( unsigned short * ) &u;    /* Non-compliant  */
  *hi_p = 0;
  display ( u );
}
```

In this example, `u` is an `unsigned int` variable. `&u` is cast to a pointer that points to an object of narrower type, `unsigned short`.

On a big-endian machine, the statement `*hi_p = 0` attempts to clear the high bits of the memory location that `&u` points to. But, from the result of `display(u)`, you might find that the high bits have not been cleared.

Compliant: Cast Adding a Type Qualifier

```
const short *p;
const volatile short *q;
void foo (void){
```

```
  q = ( const volatile short * ) p;  /* Compliant */
}
```

In this example, both `p` and `q` can point to `short` objects. The cast between them adds a `volatile` qualifier only and is therefore compliant.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.5` | `MISRA C:2012 Rule 11.8`

Introduced in R2014b

MISRA C:2012 Rule 11.4

A conversion should not be performed between a pointer to object and an integer type

Description

Rule Definition

A conversion should not be performed between a pointer to object and an integer type.

Rationale

Conversion between integers and pointers can cause errors or undefined behavior.

- If an integer is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an integer, the resulting value can be outside the allowed range for the integer type.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed between a pointer to object and an integer type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casts between pointer and integer

```
#include <stdbool.h>

typedef unsigned char      uint8_t;
typedef          char      char_t;
typedef unsigned short     uint16_t;
typedef signed   int       int32_t;

typedef _Bool bool_t;
uint8_t *PORTA = (uint8_t *) 0x0002;            /* Non-compliant */

void foo(void) {

    char_t c = 1;
    char_t *pc = &c;                              /* Compliant */


    uint16_t ui16   = 7U;
    uint16_t *pui16 = &ui16;                      /* Compliant */
    pui16 = (uint16_t *) ui16;                    /* Non-compliant */


    uint16_t *p;
    int32_t addr = (int32_t) p;                  /* Non-compliant */
    bool_t b = (bool_t) p;                       /* Non-compliant */
    enum etag { A, B } e = ( enum etag ) p;      /* Non-compliant */
}
```

In this example, the rule is violated when:

- The integer `0x0002` is cast to a pointer.

  If the integer defines an absolute address, it is more common to assign the address to a pointer in a header file. To avoid the assignment being flagged, you can then exclude headers files from coding rules checking. For more information, see . For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.
- The pointer `p` is cast to integer types such as `int32_t`, `bool_t` or `enum etag`.

The rule is not violated when the address `&ui16` is assigned to a pointer.

Check Information

Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.3` | `MISRA C:2012 Rule 11.7` | `MISRA C:2012 Rule 11.9`

Introduced in R2014b

MISRA C:2012 Rule 11.5

A conversion should not be performed from pointer to void into pointer to object

Description

Rule Definition

A conversion should not be performed from pointer to void into pointer to object.

Rationale

If a pointer to `void` is cast into a pointer to an object, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. However, such a cast can sometimes be necessary, for example, when using memory allocation functions.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A conversion should not be performed from pointer to void into pointer to object.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Cast from Pointer to `void`

```
void foo(void) {
```

```
    unsigned int  u32a = 0;
    unsigned int  *p32 = &u32a;
    void          *p;
    unsigned int  *p16;

    p   = p32;                /* Compliant - pointer to uint32_t
                               *             into pointer to void */
    p16 = p;                  /* Non-compliant */

    p   = (void *) p16;       /* Compliant */
    p32 = (unsigned int *) p; /* Non-compliant */
}
```

In this example, the rule is violated when the pointer `p` of type `void*` is cast to pointers to other types.

The rule is not violated when `p16` and `p32`, which are pointers to non-`void` types, are cast to `void*`.

Check Information

Group: Pointer Type Conversions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.2` | `MISRA C:2012 Rule 11.3`

Introduced in R2014b

MISRA C:2012 Rule 11.6

A cast shall not be performed between pointer to void and an arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to void and an arithmetic type.

Rationale

Conversion between integer types and pointers to `void` can cause errors or undefined behavior.

- If an integer type is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior.
- If a pointer is cast to an arithmetic type, the resulting value can be outside the allowed range for the type.

Conversion between non-integer arithmetic types and pointers to `void` is undefined.

Polyspace Implementation

Casts or implicit conversions from `NULL` or `(void*)0` do not generate a warning.

Message in Report

A cast shall not be performed between pointer to void and an arithmetic type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casts Between Pointer to void and Arithmetic Types

```
void foo(void) {

    void          *p;
    unsigned int  u;
    unsigned short r;

    p = (void *) 0x1234u;            /* Non-compliant - undefined */
    u = (unsigned int) p;            /* Non-compliant - undefined */

    p = (void *) 0;                  /* Compliant - Exception */

}
```

In this example, `p` is a pointer to `void`. The rule is violated when:

- An integer value is cast to `p`.
- `p` is cast to an `unsigned int` type.

The rule is not violated if an integer constant with value 0 is cast to a pointer to `void`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 11.7

A cast shall not be performed between pointer to object and a non-integer arithmetic type

Description

Rule Definition

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Rationale

This rule covers types that are essentially Boolean, character, enum or floating.

- If an essentially Boolean, character or enum variable is cast to a pointer, the resulting pointer can be incorrectly aligned. The incorrect alignment causes undefined behavior. If a pointer is cast to one of those types, the resulting value can be outside the allowed range for the type.
- Casts to or from a pointer to a floating type results in undefined behavior.

Message in Report

A cast shall not be performed between pointer to object and a non-integer arithmetic type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casts from Pointer to Non-Integer Arithmetic Types

```
int foo(void) {

    short *p;
    float  f;
    long  *l;

    f = (float)   p;              /* Non-compliant */
    p = (short *) f;              /* Non-compliant */

    l = (long *)  p;              /* Compliant */
}
```

In this example, the rule is violated when:

- The pointer `p` is cast to `float`.
- A `float` variable is cast to a pointer to `short`.

The rule is not violated when the pointer `p` is cast to `long*`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.4`

Introduced in R2014b

MISRA C:2012 Rule 11.8

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Rationale

This rule forbids:

- Casts from a pointer to a `const` object to a pointer that does not point to a `const` object.
- Casts from a pointer to a `volatile` object to a pointer that does not point to a `volatile` object.

Such casts violate type qualification. For example, the `const` qualifier indicates the read-only status of an object. If a cast removes the qualifier, the object is no longer read-only.

Polyspace Implementation

Polyspace flags both implicit and explicit conversions that violate this rule.

Message in Report

A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Casts That Remove Qualifiers

```
void foo(void) {

    /* Cast on simple type */
    unsigned short          x;
    unsigned short * const   cpi = &x;  /* const pointer */
    unsigned short * const  *pcpi;   /* pointer to const pointer */
    unsigned short **ppi;
    const unsigned short    *pci;    /* pointer to const */
    volatile unsigned short *pvi;    /* pointer to volatile  */
    unsigned short          *pi;

    pi = cpi;                        /* Compliant - no cast required */
    pi  = (unsigned short *)  pci;   /* Non-compliant */
    pi  = (unsigned short *)  pvi;   /* Non-compliant */
    ppi = (unsigned short **)pcpi;   /* Non-compliant */
}
```

In this example:

- The variables `pci` and `pcpi` have the `const` qualifier in their type. The rule is violated when the variables are cast to types that do not have the `const` qualifier.
- The variable `pvi` has a `volatile` qualifier in its type. The rule is violated when the variable is cast to a type that does not have the `volatile` qualifier.

Even though `cpi` has a `const` qualifier in its type, the rule is not violated in the statement `p=cpi;`. The assignment does not cause a type conversion because both `p` and `cpi` have type `unsigned short`.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.3`

Introduced in R2014b

MISRA C:2012 Rule 11.9

The macro NULL shall be the only permitted form of integer null pointer constant

Description

Rule Definition

The macro NULL shall be the only permitted form of integer null pointer constant.

Rationale

The following expressions require the use of a null pointer constant:

- Assignment to a pointer
- The `==` or `!=` operation, where one operand is a pointer
- The `?:` operation, where one of the operands on either side of `:` is a pointer

Using `NULL` rather than 0 makes it clear that a null pointer constant was intended.

Message in Report

The macro NULL shall be the only permitted form of integer null pointer constant.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Using 0 for Pointer Assignments and Comparisons

```
void main(void) {
```

```
    int *p1 = 0;               /* Non-compliant */
    int *p2 = ( void * ) 0;    /* Compliant     */

#define MY_NULL_1 0
#define MY_NULL_2 ( void * ) 0

    if ( p1 == MY_NULL_1 )     /* Non-compliant */
    { }
    if ( p2 == MY_NULL_2 )     /* Compliant     */
    { }

}
```

In this example, the rule is violated when the constant 0 is used instead of `(void*) 0` for pointer assignments and comparisons.

Check Information

Group: Pointer Type Conversions
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.4`

Introduced in R2014b

MISRA C:2012 Rule 12.1

The precedence of operators within expressions should be made explicit

Description

Rule Definition

The precedence of operators within expressions should be made explicit.

Rationale

The C language has a large number of operators and their precedence is not intuitive. Inexperienced programmers can easily make mistakes. Remove any ambiguity by using parentheses to explicitly define operator precedence.

The following table list the MISRA C® definition of operator precedence for this rule.

Description	Operator and Operand	Precedence
Primary	identifier, constant, string literal, (expression)	16
Postfix	`[]` `()` (function call) `.` `->` ++(post-increment) `--`(post-decrement) `()` `{}`(C99: compound literals)	15
Unary	++(post-increment) `--`(post-decrement) & `*` `+` `-` `~` `!` sizeof defined (preprocessor)	14
Cast	`()`	13
Multiplicative	`*` `/` `%`	12
Additive	`+` `-`	11
Bitwise shift	`<<` `>>`	10
Relational	`<>` `<=` `>=`	9
Equality	`==` `!=`	8
Bitwise AND	&	7

Description	Operator and Operand	Precedence
Bitwise XOR	^	6
Bitwise OR	|	5
Logical AND	&&	4
Logical OR	||	3
Conditional	?:	2
Assignment	= *= /= += -= <<= >>= &= ^= |=	1
Comma	,	0

Message in Report

Operand of logical %s is not a primary expression. The precedence of operators within expressions should be made explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Ambiguous Precedence in Multi-Operation Expressions

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof a + b;                     /* Non-compliant - MISRA-12.1 */

  x = a == b ? a : a - b;               /* Non-compliant - MISRA-12.1 */

  x = a <<  b + c ;                     /* Non-compliant - MISRA-12.1 */

  if (a || b && c) { }                  /* Non-compliant - MISRA-12.1 */
```

```
  if ( (a>x) && (b>x) || (c>x) )   { }  /* Non-compliant - MISRA-12.1 */
}
```

This example shows various violations of MISRA rule 12.1. In each violation, if you do not know the order of operations, the code could execute unexpectedly.

Correction — Clarify With Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
int a, b, c, d, x;

void foo(void) {
  x = sizeof(a) + b;

  x = ( a == b ) ? a : ( a - b );

  x = a << ( b + c );

  if ( ( a || b ) && c) { }

  if ( ((a>x) && (b>x)) || (c>x) ) { }
}
```

Ambiguous Precedence In Preprocessing Expressions

```
# if defined X && X + Y > Z    /* Non-compliant - MISRA-12.1 */
# endif

# if ! defined X && defined Y  /* Non-compliant - MISRA-12.1 */
# endif
```

In this example, two violations of MISRA rule 12.1 are shown in preprocessing code. In each violation, if you do not know the correct order of operations, the results can be unexpected and cause problems.

Correction — Clarify with Parentheses

To comply with this MISRA rule, add parentheses around individual operations in the expressions. One possible solution is shown here.

```
# if defined (X) && ( (X + Y) > Z )
# endif
```

```
# if ! defined (X) && defined (Y)
# endif
```

Compliant Expressions Without Parentheses

```
int a, b, c, x;
struct {int a; } s, *ps, *pp[2];

void foo(void) {
  ps = &s

  pp[i]-> a;          /* Compliant - no need to write (pp[i])->a */
  *ps++;              /* Compliant - no need to write *( p++ ) */

  x = f ( a + b, c ); /* Compliant - no need to write f ( (a+b),c) */

  x = a, b;           /* Compliant - parsed as ( x = a ), b */

  if (a && b && c ){  /* Compliant - all operators have
                       * the same precedence */
}
```

In this example, the expressions shown have multiple operations. However, these expressions are compliant because operator precedence is already clear.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 12.2 | MISRA C:2012 Rule 12.3 | MISRA C:2012 Rule 12.4

MISRA C:2012 Rule 12.2

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

Rationale

Consider the following statement:

```
var = abc << num;
```

If `abc` is a 16-bit integer, then `num` must be in the range 0–15, (nonnegative and less than 16). If `num` is negative or greater than 16, then the shift behavior is undefined.

Polyspace Implementation

In Polyspace, the numbers that are manipulated in preprocessing directives are 64 bits wide. The valid shift range is between 0 and 63. When bitfields are within a complex expression, Polyspace extends this check onto the bitfield field width or the width of the base type.

Message in Report

- Shift amount is bigger than *`size`*.
- Shift amount is negative.
- The right operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 12.1`

MISRA C:2012 Rule 12.3

The comma operator should not be used

Description

Rule Definition

The comma operator should not be used.

Rationale

The comma operator can be detrimental to readability. You can often write the same code in another form.

Message in Report

The comma operator should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Comma Usage in C Code

```
typedef signed int abc, xyz, jkl;

static void func1 ( abc, xyz, jkl );       /* Compliant - case 1 */

int foo(void)
{
```

```
    volatile int rd = 1;                  /* Compliant - case 2*/
    int var=0, foo=0, k=0, n=2, p, t[10]; /* Compliant - case 3*/

    int abc = 0, xyz = abc + 1;           /* Compliant - case 4*/
    int jkl = ( abc + xyz, abc + xyz );   /* Not compliant - case 1*/

    var = 1, foo += var, kkk = 3;         /* Not compliant - case 2*/
    var = (kkk = 1, foo = 2);             /* Not compliant - case 3*/

    for ( var = 0, ptr = &t[ 0 ]; var < num; ++var, ++ptr){}
                                          /* Not compliant - case 4*/

    if ((abc,xyz)<0) { return 1; }        /* Not compliant - case 5*/
}
```

In this example, the code shows various uses of commas in C code.

Noncompliant Cases

Case	Reason for noncompliance
1	When reading the code, it is not immediately obvious what `jkl` is initialized to. For example, you could infer that `jkl` has a value `abc+xyz`, `(abc+xyz)*(abc+xyz)`, `f((abc+xyz),(abc+xyz))`, and so on.
2	When reading the code, it is not immediately obvious whether `foo` has a value 0 or 1 after the statement.
3	When reading the code, it is not immediately obvious what value is assigned to `var`.
4	When reading the code, it is not immediately obvious which values control the `for` loop.
5	When reading the code, it is not immediately obvious whether the `if` statement depends on `abc`, `xyz`, or both.

Compliant Cases

Case	Reason for compliance
1	Using commas to call functions with variables is allowed.
2	Comma operator is not used.

Case	Reason for compliance
3 & 4	When using the comma for initialization, the variables and their values are immediately obvious.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 12.1`

MISRA C:2012 Rule 12.4

Evaluation of constant expressions should not lead to unsigned integer wrap-around

Description

Rule Definition

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wraparound. Although there may be good reasons to use modulo arithmetic at run time, intentional use at compile time is less likely.

Message in Report

Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 12.1

MISRA C:2012 Rule 12.5

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type"

Description

Rule Definition

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

Rationale

The `sizeof` operator acting on an array normally returns the array size in bytes. For instance, in the following code, `sizeof(arr)` returns the size of `arr` in bytes.

```
int32_t arr[4];
size_t numberOfElements = sizeof (arr) / sizeof(arr[0]);
```

However, when the array is a function parameter, it degenerates to a pointer. The `sizeof` operator acting on the array returns the corresponding pointer size and not the array size.

The use of `sizeof` operator on an array that is a function parameter typically indicates an unintended programming error.

Message in Report

The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect Use of sizeof Operator

```
int32_t glbA[] = { 1, 2, 3, 4, 5 };
void f (int32_t A[4])
{
         uint32_t numElements = sizeof(A) / sizeof(int32_t);  /* Non-compliant */
        uint32_t numElements_glbA = sizeof(glbA) / sizeof(glbA[0]);  /* Compliant */
}
```

In this example, the variable `numElements` always has the same value of 1, irrespective of the number of members that appear to be in the array (4 in this case), because `A` has type `int32_t *` and not `int32_t[4]`.

The variable `numElements_glbA` has the expected vale of 5 because the `sizeof` operator acts on the global array `glbA`.

Check Information

Group: Expressions
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2017a

MISRA C:2012 Rule 13.1

Initializer lists shall not contain persistent side effects

Description

Rule Definition

Initializer lists shall not contain persistent side effects.

Rationale

C99 permits initializer lists with expressions that can be evaluated only at run-time. However, the order in which elements of the list are evaluated is not defined. If one element of the list modifies the value of a variable which is used in another element, the ambiguity in order of evaluation causes undefined values. Therefore, this rule requires that expressions occurring in an initializer list cannot modify the variables used in them.

Message in Report

Initializer lists shall not contain persistent side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Initializers with Persistent Side Effect

```
volatile int v;
int x;
int y;
```

```
void f(void) {
    int arr[2] = {x+y,x-y};  /* Compliant */
    int arr2[2] = {v,0};      /* Non-compliant */
    int arr3[2] = {x++,y};    /* Non-compliant */
}
```

In this example, the rule is not violated in the first initialization because the initializer does not modify either `x` or `y`. The rule is violated in the other initializations.

- In the second initialization, because `v` is volatile, the initializer can modify `v`.
- In the third initialization, the initializer modifies the variable `x`.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required
Language: C99

See Also

`MISRA C:2012 Rule 13.2`

Introduced in R2014b

MISRA C:2012 Rule 13.2

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

Description

Rule Definition

The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, this rule forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation.

Message in Report

The value of 'XX' depends on the order of evaluation. The value of volatile 'XX' depends on the order of evaluation because of multiple accesses.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Noncompliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                 /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Side Effects
Category: Required

AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.9` | `MISRA C:2012 Rule 13.1` | `MISRA C:2012 Rule 13.3` | `MISRA C:2012 Rule 13.4`

Introduced in R2014b

MISRA C:2012 Rule 13.3

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

Description

Rule Definition

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Rationale

The rule is violated if the following happens in the same line of code:

- The increment or decrement operator acts on a variable.
- Another read or write operation is performed on the variable.

For example, the line `y=x++` violates this rule. The `++` and `=` operator both act on `x`.

Although the operator precedence rules determine the order of evaluation, placing the `++` and another operator in the same line can reduce the readability of the code.

Message in Report

A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Increment Operator Used in Expression with Other Side Effects

```
int input(void);
int choice(void);
int operation(int, int);

int func() {
    int x = input(), y = input(), res;
    int ch = choice();
    if (choice == -1)
        return(x++);
    if (choice == 0) {
        res = x++ + y++;
        return(res);           /* Non-compliant */
    }
    else if (choice == 1) {
        x++;                   /* Compliant */
        y++;                   /* Compliant */
        return (x+y);
    }
    else {
        res = operation(x++,y);
        return(res);           /* Non-compliant */
    }
}
```

In this example, the rule is violated when the expressions containing the `++` operator have side effects other than that caused by the operator. For example, in the expression `return(x++)`, the other side-effect is the `return` operation.

Check Information

Group: Side Effects
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 13.2`

Introduced in R2014b

MISRA C:2012 Rule 13.4

The result of an assignment operator should not be used

Description

Rule Definition

The result of an assignment operator should not be used.

Rationale

The rule is violated if the following happens in the same line of code:

- The assignment operator acts on a variable.
- Another read or operation is performed on the result of the assignment.

For example, the line `a[x]=a[x=y];` violates this rule. The `[]` operator acts on the result of the assignment `x=y`.

Message in Report

The result of an assignment operator should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Result of Assignment Used

```
int x, y, b, c, d;
int a[10];
```

```
unsigned int bool_var, false=0, true=1;

int foo(void) {

    x = y;            /* Compliant - x is not used */

    a[x] = a[x = y];  /* Non-compliant - Value of x=y is used */

    if ( bool_var = false ) {}
                     /* Non-compliant - bool_var=false is used */

    if ( bool_var == false ) {}   /* Compliant */

    if ( ( 0u == 0u ) || ( bool_var = true ) ) {}
    /* Non-compliant - even though (bool_var=true) is not evaluated */

    if ( ( x = f () ) != 0 ) {}
                   /* Non-compliant - value of x=f() is used */

    a[b += c] = a[b];
                   /* Non-compliant - value of b += c is used */

    b = c = d = 0; /* Non-compliant - value of d=0 and c=d=0 are used */

}
```

In this example, the rule is violated when the result of an assignment is used.

Check Information

Group: Side Effects
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 13.2

Introduced in R2014b

MISRA C:2012 Rule 13.5

The right hand operand of a logical && or || operator shall not contain persistent side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain persistent side effects.

Rationale

The right operand of an `||` operator is not evaluated if the left operand is true. The right operand of an && operator is not evaluated if the left operand is false. In these cases, if the right operand modifies the value of a variable, the modification does not take place. Following the operation, if you expect a modified value of the variable, the modification might not always happen.

Polyspace Implementation

- For this rule, Polyspace considers that all function calls have a persistent side effect.

  If a pure function is flagged, before ignoring this rule violation, make sure that the function has no side effects. For instance, floating-point functions such as `abs()` seem to only return a value and have no other side effect. However, these functions make use of the FPU Register Stack and can have side-effects in certain architectures, for instance, certain Intel® architectures.
- If the right operand is a volatile variable, Polyspace does not flag this as a rule violation.

Message in Report

The right hand operand of a && operator shall not contain side effects. The right hand operand of a || operator shall not contain side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Right Operand of Logical Operator with Persistent Side Effects

```
int check (int arg) {
    static int count;
    if(arg > 0) {
        count++;                   /* Persistent side effect */
        return 1;
    }
    else
        return 0;
}

int getSwitch(void);
int getVal(void);

void main(void) {
    int val = getVal();
    int mySwitch = getSwitch();
    int checkResult;

    if(mySwitch && check(val)) {   /* Non-compliant */
    }

    checkResult = check(val);
    if(checkResult && mySwitch) {  /* Compliant */
    }

    if(check(val) && mySwitch) {   /* Compliant */
    }
}
```

In this example, the rule is violated when the right operand of the && operation contains a function call. The function call has a persistent side effect because the static variable `count` is modified in the function body. Depending on `mySwitch`, this modification might or might not happen.

The rule is not violated when the left operand contains a function call. Alternatively, to avoid the rule violation, assign the result of the function call to a variable. Use this variable in the logical operation in place of the function call.

In this example, the function call has the side effect of modifying a `static` variable. Polyspace flags all function calls when used on the right-hand side of a logical && or `||` operator, even when the function does not have a side effect. Manually inspect your function body to see if it has side effects. If the function does not have side effects, add a comment and justification in your Polyspace result explaining why you retained your code.

Check Information

Group: Side Effects
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 13.6

The operand of the sizeof operator shall not contain any expression which has potential side effects

Description

Rule Definition

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Rationale

The argument of a `sizeof` operator is usually not evaluated at run time. If the argument is an expression, you might wrongly expect that the expression is evaluated.

Polyspace Implementation

The rule is not violated if the argument is a `volatile` variable.

Message in Report

The operand of the sizeof operator shall not contain any expression which has potential side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Expressions in `sizeof` Operator

```
#include <stddef.h>
int x;
int y[40];
struct S {
    int a;
    int b;
};
struct S myStruct;

void main() {
    size_t sizeOfType;
    sizeOfType = sizeof(x);         /* Compliant */
    sizeOfType = sizeof(y);         /* Compliant */
    sizeOfType = sizeof(myStruct);  /* Compliant */
    sizeOfType = sizeof(x++);       /* Non-compliant */
}
```

In this example, the rule is violated when the expression `x++` is used as argument of `sizeof` operator.

Check Information

Group: Side Effects
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 18.8`

Introduced in R2014b

MISRA C:2012 Rule 14.1

A loop counter shall not have essentially floating type

Description

Rule Definition

A loop counter shall not have essentially floating type.

Rationale

When using a floating-point loop counter, accumulation of rounding errors can result in a mismatch between the expected and actual number of iterations. This rounding error can happen when a loop step that is not a power of the floating point radix is rounded to a value that can be represented by a float.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it can give a different number of iteration on another implementation.

Polyspace Implementation

If the `for` index is a variable symbol, Polyspace checks that it is not a float.

Message in Report

A loop counter shall not have essentially floating type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

`for` Loop Counters

```
int main(void){
    unsigned int counter = 0u;
    int result = 0;
    float foo;

    // Float loop counters
    for(float foo = 0.0f; foo < 1.0f; foo +=0.001f){
        /* Non-compliant - counter = 1000 at the end of the loop */
        ++counter;
    }

    float fff = 0.0f;
    for(fff = 0.0f; fff <12.0f; fff += 1.0f){    /* Non-compliant*/
        result++;
    }

    // Integer loop count
    for(unsigned int count = 0u; count < 1000u; ++count){ /* Compliant */
        foo = (float) count * 0.001f;
    }
}
```

In this example, the three `for` loops show three different loop counters. The first and second `for` loops use float variables as loop counters, and therefore are not compliant. The third loop uses the integer `count` as the loop counter. Even though `count` is used as a float inside the loop, the variable remains an integer when acting as the loop index. Therefore, this `for` loop is compliant.

`while` Loop Counters

```
int main(void){
    unsigned int u32a;
    float foo;

    foo = 0.0f;
    while (foo < 1.0f){
        foo += 0.001f;  /* Non-compliant - foo used as a loop counter */
    }
```

```
    foo = read_float32();
    do{
        u32a = read_u32();
    }while( ((float)u32a - foo) > 10.0f );
                        /* Compliant - foo doesn't change in the loop */
                        /*  so cannot be a counter */
    return 1;
}
```

This example shows two `while` loops both of which use `foo` in the `while`-loop conditions.

The first `while` loop uses `foo` in the condition and inside the loop. Because `foo` changes, floating-point rounding errors can cause unexpected behavior.

The second `while` loop does not use `foo` inside the loop, but does use `foo` inside the `while`-condition. So `foo` is not the loop counter. The integer `u32a` is the loop counter because it changes inside the loop and is part of the while condition. Because `u32a` is an integer, the rounding error issue is not a concern, making this `while` loop compliant.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 14.2`

MISRA C:2012 Rule 14.2

A for loop shall be well-formed

Description

Rule Definition

A for loop shall be well-formed.

Rationale

The `for` statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyze.

Polyspace Implementation

Polyspace checks that:

- The `for` loop index (`V`) is a variable symbol.
- `V` is the last assigned variable in the first expression (if present).
- If the first expression exists, it contains an assignment of `V`.
- If the second expression exists, it is a comparison of `V`.
- If the third expression exists, it is an assignment of `V`.
- There are no direct assignments of the `for` loop index.

Message in Report

- 1st expression should be an assignment. The following kinds of for loops are allowed:
  - all three expressions shall be present;
  - the 2nd and 3rd expressions shall be present with prior initialization of the loop counter;

  - all three expressions shall be empty for a deliberate infinite loop.
- 3rd expression should be an assignment of a loop counter.
- 3rd expression : assigned variable should be the loop counter (*`counter`*).
- 3rd expression should be an assignment of loop counter (*`counter`*) only.
- 2nd expression should contain a comparison with loop counter (*`counter`*).
- Loop counter (*`counter`*) should not be modified in the body of the loop.
- Bad type for loop counter (*`counter`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Altering the Loop Counter Inside the Loop

```
void foo(void){

    for(short index=0; index < 5; index++){  /* Non-compliant */
        index = index + 3;       /* Altering the loop counter */
    }
}
```

In this example, the loop counter `index` changes inside the `for` loop. It is hard to determine when the loop terminates.

Correction — Use Another Variable to Terminate Early

One possible correction is to use an extra flag to terminate the loop early.

In this correction, the second clause of the `for` loop depends on the counter value, `index < 5`, and upon an additional flag, `!flag`. With the additional flag, the for loop definition and counter remain readable, and you can escape the loop early.

```
#define FALSE 0
#define TRUE  1
```

```
void foo(void){

    int flag = FALSE;

    for(short index=0; (index < 5) && !flag; index++){ /* Compliant */
        if((index % 4) == 0){
            flag = TRUE;        /* allows early termination of loop */
        }
    }
}
```

for Loops With Empty Clauses

```
void foo(void)
    for(short index = 0; ; index++) {}   /* Non-compliant */

    for(short index = 0; index < 10;) {} /* Non-compliant */

    short index;
    for(; index < 10;) {}     /* Non-compliant */

    for(; index < 10; i++) {} /* Compliant */

    for(;;){}
          /* Compliant - Exception all three clauses can be empty */
}
```

This example shows `for` loops definitions with a variety of missing clauses. To be compliant, initialize the first clause variable before the `for` loop (line 9). However, you cannot have a `for` loop without the second or third clause.

The one exception is a `for` loop with all three clauses empty, so as to allow for infinite loops.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 14.1 | MISRA C:2012 Rule 14.3 | MISRA C:2012 Rule 14.4

MISRA C:2012 Rule 14.3

Controlling expressions shall not be invariant

Description

Rule Definition

Controlling expressions shall not be invariant.

Rationale

If the controlling expression, for example an `if` condition, has a constant value, the non-changing value can point to a programming error.

Polyspace Implementation

Polyspace Bug Finder™ and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Polyspace Bug Finder flags some violations of MISRA C 14.3 through the `Dead code` and `Useless if` checkers.

Polyspace Code Prover does not use gray code to flag MISRA C 14.3 violations. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

- Boolean operations whose results are invariant shall not be permitted.
- Expression is always true.
- Boolean operations whose results are invariant shall not be permitted.
- Expression is always false.

- Controlling expressions shall not be invariant.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 2.1` | `MISRA C:2012 Rule 14.2`

MISRA C:2012 Rule 14.4

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Description

Rule Definition

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type

Rationale

Strong typing requires the controlling expression on an `if` statement or iteration statement to have *essentially Boolean* type.

Polyspace Implementation

Polyspace does not flag integer constants, for example `if(2)`.

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `-boolean-types`.

Message in Report

The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Controlling Expression in `if`, `while`, and `for`

```
#include <stdbool.h>
#include <stdlib.h>

#define TRUE = 1

typedef _Bool bool_t;
extern bool_t flag;

void foo(void){
    int *p = 1;
    int *q = 0;
    int i = 0;
    while(p){}           /* Non-compliant - p is a pointer */

    while(q != NULL){}   /* Compliant */

    while(TRUE){}        /* Compliant */

    while(flag){}        /* Compliant */

    if(i){}              /* Non-compliant - int32_t is not boolean */

    if(i != 0){}         /* Compliant */

    for(int i=-10; i;i++){}   /* Non-compliant - int32_t is not boolean */

    for(int i=0; i<10;i++){}  /* Compliant */
}
```

This example shows various controlling expressions in `while`, `if`, and `for` statements.

The noncompliant statements (the first `while`, `if`, and `for` examples), use a single non-Boolean variable. If you use a single variable as the controlling statement, it must be essentially Boolean (lines 17 and 19). Boolean expressions are also compliant with MISRA.

Check Information

Group: Control Statement Expressions
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 14.2` | `MISRA C:2012 Rule 20.8`

MISRA C:2012 Rule 15.1

The goto statement should not be used

Description

Rule Definition

The goto statement should not be used.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand.

Message in Report

The `goto` statement should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;

label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;     /* Non-compliant */
```

```
    }

label2: {
        result++;
        goto label1;                /* Non-compliant */
    }
}
```

In this example, the rule is violated when `goto` statements are used.

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4`

Introduced in R2014b

MISRA C:2012 Rule 15.2

The goto statement shall jump to a label declared later in the same function

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. You can use a forward `goto` statement together with a backward one to implement iterations. Restricting backward `goto` statements ensures that you use only iteration statements provided by the language such as `for` or `while` to implement iterations. This restriction reduces visual complexity of the code.

Message in Report

The goto statement shall jump to a label declared later in the same function.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Backward `goto` Statements

```
void foo(void) {
    int i = 0, result = 0;
```

```
label1:
    for ( i; i < 5; i++ ) {
        if (i > 2) goto label2;   /* Compliant */
    }

label2: {
        result++;
        goto label1;               /* Non-compliant */
    }
}
```

In this example, the rule is violated when a `goto` statement causes a backward jump to `label1`.

The rule is not violated when a `goto` statement causes a forward jump to `label2`.

Check Information

Group: Control Flow
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 15.4`

Introduced in R2014b

MISRA C:2012 Rule 15.3

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Rationale

Unrestricted use of `goto` statements makes the program unstructured and difficult to understand. Restricting use of `goto` statements to jump between blocks or into nested blocks reduces visual code complexity.

Message in Report

Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

`goto` Statements Jump Inside Block

```
void f1(int a) {
    if(a <= 0) {
```

```
        goto L2;        /* Non-compliant - L2 in different block*/
    }

    goto L1;            /* Compliant - L1 in same block*/

    if(a == 0) {
        goto L1;        /* Compliant - L1 in outer block*/
    }

    goto L2;            /* Non-compliant - L2 in inner block*/

    L1: if(a > 0) {
            L2:;
    }
}
```

In this example, `goto` statements cause jumps to different labels. The rule is violated when:

- The label occurs in a block different from the block containing the `goto` statement.

  The block containing the label neither encloses nor is enclosed by the current block.
- The label occurs in a block enclosed by the block containing the `goto` statement.

The rule is not violated when:

- The label occurs in the same block as the block containing the `goto` statement..
- The label occurs in a block that encloses the block containing the `goto` statement..

goto Statements in switch Block

```
void f2 ( int x, int z ) {
    int y = 0;

    switch(x) {
    case 0:
        if(x == y) {
            goto L1;  /* Non-compliant - switch-clauses are treated as blocks */
        }
        break;
    case 1:
        y = x;
        L1: ++x;
```

```
        break;
    default:
        break;
    }

}
```

In this example, the label for the `goto` statement appears to occur in a block that encloses the block containing the `goto` statement. However, for the purposes of this rule, the software considers that each `case` statement begins a new block. Therefore, the `goto` statement violates the rule.

Check Information

Group: Control Flow
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.4` | `MISRA C:2012 Rule 16.1`

Introduced in R2014b

MISRA C:2012 Rule 15.4

There should be no more than one break or goto statement used to terminate any iteration statement

Description

Rule Definition

There should be no more than one break or goto statement used to terminate any iteration statement.

Rationale

If you use one `break` or `goto` statement in your loop, you have one secondary exit point from the loop. Restricting number of exits from a loop in this way reduces visual complexity of your code.

Message in Report

There should be no more than one break or goto statement used to terminate any iteration statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

`break` Statements in Inner and Outer Loops

```
volatile int stop;
```

```
int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {   /* Compliant  */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) {  /* Compliant */
            if(stop)
                break;
            sum += arr[j];
        }
    }
}
```

In this example, the rule is not violated in both the inner and outer loop because both loops have one `break` statement each.

break and goto Statements in Loop

```
volatile int stop;

void displayStopMessage();

int func(int *arr, int size, int sat) {
    int i;
    int sum = 0;
    for (i=0; i< size; i++) {   /* Non-compliant  */
        if(sum >= sat)
            break;
        if(stop)
            goto L1;
        sum += arr[i];
    }

    L1: displayStopMessage();
}
```

In this example, the rule is violated because the `for` loop has one `break` statement and one `goto` statement.

goto Statement in Inner Loop and break Statement in Outer Loop

```
volatile int stop;

void displayMessage();

int func(int *arr, int size, int sat) {
    int i,j;
    int sum = 0;
    for (i=0; i< size; i++) {  /* Non-compliant */
        if(sum >= sat)
            break;
        for (j=0; j< i; j++) { /* Compliant */
            if(stop)
                goto L1;
            sum += arr[i];
        }
    }

    L1: displayMessage();
}
```

In this example, the rule is not violated in the inner loop because you can exit the loop only through the one `goto` statement. However, the rule is violated in the outer loop because you can exit the loop through either the `break` statement or the `goto` statement in the inner loop.

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.2` | `MISRA C:2012 Rule 15.3`

Introduced in R2014b

MISRA C:2012 Rule 15.5

A function should have a single point of exit at the end

Description

Rule Definition

A function should have a single point of exit at the end.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Message in Report

A function should have a single point of exit at the end.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

More Than One `return` Statement in Function

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0
```

```
typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;

bool_t f1(unsigned short n, char *p) {          /* Non-compliant */
    if(n > MAX) {
        return false;
    }

    if(p == NULL) {
        return false;
    }

    return true;
}
```

In this example, the rule is violated because there are three `return` statements.

Correction — Use Variable to Store Return Value

One possible correction is to store the return value in a variable and return this variable just before the function ends.

```
#define MAX ((unsigned int)2147483647)
#define NULL (void*)0

typedef unsigned int bool_t;
bool_t false = 0;
bool_t true = 1;
bool_t return_value;

bool_t f2 (unsigned short n, char *p) {         /* Compliant */
    return_value = true;
    if(n > MAX) {
        return_value = false;
    }

    if(p == NULL) {
        return_value = false;
    }

    return return_value;
}
```

Check Information

Group: Control Flow
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 17.4`

Introduced in R2014b

MISRA C:2012 Rule 15.6

The body of an iteration-statement or a selection-statement shall be a compound statement

Description

Rule Definition

The body of an iteration-statement or a selection-statement shall be a compound-statement.

Rationale

The rule applies to:

- Iteration statements such as `while`, `do ... while` or `for`.
- Selection statements such as `if ... else` or `switch`.

If the block of code associated with an iteration or selection statement is not contained in braces, you can make mistakes about the association. For example:

- You can wrongly associate a line of code with an iteration or selection statement because of its indentation.
- You can accidentally place a semicolon following the iteration or selection statement. Because of the semicolon, the line following the statement is no longer associated with the statement even though you intended otherwise.

Message in Report

- The else keyword shall be followed by either a compound statement, or another if statement.
- An if (expression) construct shall be followed by a compound statement.
- The statement forming the body of a while statement shall be a compound statement.

- The statement forming the body of a do ... while statement shall be a compound statement.
- The statement forming the body of a for statement shall be a compound statement.
- The statement forming the body of a switch statement shall be a compound statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Iteration Block

```
int data_available = 1;
void f1(void) {
    while(data_available)                /* Non-compliant */
        process_data();

    while(data_available) {              /* Compliant */
        process_data();
    }
}
```

In this example, the second `while` block is enclosed in braces and does not violate the rule.

Nested Selection Statements

```
void f1(void) {
    if(flag_1)                           /* Non-compliant */
        if(flag_2)                       /* Non-compliant */
            action_1();
    else                                 /* Non-compliant */
            action_2();
}
```

In this example, the rule is violated because the `if` or `else` blocks are not enclosed in braces. Unless indented as above, it is easy to associate the `else` statement with the inner `if`.

Correction — Place Selection Statement Block in Braces

One possible correction is to enclose each block associated with an `if` or `else` statement in braces.

```
void f1(void) {
    if(flag_1) {                        /* Compliant */
        if(flag_2) {                      /* Compliant */
            action_1();
        }
    }
    else {                              /* Compliant */
        action_2();
    }
}
```

Spurious Semicolon After Iteration Statement

```
void f1(void) {
    while(flag_1);                      /* Non-compliant */
    {
        flag_1 = action_1();
    }
}
```

In this example, the rule is violated even though the `while` statement is followed by a block in braces. The semicolon following the `while` statement causes the block to dissociated from the `while` statement.

The rule helps detect such spurious semicolons.

Check Information

Group: Control Flow
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 15.7

All if … else if constructs shall be terminated with an else statement

Description

Rule Definition

All if … else if constructs shall be terminated with an else statement.

Rationale

Unless there is a terminating `else` statement in an `if...elseif...else` construct, during code review, it is difficult to tell if you considered all possible results for the `if` condition.

Message in Report

All if … else if constructs shall be terminated with an else statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Missing `else` Block

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);
```

```
void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
        /* Non-compliant */
        action_2();
    }
}
```

In this example, the rule is violated because the `if ... else if` construct does not have a terminating `else` block.

Correction — Add `else` Block

To avoid the rule violation, add a terminating `else` block. The block can be empty.

```
int get_flag_1(void);
int get_flag_2(void);
void action_1(void);
void action_2(void);

void f1(void) {
    int flag_1 = get_flag_1(), flag_2 = get_flag_2();
    if(flag_1) {
        action_1();
    }
    else if(flag_2) {
        /* Non-compliant */
        action_2();
    }
    else {
        /* No statement required */
        /* ; is optional */
    }

}
```

Check Information

Group: Control Flow
Category: Required

AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 16.5`

Introduced in R2014b

MISRA C:2012 Rule 16.1

All switch statements shall be well-formed

Description

Rule Definition

All switch statements shall be well-formed

Rationale

The syntax for switch statements in C is not particularly rigorous and can allow complex, unstructured behavior. This rule and other rules impose a simple consistent structure on the `switch` statement.

Polyspace Implementation

Following the MISRA specifications, the coding rules checker also raises a violation of rule 16.1 if a `switch` statement violates one of these rules: 16.2, 16.3, 16.4, 16.5 or 16.6.

Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.3` | `MISRA C:2012 Rule 16.2` | `MISRA C:2012 Rule 16.3` | `MISRA C:2012 Rule 16.4` | `MISRA C:2012 Rule 16.5` | `MISRA C:2012 Rule 16.6`

MISRA C:2012 Rule 16.2

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Rationale

The C Standard permits placing a switch label (for instance, `case` or `default`) before any statement contained in the body of a switch statement. This flexibility can lead to unstructured code. To prevent unstructured code, make sure a switch label appears only at the outermost level of the body of a switch statement.

Message in Report

All messages in report file begin with "MISRA-C switch statements syntax normative restriction."

- Initializers shall not be used in switch clauses.
- The child statement of a switch shall be a compound statement.
- All switch clauses shall appear at the same level.
- A switch clause shall only contain switch labels and switch clauses, and no other code.
- A switch statement shall only contain switch labels and switch clauses, and no other code.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule 16.3

An unconditional break statement shall terminate every switch-clause

Description

Rule Definition

An unconditional break statement shall terminate every switch-clause

Rationale

A *switch-clause* is a case containing at least one statement. Two consecutive labels without an intervening statement is compliant with MISRA.

If you fail to end your switch-clauses with a break statement, then control flow "falls" into the next statement. This next statement can be another switch-clause, or the end of the switch. This behavior is sometimes intentional, but more often it is an error. If you add additional cases later, an unterminated switch-clause can cause problems.

Polyspace Implementation

Polyspace raises a warning for each noncompliant `case` clause.

Message in Report

An unconditional break statement shall terminate every switch-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule 16.4

Every switch statement shall have a default label

Description

Rule Definition

Every switch statement shall have a default label

Rationale

The requirement for a `default` label is defensive programming. Even if your switch covers all possible values, there is no guarantee that the input takes one of these values. Statements following the `default` label take some appropriate action. If the `default` label requires no action, use comments to describe why there are no specific actions.

Message in Report

Every switch statement shall have a default label.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Switch Statement Without `default`

```
short func1(short xyz){

    switch(xyz){      /* Non-compliant - default label is required */
        case 0:
```

```
            ++xyz;
            break;
        case 1:
        case 2:
            break;
    }
    return xyz;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant.

Correction — Add `default` With Error Flag

One possible correction is to use the `default` label to flag input errors. If your switch-clauses cover all expected input, then the default cases flags any input errors.

```
short func1(short xyz){

    switch(xyz){      /* Compliant */
        case 0:
            ++xyz;
            break;
        case 1:
        case 2:
            break;
        default:
            errorflag = 1;
            break;
    }
    if (errorflag == 1)
        return errorflag;
    else
        return xyz;
}
```

Switch Statement for Enumerated Inputs

```
enum Colors{
    RED, GREEN, BLUE
};

enum Colors func2(enum Colors color){
    enum Colors next;
```

```
    switch(color){      /* Non-compliant - default label is required */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
    }
    return next;
}
```

In this example, the switch statement does not include a `default` label, and is therefore noncompliant. Even though this switch statement handles all values of the enumeration, there is no guarantee that color takes one of the those values.

Correction — Add `default`

To be compliant, add the `default` label to the end of your switch. You can use this case to flag unexpected inputs.

```
enum Colors{
    RED, GREEN, BLUE, ERROR
};

enum Colors func2(enum Colors color){
    enum Colors next;

    switch(color){      /* Compliant */
        case RED:
            next = GREEN;
            break;
        case GREEN:
            next = BLUE;
            break;
        case BLUE:
            next = RED;
            break;
        default:
            next = ERROR;
            break;
```

```
    }

    return next;
}
```

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 2.1 | MISRA C:2012 Rule 16.1

MISRA C:2012 Rule 16.5

A default label shall appear as either the first or the last switch label of a switch statement

Description

Rule Definition

A default label shall appear as either the first or the last switch label of a switch statement.

Rationale

Using this rule, you can easily locate the `default` label within a `switch` statement.

Message in Report

A default label shall appear as either the first or the last switch label of a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Default Case in `switch` Statements

```
void foo(int var){

    switch(var){
        default:   /* Compliant - default is the first label */
```

```
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        default:    /* Non-compliant - default is mixed with the case labels */
        case 1:
        case 2:
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
        default:     /* Compliant - default is the last label */
            break;
    }

    switch(var){
        case 0:
            ++var;
            break;
        case 1:
        case 2:
            break;
        default:      /* Compliant - default is the last label */
            var = 0;
            break;
    }
}
```

This example shows the same switch statement several times, each with `default` in a different place. As the first, third, and fourth switch statements show, `default` must be the first or last label. `default` can be part of a compound switch-clause (for instance, the third `switch` example), but it must be the last listed.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.7` | `MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule 16.6

Every switch statement shall have at least two switch-clauses

Description

Rule Definition

Every switch statement shall have at least two switch-clauses.

Rationale

A switch statement with a single path is redundant and can indicate a programming error.

Message in Report

Every switch statement shall have at least two switch-clauses.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 16.1`

MISRA C:2012 Rule 16.7

A switch-expression shall not have essentially Boolean type

Description

Rule Definition

A switch-expression shall not have essentially Boolean type

Rationale

The C Standard requires the controlling expression to a `switch` statement to have an integer type. Because C implements Boolean values with integer types, it is possible to have a Boolean expression control a `switch` statement. For controlling flow with Boolean types, an `if-else` construction is more appropriate.

Polyspace Implementation

The analysis recognizes the Boolean types, `bool` or `_Bool` (defined in `stdbool.h`)

You can also define types that are essentially Boolean using the option `-boolean-types`.

Message in Report

A switch-expression shall not have essentially Boolean type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Switch Statements
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 17.1

The features of <stdarg.h> shall not be used

Description

Rule Definition

The features of <stdarg.h> shall not be used..

Rationale

The rule forbids use of `va_list`, `va_arg`, `va_start`, `va_end`, and `va_copy`.

You can use these features in ways where the behavior is not defined in the Standard. For instance:

- You invoke `va_start` in a function but do not invoke the corresponding `va_end` before the function block ends.
- You invoke `va_arg` in different functions on the same variable of type `va_list`.
- `va_arg` has the syntax `type va_arg (va_list ap, type)`.

  You invoke `va_arg` with a `type` that is incompatible with the actual type of the argument retrieved from `ap`.

Message in Report

The features of <stdarg.h> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of va_start, va_list, va_arg, and va_end

```
#include<stdarg.h>
void f2(int n, ...) {
    int i;
    double val;
    va_list vl;                         /* Non-compliant */

    va_start(vl, n);                    /* Non-compliant */

    for(i = 0; i < n; i++)
    {
        val = va_arg(vl, double);         /* Non-compliant */
    }

    va_end(vl);                         /* Non-compliant */
}
```

In this example, the rule is violated because `va_start`, `va_list`, `va_arg` and `va_end` are used.

Undefined Behavior of va_arg

```
#include <stdarg.h>
void h(va_list ap) {                    /* Non-compliant */
    double y;

    y = va_arg(ap, double );              /* Non-compliant */
}

void g(unsigned short n, ...) {
    unsigned int x;
    va_list ap;                         /* Non-compliant */

    va_start(ap, n);                    /* Non-compliant */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

    h(ap);
```

```
    /* Undefined - ap is indeterminate because va_arg used in h () */
    x = va_arg(ap, unsigned int);       /* Non-compliant */

}

void f(void) {
    /* undefined - uint32_t:double type mismatch when g uses va_arg () */
    g(1, 2.0, 3.0);
}
```

In this example, `va_arg` is used on the same variable `ap` of type `va_list` in both functions `g` and `h`. In `g`, the second argument is `unsigned int` and in `h`, the second argument is `double`. This type mismatch causes undefined behavior.

Check Information

Group: Function
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 17.2

Functions shall not call themselves, either directly or indirectly

Description

Rule Definition

Functions shall not call themselves, either directly or indirectly.

Rationale

Variables local to a function are stored in the call stack. If a function calls itself directly or indirectly several times, the available stack space can be exceeded, causing serious failure. Unless the recursion is tightly controlled, it is difficult to determine the maximum stack space required.

Message in Report

Message in Report: Function XX shall not call itself either directly or indirectly.
Function XX is called indirectly by YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Direct and Indirect Recursion

```
void foo1( void ) {     /* Non-compliant - Indirect recursion foo1->foo2->foo1... */
    foo2();
    foo1();               /* Non-compliant - Direct recursion */
```

```
}

void foo2( void ) {
    foo1();
}
```

In this example, the rule is violated because of:

- Direct recursion `foo1` → `foo1`.
- Indirect recursion `foo1` → `foo2` → `foo1`.

Check Information

Group: Function
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 17.3

A function shall not be declared implicitly

Description

Rule Definition

A function shall not be declared implicitly.

Rationale

An implicit declaration occurs when you call a function before declaring or defining it. When you declare a function explicitly before calling it, the compiler can match the argument and return types with the parameter types in the declaration. If an implicit declaration occurs, the compiler makes assumptions about the argument and return types. For instance, it assumes a return type of `int`. The assumptions might not agree with what you expect and cause undesired type conversions.

Polyspace Specification

To enable checking of this rule, use the value `c90` for the option `C standard version (-c-version)`.

Message in Report

Function 'XX' has no complete visible prototype at call.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function Not Declared Before Call

```
#include <math.h>

extern double power3 (double val, int exponent);
int getChoice(void);

double func() {
    double res;
    int ch = getChoice();
    if(ch == 0) {
        res = power(2.0, 10);    /* Non-compliant */
    }
    else if( ch==1) {
        res = power2(2.0, 10);   /* Non-compliant */
    }
    else {
        res = power3(2.0, 10);   /* Compliant */
        return res;
    }
}

double power2 (double val, int exponent) {
    return (pow(val, exponent));
}
```

In this example, the rule is violated when a function that is not declared is called in the code. Even if a function definition exists later in the code, the rule violation occurs.

The rule is not violated when the function is declared before it is called in the code. If the function definition exists in another file and is available only during the link phase, you can declare the function in one of the following ways:

- Declare the function with the `extern` keyword in the current file.
- Declare the function in a header file and include the header file in the current file.

Check Information

Group: Function

Category: Mandatory
AGC Category: Mandatory
Language: C90

See Also

`MISRA C:2012 Rule 8.2` | `MISRA C:2012 Rule 8.4`

Introduced in R2014b

MISRA C:2012 Rule 17.4

All exit paths from a function with non-void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rationale

If a non-`void` function does not explicitly return a value but the calling function uses the return value, the behavior is undefined. To prevent this behavior:

1. You must provide `return` statements with an explicit expression.
2. You must ensure that during run time, at least one `return` statement executes.

Message in Report

Missing return value for non-void function 'XX'.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Missing Return Statement Along Certain Execution Paths

```
int absolute(int v) {
    if(v < 0) {
```

```
        return v;
    }
}
```

In this example, the rule is violated because a `return` statement does not exist on all execution paths. If `v >= 0`, then the control returns to the calling function without an explicit return value.

Return Statement Without Explicit Expression

```
#define SIZE 10
int table[SIZE];

unsigned short lookup(unsigned short v) {
    if((v < 0) || (v > SIZE)) {
        return;
    }
    return table[v];
}
```

In this example, the rule is violated because the `return` statement in the `if` block does not have an explicit expression.

Check Information

Group: Function
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.5`

Introduced in R2014b

MISRA C:2012 Rule 17.5

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements

Description

Rule Definition

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

Rationale

If you use an array declarator for a function parameter instead of a pointer, the function interface is clearer because you can state the minimum expected array size. If you do not state a size, the expectation is that the function can handle an array of any size. In such cases, the size value is typically another parameter of the function, or the array is terminated with a sentinel value.

However, it is legal in C to specify an array size but pass an array of smaller size. This rule prevents you from passing an array of size smaller than the size you declared.

Message in Report

The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

The argument type has *`actual_size`* elements whereas the parameter type expects *`expected_size`* elements.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect Array Size Passed to Function

```
void func(int arr[4]);

int main() {
    int arrSmall[3] = {1,2,3};
    int arr[4] = {1,2,3,4};
    int arrLarge[5] ={1,2,3,4,5};

    func(arrSmall);      /* Non-compliant */
    func(arr);           /* Compliant */
    func(arrLarge);      /* Compliant */

    return 0;
}
```

In this example, the rule is violated when `arrSmall`, which has size 3, is passed to `func`, which expects at least 4 elements.

Check Information

Group: Functions
Category: Advisory
AGC Category: Readability
Language: C90. C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 17.6

The declaration of an array parameter shall not contain the static keyword between the [ ]

Description

Rule Definition

The declaration of an array parameter shall not contain the static keyword between the [ ].

Rationale

If you use the `static` keyword within `[]` for an array parameter of a function, you can inform a C99 compiler that the array contains a minimum number of elements. The compiler can use this information to generate efficient code for certain processors. However, in your function call, if you provide less than the specified minimum number, the behavior is not defined.

Message in Report

The declaration of an array parameter shall not contain the static keyword between the [ ].

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of `static` Keyword Within `[]` in Array Parameter

```
extern int arr1[20];
extern int arr2[10];
```

```
/* Non-compliant: static keyword used in array declarator */
unsigned int total (unsigned int n, unsigned int arr[static 20]) {
    unsigned int i;
    unsigned int sum = 0;

    for (i=0U; i < n; i++) {
        sum+= arr[i];
    }

    return sum;
}

void func (void) {
    int res, res2;
    res = total (10U, arr1);  /* Non-compliant - behavior not defined */
    res2 = total (20U, arr2); /* Non-compliant, even if behavior is defined */
}
```

In this example, the rule is violated when the `static` keyword is used within `[]` in the array parameter of function `total`. Even if you call `total` with array arguments where the behavior is well-defined, the rule violation occurs.

Check Information

Group: Function
Category: Mandatory
AGC Category: Mandatory
Language: C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 17.7

The value returned by a function having non-void return type shall be used

Description

Rule Definition

The value returned by a function having non-void return type shall be used.

Rationale

You can unintentionally call a function with a non-`void` return type but not use the return value. Because the compiler allows the call, you might not catch the omission. This rule forbids calls to a non-`void` function where the return value is not used. If you do not intend to use the return value of a function, explicitly cast the return value to `void`.

Message in Report

The value returned by a function having non-void return type shall be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Used and Unused Return Values

```
unsigned int cutOff(unsigned int val) {
    if (val > 10 && val < 100) {
        return val;
    }
```

```
    else {
        return 0;
    }
}

unsigned int getVal(void);

void func2(void) {
    unsigned int val = getVal(), res;
    cutOff(val);           /* Non-compliant */
    res = cutOff(val);     /* Compliant */
    (void)cutOff(val);     /* Compliant */
}
```

In this example, the rule is violated when the return value of `cutOff` is not used subsequently.

The rule is not violated when the return value is:

- Assigned to another variable.
- Explicitly cast to `void`.

Check Information

Group: Function
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 2.2`

Introduced in R2014b

MISRA C:2012 Rule 17.8

A function parameter should not be modified

Description

Rule Definition

A function parameter should not be modified.

Rationale

When you modify a parameter, the function argument corresponding to the parameter is not modified. However, you or another programmer unfamiliar with C can expect by mistake that the argument is also modified when you modify the parameter.

Message in Report

A function parameter should not be modified.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function Parameter Modified

```
int input(void);

void func(int param1, int* param2) {

    param1 = input();   /* Non-compliant */
```

```
    *param2 = input();  /* Compliant */
}
```

In this example, the rule is violated when the parameter `param1` is modified.

The rule is not violated when the parameter is a pointer `param2` and `*param2` is modified.

Check Information

Group: Functions
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 18.1

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

Description

Rule Definition

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Rationale

Using an invalid array subscript can lead to erroneous behavior of the program. Run-time derived array subscripts are especially troublesome because they cannot be easily checked by manual review or static analysis.

The C Standard defines the creation of a pointer to one beyond the end of the array. The rule permits the C Standard. Dereferencing a pointer to one beyond the end of an array causes undefined behavior and is noncompliant.

Polyspace Implementation

Polyspace flags this rule during the analysis as:

- Bug Finder — `Array access out-of-bounds` and `Pointer access out-of-bounds`.
- Code Prover — `Illegally dereferenced pointer` and `Out of bounds array index`.

Bug Finder and Code Prover check this rule differently and can show different results for this rule. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.1` | `MISRA C:2012 Rule 18.4`

MISRA C:2012 Rule 18.2

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Rationale

This rule applies to expressions of the form `pointer_expression1 - pointer_expression2`. The behavior is undefined if `pointer_expression1` and `pointer_expression2`:

- Do not point to elements of the same array,
- Or do not point to the element one beyond the end of the array.

Message in Report

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Subtracting Pointers

```
#include <stddef.h>

void f1 (int32_t *ptr)
{
    int32_t a1[10];
    int32_t a2[10];
    int32_t *p1 = &a1[ 1];
    int32_t *p2 = &a2[10];
    ptrdiff_t diff1, diff2, diff3;

    diff1 =  p1 - a1;   // Compliant
    diff2 =  p2 - a2;   // Compliant
    diff3 =  p1 - p2;   // Non-compliant
}
```

In this example, the three subtraction expressions show the difference between compliant and noncompliant pointer subtractions. The `diff1` and `diff2` subtractions are compliant because the pointers point to the same array. The `diff3` subtraction is not compliant because `p1` and `p2` point to different arrays.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.1` | `MISRA C:2012 Rule 18.4`

MISRA C:2012 Rule 18.3

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

Description

Rule Definition

The relational operators >, >=, <, and <= shall not be applied to objects of pointer type except where they point into the same object.

Rationale

If two pointers do not point to the same object, comparisons between the pointers produces undefined behavior.

You can address the element beyond the end of an array, but you cannot access this element.

Message in Report

The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Pointer and Array Comparisons

```
void f1(void){
    int arr1[10];
    int arr2[10];
    int *ptr1 = arr1;

    if(ptr1 < arr2){}    /* Non-compliant */
    if(ptr1 < arr1){}    /* Compliant */
}
```

In this example, `ptr1` is a pointer to `arr1`. To be compliant with rule 18.3, you can compare only `ptr1` with `arr1`. Therefore, the comparison between `ptr1` and `arr2` is noncompliant.

Structure Comparisons

```
struct limits{
  int lower_bound;
  int upper_bound;
};

void func2(void){
    struct limits lim_1 = { 2, 5 };
    struct limits lim_2 = { 10, 5 };

    if(&lim_1.lower_bound <= &lim_2.upper_bound){}  /* Non-compliant *
    if(&lim_1.lower_bound <= &lim_1.upper_bound){}  /* Compliant */
}
```

This example defines two `limits` structures, `lim1` and `lim2`, and compares the elements. To be compliant with rule 18.3, you can compare only the structure elements within a structure. The first comparison compares the `lower_bound` of `lim1` and the `upper_bound` of `lim2`. This comparison is noncompliant because the `lim_1.lower_bound` and `lim_2.upper_bound` are elements of two different structures.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.1`

MISRA C:2012 Rule 18.4

The +, -, += and -= operators should not be applied to an expression of pointer type

Description

Rule Definition

The +, -, += and -= operators should not be applied to an expression of pointer type.

Rationale

The preferred form of pointer arithmetic is using the array subscript syntax `ptr[expr]`. This syntax is clear and less prone to error than pointer manipulation. With pointer manipulation, any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Array indexing can also access unintended or invalid memory, but it is easier to review.

To a new C programmer, the expression `ptr+1` can be mistakenly interpreted as one plus the address of `ptr`. However, the new memory address depends on the size, in bytes, of the pointer's target. This confusion can lead to unexpected behavior.

When used with caution, pointer manipulation using ++ can be more natural (for instance, sequentially accessing locations during a memory test).

Polyspace Implementation

Polyspace flags operations on pointers, for example, `Pointer + Integer`, `Integer + Pointer`, `Pointer - Integer`.

Message in Report

The +, -, += and -= operators should not be applied to an expression of pointer type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Pointers and Array Expressions

```
void fun1(void){
    unsigned char arr[10];
    unsigned char *ptr;
    unsigned char index = 0U;

    index = index + 1U;   /* Compliant - rule only applies to pointers */

    arr[index] = 0U;      /* Compliant */
    ptr = &arr[5];        /* Compliant */
    ptr = arr;
    ptr++;                /* Compliant - increment operator not + */
    *(ptr + 5) = 0U;      /* Non-compliant */
    ptr[5] = 0U;          /* Compliant */
}
```

This example shows various operations with pointers and arrays. The only operation in this example that is noncompliant is using the + operator directly with a pointer (line 12).

Adding Array Elements Inside a `for` Loop

```
void fun2(void){
    unsigned char array_2_2[2][2] = {{1U, 2U}, {4U, 5U}};
    unsigned char i = 0U;
    unsigned char j = 0U;
    unsigned char sum = 0U;

    for(i = 0u; i < 2U; i++){
        unsigned char *row = array_2_2[ i ];

        for(j = 0u; j < 2U; j++){
            sum += row[ j ];                   /* Compliant */
        }
```

```
    }
}
```

In this example, the second `for` loop uses the array pointer `row` in an arithmetic expression. However, this usage is compliant because it uses the array index form.

Pointers and Array Expressions

```
void fun3(unsigned char *ptr1, unsigned char ptr2[ ]){
    ptr1++;               /* Compliant */
    ptr1 = ptr1 - 5;      /* Non-compliant */
    ptr1 -= 5;            /* Non-compliant */
    ptr1[2] = 0U;         /* Compliant */

    ptr2++;               /* Compliant */
    ptr2 = ptr2 + 3;      /* Non-compliant */
    ptr2 += 3;            /* Non-compliant */
    ptr2[3] = 0U;         /* Compliant */
}
```

This example shows the offending operators used on pointers and arrays. Notice that the same types of expressions are compliant and noncompliant for both pointers and arrays.

If `ptr1` does not point to an array with at least six elements, and `ptr2` does not point to an array with at least 4 elements, this example violates rule 18.1.

Check Information

Group: Pointers and Arrays
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 18.1` | `MISRA C:2012 Rule 18.2`

MISRA C:2012 Rule 18.5

Declarations should contain no more than two levels of pointer nesting

Description

Rule Definition

Declarations should contain no more than two levels of pointer nesting.

Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behavior of the code. Avoid this usage.

Message in Report

Declarations should contain no more than two levels of pointer nesting.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Pointer Nesting

```
typedef char *INTPTR;

void function(char ** arrPar[ ])    /* Non-compliant - 3 levels */
{
    char   **  obj2;            /* Compliant */
    char   *** obj3;            /* Non-compliant */
```

```
    INTPTR *   obj4;            /* Compliant */
    INTPTR * const * const obj5;    /* Non-compliant */
    char   ** arr[10];            /* Compliant */
    char   ** (*parr)[10];        /* Compliant */
    char   *  (**pparr)[10];        /* Compliant */
}

struct s{
    char *   s1;                /* Compliant */
    char **  s2;                /* Compliant */
    char *** s3;                /* Non-compliant */
};

struct s *   ps1;            /* Compliant */
struct s **  ps2;            /* Compliant */
struct s *** ps3;            /* Non-compliant */

char **  (  *pfunc1)(void);        /* Compliant */
char **  ( **pfunc2)(void);        /* Compliant */
char **  (***pfunc3)(void);        /* Non-compliant */
char *** ( **pfunc4)(void);        /* Non-compliant */
```

This example shows various pointer declarations and nesting levels. Any pointer with more than two levels of nesting is considered noncompliant.

Check Information

Group: Pointers and Arrays
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 18.6

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behavior.

Polyspace Implementation

Polyspace flags a violation when assigning an address to a global variable, returning a local variable address, or returning a parameter address.

Message in Report

The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Address of Local Variables

```
char *func(void){
    char local_auto;
    return &local_auto  /* Non-compliant
                         * &local_auto is indeterminate */
}
```

In this example, because `local_auto` is a local variable, after the function returns, the address of `local_auto` is indeterminate.

Copying Pointer Addresses to Local Variables

```
char *sp;

void f(unsigned short u){
    g(&u);
}

void g(unsigned short *p){
    sp = p;  /* Non-compliant
              * the parameter u from f is copied to static sp */
}

void h(void){
    static unsigned short *q;

    unsigned short x =0u;
    q = &x;  /* Non-compliant -
              * &x stored in object with greater lifetime */
}
```

In this example, the function `g` stores a copy of its pointer parameter `p`. If `p` always points to an object with static storage duration, then the code is compliant with this rule. However, in this example, `p` points to an object with automatic storage duration. In such a case, copying the parameter `p` is noncompliant.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 18.7

Flexible array members shall not be declared

Description

Rule Definition

Flexible array members shall not be declared.

Rationale

Flexible array members are usually used with dynamic memory allocation. Dynamic memory allocation is banned by Directive 4.12 and Rule 21.3 on page 2-241.

Message in Report

Flexible array members shall not be declared.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.3`

MISRA C:2012 Rule 18.8

Variable-length array types shall not be used

Description

Rule Definition

Variable-length array types shall not be used.

Rationale

When the size of an array declared in a block or function prototype is not an integer constant expression, you specify variable array types. Variable array types are typically implemented as a variable size object stored on the stack. Using variable type arrays can make it impossible to determine statistically the amount of memory for the stack requires.

If the size of a variable-length array is negative or zero, the behavior is undefined.

If a variable-length array must be compatible with another array type, then the size of the array types must be identical and positive integers. If your array does not meet these requirements, the behavior is undefined.

If you use a variable-length array type in a `sizeof`, it is uncertain if the array size is evaluated or not.

Message in Report

Variable-length array types shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Pointers and Arrays
Category: Required
AGC Category: Required
Language: C99

See Also

`MISRA C:2012 Rule 13.6`

MISRA C:2012 Rule 19.1

An object shall not be assigned or copied to an overlapping object

Description

Rule Definition

An object shall not be assigned or copied to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined. The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another using `memmove`.

Message in Report

- An object shall not be assigned or copied to an overlapping object.
- Destination and source of XX overlap, the behavior is undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Assignment of Union Members

```
void func (void) {
    union {
```

```
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   /* Non-compliant */
    a = b;       /* Compliant */
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Assignment of Array Segments

```
#include <string.h>

int arr[10];

void func(void) {
    memcpy (&arr[5], &arr[4], 2u * sizeof(arr[0]));    /* Non-compliant */
    memcpy (&arr[5], &arr[4], sizeof(arr[0]));         /* Compliant */
    memcpy (&arr[1], &arr[4], 2u * sizeof(arr[0]));    /* Compliant */
}
```

In this example, memory equal to twice `sizeof(arr[0])` is the memory space taken up by two array elements. If that memory space begins from `&a[4]` and `&a[5]`, the two memory regions overlap. The rule is violated when the `memcpy` function is used to copy the contents of these two overlapping memory regions.

Check Information

Group: Overlapping Storage
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 19.2`

Introduced in R2014b

MISRA C:2012 Rule 19.2

The union keyword should not be used

Description

Rule Definition

The union keyword should not be used.

Rationale

If you write to a union member and read the same union member, the behavior is well-defined. But if you read a different member, the behavior depends on the relative sizes of the members. For instance:

- If you read a union member with wider memory size, the value you read is unspecified.
- Otherwise, the value is implementation-dependant.

Message in Report

The union keyword should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Possible Problems with `union` Keyword

```
unsigned int zext(unsigned int s)
{
```

```
    union                 /* Non-compliant */
    {
        unsigned int ul;
        unsigned short us;
    } tmp;

    tmp.us = s;
    return tmp.ul;        /* Unspecified value */
}
```

In this example, the 16-bit `short` field `tmp.us` is written but the wider 32-bit `int` field `tmp.ul` is read. Using the `union` keyword can cause such unspecified behavior. Therefore, the rule forbids using the `union` keyword.

Check Information

Group: Overlapping Storage
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 19.1`

Introduced in R2014b

MISRA C:2012 Rule 2.1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

Unless a program exhibits any undefined behavior, unreachable code cannot execute. The unreachable code cannot affect the program output. The presence of unreachable code can indicate an error in the program logic. Unreachable code that the compiler does not remove wastes resources, for example:

- It occupies space in the target machine memory.
- Its presence can cause a compiler to select longer, slower jump instructions when transferring control around the unreachable code.
- Within a loop, it can prevent the entire loop from residing in an instruction cache.

Polyspace Implementation

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

The Code Prover run-time check for unreachable code shows more cases than the MISRA checker for rule 2.1. See also `Unreachable code`. The run-time check performs a more exhaustive analysis. In the process, the check can show some instances that are not strictly unreachable code but unreachable only in the context of the analysis. For instance, in the following code, the run-time check shows a potential division by zero in the first line and then removes the zero value of `flag` for the rest of the analysis. Therefore, it considers the `if` block unreachable.

```
val=1.0/flag;
if(!flag) {}
```

The MISRA checker is designed to prevent these kinds of results.

Message in Report

A project shall not contain unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Code Following `return` Statement

```
enum light { red, amber, red_amber, green };

enum light next_light ( enum light color )
{
    enum light res;

    switch ( color )
    {
    case red:
        res = red_amber;
        break;
    case red_amber:
        res = green;
        break;
    case green:
        res = amber;
        break;
    case amber:
        res = red;
        break;
    default:
    {
```

```
            error_handler ();
            break;
        }
        }

        res = color;
        return res;
        res = color;     /* Non-compliant */
}
```

In this example, the rule is violated because there is an unreachable operation following the `return` statement.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 14.3` | `MISRA C:2012 Rule 16.4`

Introduced in R2014b

MISRA C:2012 Rule 2.2

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Operations involving language extensions such as `__asm ( "NOP" );` are not considered dead code.

Polyspace Implementation

Polyspace Bug Finder detects useless write operations during analysis.

Polyspace Code Prover does not detect useless write operations. For instance, if you assign a value to a local variable but do not read it later, Polyspace Code Prover does not detect this useless assignment. Use Polyspace Bug Finder to detect such useless write operations.

In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. See the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

There shall be no dead code.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Redundant Operations

```
extern volatile unsigned int v;
extern char *p;

void f ( void ) {
    unsigned int x;


    ( void ) v;      /* Compliant - Exception*/
    ( int ) v;       /* Non-compliant  */
    v >> 3;          /* Non-compliant  */

    x = 3;           /* Non-compliant - Detected in Bug Finder only */

    *p++;            /* Non-compliant  */
    ( *p )++;        /* Compliant  */
}
```

In this example, the rule is violated when an operation is performed on a variable, but the result of that operation is not used. For instance,

- The operations `(int)` and `>>` on the variable `v` are redundant because the results are not used.
- The operation `=` is redundant because the local variable `x` is not read after the operation.
- The operation `*` on `p++` is redundant because the result is not used.

The rule is not violated when:

- A variable is cast to `void`. The cast indicates that you are intentionally not using the value.

- The result of an operation is used. For instance, the operation * on p is not redundant, because *p is incremented.

Redundant Function Call

```
void g ( void ) {
               /* Compliant  */
}

void h ( void) {
     g( );     /* Non-compliant */
}
```

In this example, g is an empty function. Though the function itself does not violate the rule, a call to the function violates the rule.

Check Information

Group: Unused Code
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 17.7

Introduced in R2014b

MISRA C:2012 Rule 2.3

A project should not contain unused type declarations

Description

Rule Definition

A project should not contain unused type declarations.

Rationale

If a type is declared but not used, a reviewer does not know if the type is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused type declarations: type XX is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Local Type

```
signed short unusedType (void){

    typedef signed short myType;   /* Non-compliant */
    return 67;

}
```

```
signed short usedType (void){

    typedef signed short myType;  /* Compliant */
    myType tempVar = 67;
    return tempVar;

}
```

In this example, in function `unusedType`, the `typedef` statement defines a new local type `myType`. However, this type is never used in the function. Therefore, the rule is violated.

The rule is not violated in the function `usedType` because the new type `myType` is used.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 2.4`

Introduced in R2014b

MISRA C:2012 Rule 2.4

A project should not contain unused tag declarations

Description

Rule Definition

A project should not contain unused tag declarations.

Rationale

If a tag is declared but not used, a reviewer does not know if the tag is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused tag declarations: tag *tag_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Tag Defined in Function but Not Used

```
void unusedTag ( void )
{
    enum state1 { S_init, S_run, S_sleep };    /* Non-compliant  */
}

void usedTag ( void )
```

```
{
    enum state2 { S_init, S_run, S_sleep };    /* Compliant  */
    enum state2 my_State = S_init;
}
```

In this example, in the function `unusedTag`, the tag `state1` is defined but not used. Therefore, the rule is violated.

Tag Used in `typedef` Only

```
typedef struct record_t        /* Non-compliant  */
{
    unsigned short key;
    unsigned short val;
} record1_t;


typedef struct               /* Compliant */
{
    unsigned short key;
    unsigned short val;
} record2_t;

record1_t myRecord1_t;
record2_t myRecord2_t;
```

In this example, the tag `record_t` appears only in the `typedef` of `record1_t`. In the rest of the translation unit, the type `record1_t` is used. Therefore, the rule is violated.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 2.3`

Introduced in R2014b

MISRA C:2012 Rule 2.5

A project should not contain unused macro declarations

Description

Rule Definition

A project should not contain unused macro declarations.

Rationale

If a macro is declared but not used, a reviewer does not know if the macro is redundant or if it is unused by mistake.

Message in Report

A project should not contain unused macro declarations: macro *macro_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Macro Definition

```
void use_macro (void)
{
    #define SIZE 4
    #define DATA 3

    use_int16(SIZE);
}
```

In this example, the macro `DATA` is never used in the `use_macro` function.

Check Information

Group: Unused Code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 2.6

A function should not contain unused label declarations

Description

Rule Definition

A function should not contain unused label declarations.

Rationale

If you declare a label but do not use it, it is not clear to a reviewer of your code if the label is redundant or unused by mistake.

Message in Report

A function should not contain unused label declarations.

Label *label_name* is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Label Declarations

```
void use_var(signed short);

void unused_label ( void )
{
```

```
    signed short x = 6;

label1:                        /* Non-compliant - label1 not used */
    use_var ( x );
}

void used_label ( void )
{
    signed short x = 6;

    for (int i=0; i < 5; i++) {
        if ( i==2 ) goto label1;
    }

label1:                       /* Compliant - label1 used */
    use_var ( x );
}
```

In this example, the rule is violated when the label `label1` in function `unused_label` is not used.

Check Information

Group: Unused code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 2.7

There should be no unused parameters in functions

Description

Rule Definition

There should be no unused parameters in functions.

Rationale

If a parameter is unused, it is possible that the implementation of the function does not match its specifications. This rule can highlight such mismatches.

Message in Report

There should be no unused parameters in functions.

Parameter `parameter_name` is not used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Function Parameters

```
double func(int param1, int* param2) {
    return (param1/2.0);
}
```

In this example, the rule is violated because the parameter `param2` is not used.

Check Information

Group: Unused code
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 20.1

#include directives should only be preceded by preprocessor directives or comments

Description

Rule Definition

#include directives should only be preceded by preprocessor directives or comments.

Rationale

For better code readability, group all `#include` directives in a file at the top of the file. Undefined behavior can occur if you use `#include` to include a standard header file within a declaration or definition, or if you use part of the Standard Library before including the related standard header files.

Polyspace Implementation

Polyspace flags text that precedes a `#include` directive. Polyspace ignores preprocessor directives, comments, spaces, or "new lines".

Message in Report

#include directives should only be preceded by preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives

Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 20.10

The # and ## preprocessor operators should not be used

Description

Rule Definition

The # and ## preprocessor operators should not be used.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators is unspecified. In some cases, it is therefore not possible to predict the result of macro expansion.

The use of ## can result in obscured code.

Message in Report

The # and ## preprocessor operators should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

MISRA C:2012 Rule 1.3 | MISRA C:2012 Rule 20.11

MISRA C:2012 Rule 20.11

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator

Description

Rule Definition

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

Rationale

The order of evaluation associated with multiple #, multiple ##, or a mix of # and ## preprocessor operators, is unspecified. Rule 20.10 discourages the use of # and ##. The result of a # operator is a string literal. It is extremely unlikely that pasting this result to any other preprocessing token results in a valid token.

Message in Report

The ## preprocessor operator shall not follow a macro parameter following a # preprocessor operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of # and

```
#define A( x )    #x             /* Compliant */
#define B( x, y ) x ## y         /* Compliant */
#define C( x, y ) #x ## y     /* Non-compliant */
```

In this example, you can see three uses of the # and ## operators. You can use these preprocessing operators alone (line 1 and line 2), but using # then ## is noncompliant (line 3).

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.10

MISRA C:2012 Rule 20.12

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators

Description

Rule Definition

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

Rationale

The parameter to # or ## is not expanded prior to being used. The same parameter appearing elsewhere in the replacement text is expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement might not meet developer expectations.

Message in Report

Expanded macro parameter *`param1`* is also an operand of *`op`* operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.13

A line whose first token is # shall be a valid preprocessing directive

Description

Rule Definition

A line whose first token is # shall be a valid preprocessing directive

Rationale

You typically use a preprocessing directive to conditionally exclude source code until a corresponding `#else`, `#elif`, or `#endif` directive is encountered. If your compiler does not detect a preprocessing directive because it is malformed or invalid, you can end up excluding more code than you intended.

If all preprocessing directives are syntactically valid, even in excluded code, this unintended code exclusion cannot happen.

Message in Report

Directive is not syntactically meaningful.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.14

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

Rationale

When conditional compilation directives include or exclude blocks of code and are spread over multiple files, confusion arises. If you terminate an `#if` directive within the same file, you reduce the visual complexity of the code and the chances of an error.

If you terminate `#if` directives within the same file, you can use `#if` directives in included files

Message in Report

- '#else' not within a conditional.
- '#elseif' not within a conditional.
- '#endif' not within a conditional.

  Unterminated conditional directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.2

The ', " or \ characters and the /* or // character sequences shall not occur in a header file name

Description

Rule Definition

The ', " or \ characters and the / or // character sequences shall not occur in a header file name.*

Rationale

The program’s behavior is undefined if:

- You use ', ", \, /* or // between < > delimiters in a header name preprocessing token.
- You use ', \, /* or // between " delimiters in a header name preprocessing token.

Although \ results in undefined behavior, many implementations accept / in its place.

Polyspace Implementation

Polyspace flags the characters ', ", \, /* or // between < and > in #include <filename>.

Polyspace flags the characters ', \, /* or // between " and " in #include "filename".

Message in Report

The ', "or \ characters and the /* or // character sequences shall not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.3

The #include directive shall be followed by either a <filename> or \"filename\" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Rationale

This rule applies only after macro replacement.

The behavior is undefined if an `#include` directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

Message in Report

- ‘#include' expects \"FILENAME\" or <FILENAME>
- ‘#include_next' expects \"FILENAME\" or <FILENAME>
- ‘#include' does not expect string concatenation.
- ‘#include_next' does not expect string concatenation.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.4

A macro shall not be defined with the same name as a keyword

Description

Rule Definition

A macro shall not be defined with the same name as a keyword.

Rationale

Using macros to change the meaning of keywords can be confusing. The behavior is undefined if you include a standard header while a macro is defined with the same name as a keyword.

Message in Report

- The macro *macro_name* shall not be redefined.
- The macro *macro_name* shall not be undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Redefining `int` keyword

```
#define int some_other_type
           /* Non-compliant - int keyword behavior altered */
#include <stdlib.h>
...
```

In this example, the `#define` violates Rule 20.4 because it alters the behavior of the `int` keyword. The inclusion of the standard header results in undefined behavior.

Correction — Rename keyword

One possible correction is to use a different keyword:

```
#define int_mine some_other_type
#include <stdlib.h>
...
```

Redefining keywords versus statements

```
#define while(E) for ( ; (E) ; )  /* Non-compliant - while redefined*/
#define unless(E) if ( !(E) )     /* Compliant*/

#define seq(S1, S2) do{ S1; S2;} while(false)  /* Compliant*/
#define compound(S) {S;}                       /* Compliant*/
...
```

In this example, it is noncompliant to redefine the keyword `while`, but it is compliant to define a macro that expands to statements.

Redefining keywords in different standards

```
#define inline
```

In this example, redefining `inline` is compliant in C90, but not in C99 because `inline` is not a keyword in C90.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Languages: C90, C99

See Also

MISRA C:2012 Rule 20.5

#undef should not be used

Description

Rule Definition

#undef should not be used.

Rationale

`#undef` can make the software unclear which macros exist at a particular point within a translation unit.

Message in Report

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

MISRA C:2012 Rule 20.6

Tokens that look like a preprocessing directive shall not occur within a macro argument

Description

Rule Definition

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Rationale

An argument containing sequences of tokens that otherwise act as preprocessing directives leads to undefined behavior.

Polyspace Implementation

Polyspace looks for the # character in a macro arguments (outside a string or character constant).

Message in Report

Macro argument shall not look like a preprocessing directive.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Macro Expansion Causing Non-Compliance

```
#define M( A ) printf ( #A )
```

```
#include <stdio.h>

void foo(void){
    M(
#ifdef SW         /* Non-compliant */
    "Message 1"
#else
    "Message 2"   /* Compliant - SW not defined */
#endif            /* Non-compliant */
    );
}
```

This example shows a macro definition and the macro usage. `#ifdef SW` and `#endif` are noncompliant because they look like a preprocessing directive. Polyspace does not flag `#else "Message 2"` because after macro expansion, Polyspace knows `SW` is not defined. The expanded macro is `printf ("\"Message 2\"");`

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 20.7

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

Description

Rule Definition

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

Rationale

If you do not use parentheses, then it is possible that operator precedence does not give the results that you want when macro substitution occurs.

If you are not using a macro parameter as an expression, then the parentheses are not necessary because no operators are involved in the macro.

Message in Report

Expanded macro parameter *`param`* shall be enclosed in parentheses.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Macro Expressions

```
#define mac1(x, y) (x * y)
#define mac2(x, y) ((x) * (y))
```

```
void foo(void){
    int r;

    r = mac1(1 + 2, 3 + 4);       /* Non-compliant */
    r = mac1((1 + 2), (3 + 4));   /* Compliant */

    r = mac2(1 + 2, 3 + 4);       /* Compliant */
}
```

In this example, `mac1` and `mac2` are two defined macro expressions. The definition of `mac1` does not enclose the arguments in parentheses. In line 7, the macro expands to `r = (1 + 2 * 3 + 4);` This expression can be `(1 + (2 * 3) + 4)` or `(1 + 2) * (3 + 4)`. However, without parentheses, the program does not know the intended expression. Line 8 uses parentheses, so the line expands to `(1 + 2) * (3 + 4)`. This macro expression is compliant.

The definition of `mac2` does enclose the argument in parentheses. Line 10 (the same macro arguments in line 7) expands to `(1 + 2) * (3 + 4)`. This macro and macro expression are compliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.9`

MISRA C:2012 Rule 20.8

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1

Description

Rule Definition

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Rationale

Strong typing requires that conditional inclusion preprocessing directives, `#if` or `#elif`, have a controlling expression that evaluates to a Boolean value.

Message in Report

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 14.4`

MISRA C:2012 Rule 20.9

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation

Description

Rule Definition

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.

Rationale

If attempt to use a macro identifier in a preprocessing directive, and you have not defined that identifier, then the preprocessor assumes that it has a value of zero. This value might not meet developer expectations.

Message in Report

`Identifier` is not defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Macro Identifiers

```
#if M == 0                    /* Non-compliant - Not defined */
#endif
```

```
#if defined (M)              /* Compliant - M is not evaluate */
#if M == 0                   /* Compliant - M is known to be defined */
#endif
#endif

#if defined (M) && (M == 0)  /* Compliant
                              * if M defined, M evaluated in ( M == 0 ) */
#endif
```

This example shows various uses of `M` in preprocessing directives. The second and third `#if` clauses check to see if the software defines `M` before evaluating `M`. The first `#if` clause does not check to see if `M` is defined, and because `M` is not defined, the statement is noncompliant.

Check Information

Group: Preprocessing Directives
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.9`

MISRA C:2012 Rule 21.1

#define and #undef shall not be used on a reserved identifier or reserved macro name

Description

Rule Definition

#define and #undef shall not be used on a reserved identifier or reserved macro name.

Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro can result in undefined behavior. This rule applies to the following:

- Identifiers or macro names beginning with an underscore
- Identifiers in file scope described in the C Standard Library (ISO/IEC 9899:1999, Section 7, "Library")
- Macro names described in the C Standard Library as being defined in a standard header (ISO/IEC 9899:1999, Section 7, "Library").

Message in Report

- The macro *`macro_name`* shall not be redefined.
- The macro *`macro_name`* shall not be undefined.
- The macro *`macro_name`* shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Defining or Undefining Reserved Identifiers

```
#undef __LINE__              /* Non-compliant - begins with _ */
#define _Guard_H 1           /* Non-compliant - begins with _ */
#undef _ BUILTIN_sqrt       /* Non-compliant - implementation may
                              * use _BUILTIN_sqrt for other purposes,
                              * e.g. generating a sqrt instruction */
#define defined              /* Non-compliant - reserved identifier */
#define errno my_errno       /* Non-compliant - library identifier */
#define isneg(x) ( (x) < 0 )  /* Compliant - rule doesn't include
                              * future library directions   */
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Languages: C90, C99

See Also

`MISRA C:2012 Rule 20.4`

Introduced in R2014b

MISRA C:2012 Rule 21.10

The Standard Library time and date functions shall not be used

Description

Rule Definition

The Standard Library time and date functions shall not be used.

Rationale

Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required

AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.11

The standard header file <tgmath.h> shall not be used

Description

Rule Definition

The standard header file <tgmath.h> shall not be used.

Rationale

Using the facilities of this header file can cause undefined behavior.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Function in tgmath.h

```
#include <tgmath.h>

float f1,res;


void func(void) {
    res = sqrt(f1); /* Non-compliant */
}
```

In this example, the rule is violated when the `sqrt` macro defined in `tgmath.h` is used.

Correction — Use Appropriate Function in math.h

For this example, one possible correction is to use the function `sqrtf` defined in `math.h` for `float` arguments.

```
#include <math.h>

float f1, res;


void func(void) {
 res = sqrtf(f1);
}
```

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.12

The exception handling features of `<fenv.h>` should not be used

Description

Rule Definition

The exception handling features of `<fenv.h>` should not be used.

Rationale

In some cases, the values of the floating-point status flags are unspecified. Attempts to access them can cause undefined behavior.

Message in Report

The exception handling features of `<fenv.h>` should not be used

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Features in `<fenv.h>`

```
#include <fenv.h>

void func(float x, float y) {
    float z;

    feclearexcept(FE_DIVBYZERO);              /* Non-compliant */
```

```
    z = x/y;

    if(fetestexcept (FE_DIVBYZERO))  {        /* Non-compliant */
    }
    else {
#pragma STDC FENV_ACCESS ON
        z=x*y;
        if(z>x) {
#pragma STDC FENV_ACCESS OFF
            if(fetestexcept (FE_OVERFLOW)) {  /* Non-compliant */
            }
        }
    }
}
```

In this example, the rule is violated when the identifiers `feclearexcept` and `fetestexcept`, and the macros `FE_DIVBYZERO` and `FE_OVERFLOW` are used.

Check Information

Group: Standard libraries
Category: Advisory
AGC Category: Advisory
Language: C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 21.13

Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value EOF

Description

Rule Definition

Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value EOF.

Rationale

Functions in `<ctype.h>` have a well-defined behavior only for `int` arguments whose value is within the range of `unsigned char` or the negative value equivalent of `EOF`. The use of other values results in undefined behavior.

Polyspace Implementation

Polyspace considers that the negative value equivalent of EOF is -1 and does not raise a violation if you pass -1 as argument to a function in `ctype.h`.

Message in Report

Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value `EOF`.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Invalid Arguments for Functions from <ctype.h>

```
bool_t f (uint8_t a)
{
    return (         isdigit ((int32_t) a  )     /* Compliant     */
                &&  isalpha ((int32_t) 'b')     /* Compliant     */
                &&  islower (          EOF)     /* Compliant     */
                &&  isalpha (          256));   /* Non-compliant */
}
```

In this example, the rule is violated when `256`, which is an neither an `unsigned char` or the value `EOF`, is passed as an input argument to the `isalpha` function.

Note The `int` casts in the above example are required to comply with Rule 10.3 on page 2-17.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 10.3`

Introduced in R2017a

MISRA C:2012 Rule 21.14

The Standard Library function `memcmp` shall not be used to compare null terminated strings

Description

Rule Definition

The Standard Library function `memcmp` shall not be used to compare null terminated strings.

Rationale

If `memcmp` is used to compare two strings and the length of either string is less than the number of bytes compared, the strings can appear different even when they are logically the same. The characters after the null terminator are compared even though they do not form part of the string.

For instance:

```
memcmp(string1, string2, sizeof(string1))
```

can compare bytes after the null terminator if `string1` is longer than `string2`.

Message in Report

The Standard Library function `memcmp` shall not be used to compare null terminated strings.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Using memcmp for String Comparison

```
extern char buffer1[ 12 ];
extern char buffer2[ 12 ];
void f1 ( void )
{
    ( void ) strcpy ( buffer1, "abc" );
    ( void ) strcpy ( buffer2, "abc" );
    if ( memcmp ( buffer1, buffer2, sizeof ( buffer1 ) ) != 0 )
    {
        /* Non-compliant */
    }
}
```

In this example, the comparison in the `if` statement is noncompliant. The strings stored in `buffer1` and `buffer2` can be reported different, but this difference comes from uninitialized characters after the null terminators.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.15` | `MISRA C:2012 Rule 21.16`

Introduced in R2017a

MISRA C:2012 Rule 21.15

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

Description

Rule Definition

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

Rationale

The functions

```
memcpy( arg1, arg2, num_bytes );
memmove( arg1, arg2, num_bytes );
memcmp( arg1, arg2, num_bytes );
```

perform a byte-by-byte copy, move or comparison between the memory locations that `arg1` and `arg2` point to. A byte-by-byte copy, move or comparison is meaningful only if `arg1` and `arg2` have compatible types.

Using pointers to different data types for `arg1` and `arg2` typically indicates a coding error.

Message in Report

The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incompatible Argument Types for memcpy

```
void f ( uint8_t s1[ 8 ], uint16_t s2[ 8 ] )
{
    ( void ) memcpy ( s1, s2, 8 ); /* Non-compliant */
}
```

In this example, `s1` and `s2` are pointers to different data types. The `memcpy` statement copies eight bytes from one buffer to another.

Eight bytes represent the entire span of the buffer that `s1` points to, but only part of the buffer that `s2` points to. Therefore, the `memcpy` statement copies only part of `s2` to `s1`, which might be unintended.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.16`

Introduced in R2017a

MISRA C:2012 Rule 21.16

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

Description

Rule Definition

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

Rationale

The Standard Library function

```
memcmp ( lhs, rhs, num );
```

performs a byte-by-byte comparison of the first `num` bytes of the two objects that `lhs` and `rhs` point to.

Do not use `memcmp` for a byte-by-byte comparison of the following.

Type	Rationale
Structures	If members of a structure have different data types, your compiler introduces additional padding for data alignment in memory. The content of these extra padding bytes is meaningless. If you perform a byte-by-byte comparison of structures with `memcmp`, you compare even the meaningless data stored in the padding. You might reach the false conclusion that two data structures are not equal, even if their corresponding members have the same value.

Type	Rationale
Objects with essentially floating type	The same floating point value can be stored using different representations. If you perform a byte-by-byte comparison of two variables with `memcmp`, you can reach the false conclusion that the variables are unequal even when they have the same value. The reason is that the values are stored using two different representations.
Essentially char arrays	Essentially char arrays are typically used to store strings. In strings, the content in bytes after the null terminator is meaningless. If you perform a byte-by-byte comparison of two strings with `memcmp`, you might reach the false conclusion that two strings are not equal, even if the bytes before the null terminator store the same value.

Message in Report

The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an *essentially signed* type, an *essentially unsigned* type, an *essentially Boolean* type or an *essentially enum* type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Using `memcmp` for Comparison of Structures, Unions, and *essentially char* Arrays

```
struct S;
bool_t f1 ( struct S *s1, struct S *s2 )
{
        return ( memcmp ( s1, s2, sizeof ( struct S ) ) != 0 ); /* Non-compliant */
}

union U
{
uint32_t range;
```

```
uint32_t height;
};
bool_t f2 ( union U *u1, union U *u2 )
{
        return ( memcmp ( u1, u2, sizeof ( union U ) ) != 0 ); /* Non-compliant */
}

const char a[ 6 ] = "task";
bool_t f3 ( const char b[ 6 ] )
{
        return ( memcmp ( a, b, 6 ) != 0 ); /* Non-compliant */
}
```

In this example:

- Structures `s1` and `s2` are compared in the `bool_t f1` function. The return value of this function might indicate that `s1` and `s2` are different due to padding. This comparison is noncompliant.
- Unions `u1` and `u2` are compared in the `bool_t f2` function. The return value of this function might indicate that `u1` and `u2` are the same due to unintentional comparison of `u1.range` and `u2.height`, or `u1.height` and `u2.range`. This comparison is noncompliant.
- Essentially char arrays `a` and `b` are compared in the `bool_t f3` function. The return value of this function might incorrectly indicate that the strings are different because the length of `a` (four) is less than the number of bytes compared (six). This comparison is noncompliant.

Check Information

Group: Standard libraries
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.15`

Introduced in R2017a

MISRA C:2012 Rule 21.17

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters

Description

Rule Definition

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

Rationale

Incorrect use of a string handling function might result in a read or write access beyond the bounds of the function arguments, resulting in undefined behavior.

Message in Report

Use of the string handling function from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Pointer Access Out of Bounds from `strcpy` Usage

```
char string[] = "Short";
void f1 ( const char *str )
{
```

```
        ( void ) strcpy ( string, "Too long to fit" );              /* Non-compliant */
        if ( strlen ( str ) < ( sizeof ( string ) - 1u ) )
        {
            ( void ) strcpy ( string, str );                         /* Compliant */
        }
}

size_t f2 ( void )
{
        char text[ 5 ] = "Token";
        return strlen ( text );                                     /* Non-compliant */
}
```

In this example:

- The first use of `strcpy` is noncompliant because it attempts to write beyond the end of its destination argument `string`.
- The second use of `strcpy` is compliant because it attempts to write to the destination argument `string` only if the source argument `str` fits.
- The use of `strlen` is noncompliant. `strlen` computes the length of a string up to the null terminator. The character array `text` has no null terminator.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.18`

Introduced in R2017a

MISRA C:2012 Rule 21.18

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value

Description

Rule Definition

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

Rationale

The value must be positive and not greater than the size of the smallest object passed by pointer to the function. For instance, suppose you use the `strncmp` function to compare two strings `lhs_string` and `rhs_string` as follows:

```
strncmp (lhs_string, rhs_string, num)
```

The third argument `num` must be positive and must not be greater than the size of `lhs_string` or `rhs_string`, whichever is smaller.

Otherwise, using the function can result in read or write access beyond the bounds of the function argument.

Message in Report

The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect `size_t` Argument for `memcmp`

```
char buf1[ 5 ] = "12345";
char buf2[ 10 ] = "1234567890";

void f ( void )
{
        if ( memcmp ( buf1, buf2, 5 ) == 0 )
        {
            /* Compliant */
        }
        if ( memcmp ( buf1, buf2, 6 ) == 0 )
        {
            /* Non-compliant */
        }
}
```

In this example, the first `if` statement is compliant. The `size_t` argument is five, which is same as the size of the smaller string, `buf1`.

By the same reasoning, the second `if` statement is noncompliant.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 21.17`

Introduced in R2017a

MISRA C:2012 Rule 21.19

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type

Description

Rule Definition

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type.

Rationale

The C99 Standard states that if the program modifies the structure pointed to by the value returned by `localeconv`, or the strings returned by `getenv`, `setlocale` or `strerro`, undefined behavior occurs. Treating the pointers returned by the various functions as if they were `const`-qualified allows an analysis tool to detect any attempt to modify an object through one of the pointers. Assigning the return values of the functions to `const`-qualified pointers results in the compiler issuing a diagnostic if an attempt is made to modify an object.

Message in Report

The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to `const`-qualified type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Returning Pointers fromsetlocale and localeconv

```
void f1 ( void )
{
        char *s1 = setlocale ( LC_ALL, 0 ); /* Non-compliant */
        struct lconv *conv = localeconv (); /* Non-compliant */
        s1[ 1 ] = 'A'; /* Undefined behavior */
        conv->decimal_point = "^"; /* Undefined behavior */
}

void f2 ( void )
{
        char str[128];
        (void) strcpy (str, setlocale ( LC_ALL,0 ) ); /* Compliant */
        const struct lconv *conv = localeconv ();     /* Compliant */
        conv->decimanl_point = "^"                    /* Constraint violation */
}

void f3 ( void )
{
const struct lconv *conv = localeconv (); /* Compliant */
conv->grouping[ 2 ] = 'x';                /* Non-compliant */
}
```

In the above example:

- The usage of `setlocale` and `localeconv` in the function `f1` are non-compliant as the returned pointers are assigned to non-`const`—qualified pointers.

  Note The usage of `setlocale` and `localeconv` above are not constraint violations and will therefore not be reported by a compiler. However, an analysis tool will be able to report a violation.

- The usage of `setlocale` in the function `f2` is compliant as `strcpy` takes a `const char *` as its second parameter. The usage of `localeconv` in the function `f2` is compliant as the returned pointers are assigned to a `const`-qualified pointer. Any attempt to modify an object through a pointer will be reported by a compiler or analysis tool as this is a constraint violation.

- The usage of a `const`-qualified pointer in the function `f3` gives compile time protection of the value returned by `localeconv` but the same is not true for the strings it references. Modification of these strings can be detected by an analysis tool.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 7.4` | `MISRA C:2012 Rule 11.8` | `MISRA C:2012 Rule 21.8`

Introduced in R2017a

MISRA C:2012 Rule 21.2

A reserved identifier or macro name shall not be declared

Description

Rule Definition

A reserved identifier or macro name shall not be declared.

Rationale

The Standard allows implementations to treat reserved identifiers specially. If you reuse reserved identifiers, you can cause undefined behavior.

Polyspace Implementation

- If you define a macro name that corresponds to a standard library macro, object, or function, rule 21.1 is violated.
- The rule considers tentative definitions as definitions.

Message in Report

Identifier 'XX' shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required

AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.20

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function

Description

Rule Definition

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

Rationale

The preceding functions return a pointer to an object within the Standard Library. Implementation for this object can use a static buffer that can be modified by a second call to the same function. Therefore the value accessed through a pointer before a subsequent call to the same function can change unexpectedly.

Message in Report

The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Return Value from getenv After Another Call to getenv

```
void f1( void )
{
        const char *res1;
        const char *res2;
        char copy[ 128 ];
        res1 = setlocale ( LC_ALL, 0 );
        ( void ) strcpy ( copy, res1 );
        res2 = setlocale ( LC_MONETARY, "French" );
        printf ( "%s\n", res1 ); /* Non-compliant */
        printf ( "%s\n", copy ); /* Compliant */
        printf ( "%s\n", res2 ); /* Compliant */
}
```

In this example:

- The first `printf` statement is non-compliant because the pointer returned by `setlocale` is used following a subsequent call to it when `res2` is assigned.
- The second `printf` statement is compliant because the copy operation performed by `strcpy` is made before a subsequent call to `setlocale` function is made.
- The third `printf` statement is compliant because there is no subsequent call to the `setlocale` function is made before use.

Check Information

Group: Standard libraries
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2017a

MISRA C:2012 Rule 21.3

The memory allocation and deallocation functions of <stdlib.h> shall not be used

Description

Rule Definition

The memory allocation and deallocation functions of <stdlib.h> shall not be used.

Rationale

Using memory allocation and deallocation routines can cause undefined behavior. For instance:

- You free memory that you had not allocated dynamically.
- You use a pointer that points to a freed memory location.

Polyspace Implementation

If you use names of dynamic heap memory allocation functions for macros, and you expand the macros in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro <name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of malloc, calloc, realloc and free

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */

    free(ad_1);                                    /* Non-compliant */
    free(ad_2);                                    /* Non-compliant */
    free(ad_3);                                    /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 18.7`

Introduced in R2014b

MISRA C:2012 Rule 21.4

The standard header file <setjmp.h> shall not be used

Description

Rule Definition

The standard header file <setjmp.h> shall not be used.

Rationale

Using `setjmp` and `longjmp`, you can bypass normal function call mechanisms and cause undefined behavior.

Polyspace Implementation

If the `longjmp` function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required

AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.5

The standard header file <signal.h> shall not be used

Description

Rule Definition

The standard header file <signal.h> shall not be used.

Rationale

Using signal handling functions can cause implementation-defined and undefined behavior.

Polyspace Implementation

If the signal function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required

AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.6

The Standard Library input/output functions shall not be used

Description

Rule Definition

The Standard Library input/output functions shall not be used.

Rationale

This rule applies to the functions that are provided by <stdio.h> and in C99, their character-wide equivalents provided by <wchar.h>. Using these functions can cause unspecified, undefined and implementation-defined behavior.

Polyspace Implementation

If the Standard Library function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries

Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.7

The `atof`, `atoi`, `atol`, and `atoll` functions of `<stdlib.h>` shall not be used

Description

Rule Definition

The atof, atoi, atol, and atoll functions of <stdlib.h> shall not be used.

Rationale

When a string cannot be converted, the behavior of these functions can be undefined.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required
AGC Category: Required

Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.8

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system of <stdlib.h> shall not be used.

Rationale

Using these functions can cause undefined and implementation-defined behaviors.

Polyspace Implementation

In case the abort, exit, getenv, and system functions are actually macros, and the macros are expanded in the code, this rule is detected as violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries
Category: Required

AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 21.9

The library functions bsearch and qsort of <stdlib.h> shall not be used

Description

Rule Definition

The library functions bsearch and qsort of <stdlib.h> shall not be used.

Rationale

The comparison function in these library functions can behave inconsistently when the elements being compared are equal. Also, the implementation of `qsort` can be recursive and place unknown demands on the call stack.

Polyspace Implementation

If the function is a macro and the macro is expanded in the code, this rule is violated. It is assumed that rule 21.2 is not violated.

Message in Report

- The macro '<name> shall not be used.
- Identifier XX should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Libraries

Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 22.1

All resources obtained dynamically by means of Standard Library functions shall be explicitly released

Description

Rule Definition

All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

Rationale

Resources are something that you must return to the system once you have used them. Examples include dynamically allocated memory and file descriptors.

If you do not release resources explicitly as soon as possible, then a failure can occur due to exhaustion of resources.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Message in Report

All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Dynamic Memory

```
#include<stdlib.h>


void performOperation(int);

int func1(int num) {
    int *arr1 = (int*) malloc(num * sizeof(int));

    return 0;
}            /* Non-compliant - memory allocated to arr1 is not released */

int func2(int num) {
    int *arr2 = (int*) malloc(num * sizeof(int));

    free(arr2);
    return 0;
}           /* Compliant - memory allocated to arr2 is released */
```

In this example, the rule is violated when memory dynamically allocated using the `malloc` function is not freed using the `free` function before the end of scope.

File Pointers

```
#include <stdio.h>
void func1( void ) {
    FILE *fp1;
    fp1 = fopen ( "data1.txt", "w" );
    fprintf ( fp1, "*" );

    fp1 = fopen ( "data2.txt", "w" );              /* Non-compliant */
    fprintf ( fp1, "!" );
    fclose ( fp1 );
}

void func2( void ) {
    FILE *fp2;
    fp2 = fopen ( "data1.txt", "w" );
```

```
    fprintf ( fp2, "*" );
    fclose(fp2);

    fp2 = fopen ( "data2.txt", "w" );              /* Compliant */
    fprintf ( fp2, "!" );
    fclose ( fp2 );
}
```

In this example, the file pointer `fp1` is pointing to a file `data1.txt`. Before `fp1` is explicitly dissociated from the file stream of `data1.txt`, it is used to access another file `data2.txt`. Therefore, the rule 22.1 is violated.

The rule is not violated in `func2` because file `data1.txt` is closed and the file pointer `fp2` is explicitly dissociated from `data1.txt` before it is reused.

Check Information

Group: Resources
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.10

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function

Description

Rule Definition

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function.

Rationale

Besides the `errno`-setting functions, the Standard does not enforce that other functions set `errno` on errors. Whether these functions set `errno` or not is implementation-dependent.

To detect errors, if you check `errno` alone, the validity of this check also becomes implementation-dependent. On implementations that do not require `errno` setting, even if you check `errno` alone, you can overlook error conditions.

For a list of `errno`-setting functions, see `MISRA C:2012 Rule 22.8`.

Message in Report

The value of `errno` shall only be tested when the last function to be called was an `errno`-setting function.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect Test of `errno`

```
void f ( void )
{
    float64_t f64;
    errno = 0;
    f64 = atof ( "A.12" );
    if ( 0 == errno ) /* Non-compliant */
    {
    }
    errno = 0;
    f64 = strtod ( "A.12", NULL );
    if ( 0 == errno ) /* Compliant */
    {
    }
}
```

In this example:

- The first `if` statement is noncompliant because `atof` may or may not set `errno` when an error is detected. `f64` may not have a valid value within this `if` statement.
- The second `if` statement is compliant because `strtod` is an *`errno`-setting function*. `f64` will have a valid value within this `if` statement.

Check Information

Group: Resources
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 22.8` | `MISRA C:2012 Rule 22.9`

Introduced in R2017a

MISRA C:2012 Rule 22.2

A block of memory shall only be freed if it was allocated by means of a Standard Library function

Description

Rule Definition

A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Rationale

The Standard Library functions that allocate memory are `malloc`, `calloc` and `realloc`.

You free a block of memory when you pass its address to the `free` or `realloc` function. The following causes undefined behavior:

- You free a block of memory that you did not allocate.
- You free a block of memory that have already freed before.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Message in Report

A block of memory shall only be freed if it was allocated by means of a Standard Library function.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Memory Not Allocated Is Freed

```
#include <stdlib.h>

void func1(void) {
    int x=0;
    int *ptr=&x;

    free(ptr);
   /* Non-compliant: ptr is not dynamically allocated */
}
```

In this example, the rule is violated because the `free` function operates on a pointer that does not point to dynamically allocated memory.

Memory Freed Twice

```
#include <stdlib.h>

void func(int arrSize) {
    int *ptr = (int*) malloc(arrSize* sizeof(int));

    free(ptr);   /* Block of memory freed once */
    free(ptr);   /* Non-compliant - Block of memory freed twice */
}
```

In this example, the rule is violated when the `free` function operates on `ptr` twice without a reallocation in between.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.3

The same file shall not be open for read and write access at the same time on different streams

Description

Rule Definition

The same file shall not be open for read and write access at the same time on different streams.

Rationale

If a file is both written and read via different streams, the behavior can be undefined.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Message in Report

The same file shall not be open for read and write access at the same time on different streams.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Opening File That Is Open in Another Stream

```
#include <stdio.h>

void func(void) {
    FILE *fw = fopen("tmp.txt", "r+");
    FILE *fr = fopen("tmp.txt", "r");   /* Non-compliant: File open in stream fw*/
}
```

In this example, the rule is violated when the same file `tmp.txt` is opened in two streams. The `FILE` pointers `fw` and `fr` point to two different streams here.

Check Information

Group: Resources
Category: Required
AGC Category: Required
Language: C

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.4

There shall be no attempt to write to a stream which has been opened as read-only

Description

Rule Definition

There shall be no attempt to write to a stream which has been opened as read-only.

Rationale

The Standard does not specify the behavior if an attempt is made to write to a read-only stream.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Message in Report

There shall be no attempt to write to a stream which has been opened as read-only.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Writing to File Opened as Read-Only

```
#include <stdio.h>
```

```
void func1(void) {
    FILE *fp1 = fopen("tmp.txt", "r");
    (void) fprintf(fp1, "Some text"); /* Non-compliant: Read-only stream */
    (void) fclose(fp1);
}

void func2(void) {
    FILE *fp2 = fopen("tmp.txt", "r+");
    (void) fprintf(fp2, "Some text"); /* Compliant */
    (void) fclose(fp2);
}
```

In this example, the file stream associated with `fp1` is opened as read-only. The rule is violated when the stream is written.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.5

A pointer to a `FILE` object shall not be dereferenced

Description

Rule Definition

A pointer to a `FILE` object shall not be dereferenced.

Rationale

The Standard states that the address of a `FILE` object used to control a stream can be significant. Copying that object might not give the same behavior. This rule ensures that you cannot perform such a copy.

Directly manipulating a `FILE` object might be incompatible with its use as a stream designator.

Message in Report

A pointer to a `FILE` object shall not be dereferenced

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

FILE* Pointer Dereferenced

```
#include <stdio.h>
```

```
void func(void) {
    FILE *pf1;
    FILE *pf2;
    FILE f3;

    pf2 = pf1;        /* Compliant */
    f3 = *pf2;        /* Non-compliant */
    pf2->_flags=0;    /* Non-compliant */
 }
```

In this example, the rule is violated when the `FILE*` pointer `pf2` is dereferenced.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.6

The value of a pointer to a `FILE` shall not be used after the associated stream has been closed

Description

Rule Definition

The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.

Rationale

The Standard states that the value of a `FILE*` pointer is indeterminate after you close the stream associated with it.

Polyspace Implementation

You can check for this rule with a Bug Finder analysis only.

Message in Report

The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of FILE Pointer After Closing Stream

```
#include <stdio.h>

void func(void) {
    FILE *fp;
    void *ptr;

    fp = fopen("tmp","w");
    if(fp != NULL) {
        fclose(fp);
        fprintf(fp,"text");
    }
}
```

In this example, the stream associated with the `FILE*` pointer `fp` is closed with the `fclose` function. The rule is violated `FILE*` pointer `fp` is used before the stream is re-opened.

Check Information

Group: Resources
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Rule 22.7

The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF

Description

Rule Definition

The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.

Rationale

The `EOF` value may become indistinguishable from a valid character code if the value returned is converted to another type. In such cases, testing the converted value against `EOF` will not reliably identify if the end of the file has been reached or if an error has occurred.

Message in Report

The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Possibly Misleading Results from Comparison with EOF

```
void f1 ( void )
{
```

```
    char ch;
    ch = ( char ) getchar ();
    if ( EOF != ( int32_t ) ch )  /* Non-compliant */
    {
    }
}

void f2 ( void )
{
    char ch;
    ch = ( char ) getchar ();
    if ( !feof ( stdin ) )        /* Compliant */
    {
    }
}

void f3 ( void )
{
    int32_t i_ch;
    i_ch = getchar ();
    if ( EOF != i_ch )            /* Compliant */
     {
        char ch;
        ch = ( char ) i_ch;
    }
}
```

In this example:

- The test in the `f1` function is non-compliant. It will not be reliable as the return value is cast to a narrower type before checking for `EOF`.
- The test in the `f2` function is compliant. It shows how *`feof()`* can be used to check for `EOF` when the return value from *`getchar()`* has been subjected to type conversion.
- The test in the `f3` function is compliant. It is reliable as the unconverted return value is used when checking for `EOF`.

Check Information

Group: Resources
Category: Required
AGC Category: Required

Language: C90, C99

See Also

Introduced in R2017a

MISRA C:2012 Rule 22.8

The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function

Description

Rule Definition

The value of `errno` shall be set to zero prior to a call to an `errno`-setting-function.

Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

If you do not explicitly set `errno` to zero before a function call, it can contain values from a previous call. Checking `errno` for nonzero values after the function call can give the false impression that an error occurred.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

Message in Report

The value of `errno` shall be set to zero prior to a call to an *`errno`-setting-function*.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

`errno` Not Reset Before Use

```
#include <stdlib.h>
#include <errno.h>

double val = 0.0;

void f ( void )
{
    val = strtod("1.0",NULL); /* Non-compliant */
    if ( 0 == errno ) /* Check errno for nonzero values */
    {
        val = strtod("1.0",NULL); /* Compliant - case 1*/
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
    else
    {
        errno = 0;
        val = strtod("1.0",NULL); /* Compliant - case 2*/
        if ( 0 == errno ) /* Check errno for nonzero values */
        {
        }
    }
}
```

In this example, the rule is violated when `strtod` is called but `errno` is not reset prior to the call.

The rule is not violated in the following cases:

- Case 1: `errno` is compared against zero and then `strtod` is called in the `if( 0 == errno )` branch.
- Case 2: `errno` is explicitly set to zero and then `strtod` is called.

Check Information

Group: Resources

Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 22.9` | `MISRA C:2012 Rule 22.10`

Introduced in R2017a

MISRA C:2012 Rule 22.9

The value of `errno` shall be tested against zero after calling an `errno`-setting function

Description

Rule Definition

The value of `errno` shall be tested against zero after calling an `errno`-setting function.

Rationale

If an error occurs during a call to an `errno`-setting-function, the function writes a nonzero value to `errno`. Otherwise, `errno` is not modified.

When `errno` is nonzero, the function return value is not likely to be correct. Before using this return value, you must test `errno` for nonzero values.

`Errno`-setting functions include:

- `ftell`, `fgetpos`, `fgetwc` and related functions.
- `strtoimax`, `strtol` and related functions.

  The wide-character equivalents such as `wcstoimax` and `wcstol` are also covered.

Message in Report

The value of `errno` shall be tested against zero after calling an *`errno`-setting function*.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

errno Not Tested After Function Call

```
#include <stdlib.h>
#include <errno.h>

void func(void);
double val = 0.0;

void f1 ( void )
{
  errno = 0;
  val = strtod ( "1.0", NULL ); /* Non-compliant */
  func ();

  if ( 0 != errno )
    {
    }

  errno = 0;
  val = strtod ( "1.0", NULL ); /* Compliant */
  if ( 0 == errno )
    {
      func();
    }
}
```

In this example, the rule is violated when errno is not checked immediately after the first call to strtod. Instead, a second function func is called. func might use the value in the global variable val. The value can be incorrect if an error has occurred during the call to strtod.

The rule is not violated when errno is checked before operations that potentially use the return value of strtod.

Check Information

Group: Resources
Category: Required
AGC Category: Required
Language: C90, C99

See Also

MISRA C:2012 Rule 22.8 | MISRA C:2012 Rule 22.10

Introduced in R2017a

MISRA C:2012 Rule 3.1

The character sequences `/*` and `//` shall not be used within a comment

Description

Rule Definition

The character sequences `/*` and `//` shall not be used within a comment.

Rationale

These character sequences are not allowed in code comments because:

- If your code contains a `/*` or a `//` in a `/*  */` comment, it typically means that you have inadvertently commented out code.
- If your code contains a `/*` in a `//` comment, it typically means that you have inadvertently uncommented a `/*  */` comment.

Polyspace Implementation

You cannot annotate this rule in the source code.

For information on annotations, see “Annotate Code and Hide Known or Acceptable Results”.

Message in Report

The character sequence `/*` shall not appear within a comment.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

/* Used in // Comments

```
int x;
int y;
int z;

void non_compliant_comments ( void )
{
    x = y //      /* Non-compliant
        + z
        //  */
        ;
    z++;    //    Compliant with exception: // permitted within a // comment
}

void compliant_comments ( void )
{
    x = y /*      Compliant
      + z
      */
        ;
    z++;    //    Compliant with exception: // is permitted within a // comment
}
```

In this example, in the `non_compliant_comments` function, the `/*` character occurs in what appears to be a `//` comment, violating the rule. Because of the comment structure, the operation that takes place is `x = y + z;`. However, without the two `//`-s, an entirely different operation `x=y;` takes place. It is not clear which operation is intended.

Use a comment format that makes your intention clear. For instance, in the `compliant_comments` function, it is clear that the operation `x=y;` is intended.

Check Information

Group: Comments
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 3.2

Line-splicing shall not be used in `//` comments

Description

Rule Definition

Line-splicing shall not be used in // comments.

Rationale

Line-splicing occurs when the `\` character is immediately followed by a new-line character. Line splicing is used for statements that span multiple lines.

If you use line-splicing in a `//` comment, the following line can become part of the comment. In most cases, the `\` is spurious and can cause unintentional commenting out of code.

Message in Report

Line-splicing shall not be used in `//` comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Line Splicing in // Comment

```
#include <stdbool.h>
```

```
extern _Bool b;

void func ( void )
{
    unsigned short x = 0;   // Non-compliant - Line-splicing \
    if ( b )
    {
        ++b;
    }
}
```

Because of line-splicing, the statement `if ( b )` is a part of the previous `//` comment. Therefore, the statement `b++` always executes, making the `if` block redundant.

Check Information

Group: Comments
Category: Required
AGC Category: Required
Language: C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 4.1

Octal and hexadecimal escape sequences shall be terminated

Description

Rule Definition

Octal and hexadecimal escape sequences shall be terminated.

Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant `'\x1f'` consists of a single character, whereas the character constant `'\x1g'` consists of the two characters `'\x1'` and `'g'`. The manner in which multi-character constants are represented as integers is implementation-defined.

If every octal or hexadecimal escape sequence in a character constant or string literal is terminated, you reduce potential confusion.

Message in Report

Octal and hexadecimal escape sequences shall be terminated.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Compliant and Noncompliant Escape Sequences

```
const char *s1 = "\x41g";     /* Non-compliant */
const char *s2 = "\x41" "g";  /* Compliant - Terminated by end of literal */
const char *s3 = "\x41\x67";  /* Compliant - Terminated by another escape sequence*/

int c1 = '\141t';             /* Non-compliant */
int c2 = '\141\t';            /* Compliant - Terminated by another escape sequence*/
```

In this example, the rule is violated when an escape sequence is not terminated with the end of string literal or another escape sequence.

Check Information

Group: Character Sets and Lexical Conventions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 4.2

Trigraphs should not be used

Description

Rule Definition

Trigraphs should not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance, `'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

Note Digraphs (`<:  :>`, `<%  %>`, `%:`, `%:%:`) are permitted because they are tokens.

Polyspace Implementation

The Polyspace analysis converts trigraphs to the equivalent character for the . However, Polyspace also raises a MISRA violation.

The standard requires that trigraphs must be transformed *before* comments are removed during preprocessing. Therefore, Polyspace raises a violation of this rule even if a trigraph appears in code comments.

Message in Report

Trigraphs should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Character Sets and Lexical Conventions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 5.1

External identifiers shall be distinct

Description

Rule Definition

External identifiers shall be distinct.

Rationale

External identifiers are ones declared with global scope or storage class `extern`.

Polyspace considers two names as distinct if there is a difference between their first 31 characters. If the difference between two names occurs only beyond the first 31 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 6 characters. To use the C90 rules checking, use the value `c90` for the option `C standard version (-c-version)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

External `%s %s` conflicts with the external identifier XX in file YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

C90: First Six Characters of Identifiers Not Unique

```
int engine_temperature_raw;
int engine_temperature_scaled;   /* Non-compliant */
int engin2_temperature;          /* Compliant */
```

In this example, the identifier `engine_temperature_scaled` has the same first six characters as a previous identifier, `engine_temperature_raw`.

C99: First 31 Characters of Identifiers Not Unique

```
int engine_exhaust_gas_temperature_raw;
int engine_exhaust_gas_temperature_scaled; /* Non-compliant */

int eng_exhaust_gas_temp_raw;
int eng_exhaust_gas_temp_scaled;           /* Compliant */
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same first 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

C90: First Six Characters Identifiers in Different Translation Units Differ in Case Alone

```
/* file1.c */
int abc = 0;

/* file2.c */
int ABC = 0; /* Non-compliant */
```

In this example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation are different but are not distinct in their significant characters.

Check Information

Group: Identifiers

Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5`

Introduced in R2014b

MISRA C:2012 Rule 5.2

Identifiers declared in the same scope and name space shall be distinct

Description

Rule Definition

Identifiers declared in the same scope and name space shall be distinct.

Rationale

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option .

Message in Report

Identifier XX has same significant characters as identifier YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

C90: First 31 Characters of Identifiers Not Unique

```
extern int engine_exhaust_gas_temperature_raw;
static int engine_exhaust_gas_temperature_scaled;     /* Non-compliant */
```

```
extern double engine_exhaust_gas_temperature_raw;
static double engine_exhaust_gas_temperature2_scaled;  /* Compliant */

void func ( void )
{
  /* Not in the same scope */
  int engine_exhaust_gas_temperature_local;            /* Compliant */
}
```

In this example, the identifier `engine_exhaust_gas_temperature_scaled` has the same 31 characters as a previous identifier, `engine_exhaust_gas_temperature_raw`.

The rule does not apply if the two identifiers have the same 31 characters but have different scopes. For instance, `engine_exhaust_gas_temperature_local` has the same 31 characters as `engine_exhaust_gas_temperature_raw` but different scope.

C99: First 63 Characters of Identifiers Not Unique

```
extern int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw;
static int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale;
    /* Non-compliant */

extern int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__raw;
static int engine_gas_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx__scale;
    /* Compliant */

void func ( void )
{
/* Not in the same scope */
    int engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_local;
          /* Compliant */
}
```

In this example, the identifier `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_scale` has the same 63 characters as a previous identifier, `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_x_raw`.

Check Information

Group: Identifiers

Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.3` | `MISRA C:2012 Rule 5.4` | `MISRA C:2012 Rule 5.5`

Introduced in R2014b

MISRA C:2012 Rule 5.3

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If two identifiers have the same name but different scope, the identifier in the inner scope hides the identifier in the outer scope. All uses of the identifier name refers to the identifier in the inner scope. This behavior forces the developer to keep track of the scope and reduces code readability.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option .

Message in Report

Variable XX hides variable XX (FILE line LINE column COLUMN).

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Local Variable Hidden by Another Local Variable in Inner Block

```
typedef signed short int16_t;

void func( void )
{
    int16_t i;
    {
        int16_t i;                /* Non-compliant */
        i = 3;
    }
}
```

In this example, the identifier `i` defined in the inner block in `func` hides the identifier `i` with function scope.

It is not immediately clear to a reader which `i` is referred to in the statement `i=3`.

Global Variable Hidden by Function Parameter

```
typedef signed short int16_t;

struct astruct
{
    int16_t m;
};

extern void g ( struct astruct *p );
int16_t xyz = 0;

void func ( struct astruct xyz )  /* Non-compliant */
{
    g ( &xyz );
}
```

In this example, the parameter `xyz` of function `func` hides the global variable `xyz`.

It is not immediately clear to a reader which `xyz` is referred to in the statement `g (&xyz )`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.8`

Introduced in R2014b

MISRA C:2012 Rule 5.4

Macro identifiers shall be distinct

Description

Rule Definition

Macro identifiers shall be distinct.

Rationale

The names of macro identifiers must be distinct from both other macro identifiers and their parameters.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option .

Message in Report

- Macro identifiers shall be distinct. Macro XX has same significant characters as macro YY.
- Macro identifiers shall be distinct. Macro parameter XX has same significant characters as macro parameter YY in macro ZZ.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

C90: First 31 Characters of Macro Names Not Unique

```
#define engine_exhaust_gas_temperature_raw egt_r
#define engine_exhaust_gas_temperature_scaled egt_s   /* Non-compliant */

#define engine_exhaust_gas_temp_raw egt_r
#define engine_exhaust_gas_temp_scaled egt_s          /* Compliant */
```

In this example, the macro `engine_exhaust_gas_temperature_scaled egt_s` has the same first 31 characters as a previous macro `engine_exhaust_gas_temperature_scaled`.

C99: First 63 Characters of Macro Names Not Unique

```
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw_scaled egt_s
    /* Non-compliant */

/* 63 significant case-sensitive characters in macro identifiers */
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_raw egt_r
#define new_engine_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_scaled egt_s
    /* Compliant */
```

In this example, the macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___gaz_scaled` has the same first 63 characters as a previous macro `engine_xxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx___raw`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.5`

Introduced in R2014b

MISRA C:2012 Rule 5.5

Identifiers shall be distinct from macro names

Description

Rule Definition

Identifiers shall be distinct from macro names.

Rationale

The rule requires that macro names that exist only prior to processing must be different from identifier names that also exist after preprocessing. Keeping macro names and identifiers distinct help avoid confusion.

Polyspace considers two names as distinct if there is a difference between their first 63 characters. If the difference between two names occurs only beyond the first 63 characters, they can be easily mistaken for each other. The readability of the code is reduced. For C90, the difference must occur between the first 31 characters. To use the C90 rules checking, use the value `c90` for the option .

Message in Report

Identifier XX has same significant characters as macro YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Macro Names Same as Identifier Names

```
#define Sum_1(x, y) ( ( x ) + ( y ) )
short Sum_1;                    /* Non-compliant */

#define Sum_2(x, y) ( ( x ) + ( y ) )
short x = Sum_2 ( 1, 2 );       /* Compliant */
```

In this example, `Sum_1` is both the name of an identifier and a macro. `Sum_2` is used only as a macro.

C90: First 31 Characters of Macro Name Same as Identifier Name

```
#define       low_pressure_turbine_temperature_1 lp_tb_temp_1
static int low_pressure_turbine_temperature_2;      /* Non-compliant  */
```

In this example, the identifier `low_pressure_turbine_temperature_2` has the same first 31 characters as a previous macro `low_pressure_turbine_temperature_1`.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.1` | `MISRA C:2012 Rule 5.2` | `MISRA C:2012 Rule 5.4`

Introduced in R2014b

MISRA C:2012 Rule 5.6

A typedef name shall be a unique identifier

Description

Rule Definition

A typedef name shall be a unique identifier.

Rationale

Reusing a `typedef` name as another `typedef` or as the name of a function, object or `enum` constant can cause developer confusion.

Message in Report

XX conflicts with the typedef name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

`typedef` Names Reused

```
void func ( void ){
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t; /* Non-compliant */
```

```
  }
}

typedef float mass;
void func1 ( void ){
  float mass = 0.0f;            /* Non-compliant */
}
```

In this example, the `typedef` name `u8_t` is used twice. The `typedef` name `mass` is also used as an identifier name.

`typedef` Name Same as Structure Name

```
typedef struct list{          /* Compliant - exception */
  struct list *next;
  unsigned short element;
} list;

typedef struct{
  struct chain{               /* Non-compliant */
    struct chain *list2;
    unsigned short element;
  } s1;
  unsigned short length;
} chain;
```

In this example, the `typedef` name `list` is the same as the original name of the `struct` type. The rule allows this exceptional case.

However, the `typedef` name `chain` is not the same as the original name of the `struct` type. The name `chain` is associated with a different `struct` type. Therefore, it clashes with the `typedef` name.

Check Information

Group: Identifiers
Category:
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.7`

Introduced in R2014b

MISRA C:2012 Rule 5.7

A tag name shall be a unique identifier

Description

Rule Definition

A tag name shall be a unique identifier.

Rationale

Reusing a tag name can cause developer confusion.

Message in Report

XX conflicts with the tag name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.6`

Introduced in R2014b

MISRA C:2012 Rule 5.8

Identifiers that define objects or functions with external linkage shall be unique

Description

Rule Definition

Identifiers that define objects or functions with external linkage shall be unique.

Rationale

External identifiers are those declared with global scope or with storage class `extern`. Reusing an external identifier name can cause developer confusion.

Identifiers defined within a function have smaller scope. Even if names of such identifiers are not unique, they are not likely to cause confusion.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Identifiers
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 5.3`

Introduced in R2014b

MISRA C:2012 Rule 5.9

Identifiers that define objects or functions with internal linkage should be unique

Description

Rule Definition

Identifiers that define objects or functions with internal linkage should be unique.

Polyspace Implementation

This rule checker assumes that rule 5.8 is not violated.

Message in Report

- Object XX conflicts with the object name YY.
- Function XX conflicts with the function name YY.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Identifiers
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

`MISRA C:2012 Rule 8.10`

Introduced in R2014b

MISRA C:2012 Rule 6.1

Bit-fields shall only be declared with an appropriate type

Description

Rule Definition

Bit-fields shall only be declared with an appropriate type.

Rationale

Using `int` is implementation-defined because bit-fields of type `int` can be either `signed` or `unsigned`.

The use of `enum`, `short char`, or any other type of bit-field is not permitted in C90 because the behavior is undefined.

In C99, the implementation can potentially define other integer types that are permitted in bit-field declarations.

Message in Report

Bit-fields shall only be declared with an appropriate type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Types
Category: Required
AGC Category: Required

Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 6.2

Single-bit named bit fields shall not be of a signed type

Description

Rule Definition

Single-bit named bit fields shall not be of a signed type.

Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has one sign bit and no value bits. In any representation of integers, zero value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way. Its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide much detail regarding the representation of types, the same single-bit bit-field considerations apply.

Polyspace Implementation

This rule does not apply to unnamed bit fields because their values cannot be accessed.

Message in Report

Single-bit named bit fields shall not be of a signed type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Types
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 7.1

Octal constants shall not be used

Description

Rule Definition

Octal constants shall not be used.

Rationale

Octal constants are denoted by a leading zero. Developers can mistake an octal constant as a decimal constant with a redundant leading zero.

Polyspace Implementation

If you use octal constants in a macro definition, the rule checker flags the issue even if the macro is not used.

Message in Report

Octal constants shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of octal constants

```
#define CST     021
#define VALUE   010             /* Compliant - constant not used */
```

```
#if 010 == 01                    /* Non-Compliant - constant used */
#define CST 021                  /* Non-Compliant - constant not used */
#endif

extern short code[5];
static char* str2 = "abcd\0efg";  /* Compliant */

void main(void) {
    int value1 = 0;               /* Compliant */
    int value2 = 01;              /* Non-Compliant - decimal 01 */
    int value3 = 1;               /* Compliant */
    int value4 = '\109';          /* Compliant */

    code[1] = 109;                /* Compliant     - decimal 109 */
    code[2] = 100;                /* Compliant     - decimal 100 */
    code[3] = 052;                /* Non-Compliant - decimal 42 */
    code[4] = 071;                /* Non-Compliant - decimal 57 */

    if (value1 != CST) {          /* Non-Compliant - decimal 17 */
        value1 = !(value1 != 0);  /* Compliant */
    }
}
```

In this example, the rule is not violated when octal constants are used to define macros `CST` and `VALUE`. The rule is violated only when the macros are used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 7.2

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type

Description

Rule Definition

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Message in Report

A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 7.3

The lowercase character “l” shall not be used in a literal suffix

Description

Rule Definition

The lowercase character “l” shall not be used in a literal suffix.

Rationale

The lowercase character “l” can be confused with the digit “1”. Use the uppercase “L” instead.

Message in Report

The lowercase character “l” shall not be used in a literal suffix.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 7.4

A string literal shall not be assigned to an object unless the object's type is “pointer to const-qualified char”

Description

Rule Definition

A string literal shall not be assigned to an object unless the object's type is “pointer to const-qualified char”.

Rationale

This rule prevents assignments that allow modification of a string literal.

An attempt to modify a string literal can result in undefined behavior. For example, some implementations can store string literals in read-only memory. An attempt to modify the string literal can result in an exception or crash.

Message in Report

A string literal shall not be assigned to an object unless the object's type is “pointer to const-qualified char”.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect Assignment of String Literal

```
char *str1 = "AccountHolderName";
const char *str2 = "AccountHolderName";

void checkAccount1(char*);              /* Non-Compliant */
void checkAccount2(const char*);        /* Compliant */

void main() {
 checkAccount1("AccountHolderName");    /* Non-Compliant */
 checkAccount2("AccountHolderName");    /* Compliant */
}
```

In this example, the rule is not violated when string literals are assigned to `const char*` pointers, either directly or through copy of function arguments. The rule is violated only when the `const` qualifier is not used.

Check Information

Group: Literals and Constants
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 11.4` | `MISRA C:2012 Rule 11.8`

Introduced in R2014b

MISRA C:2012 Rule 8.1

Types shall be explicitly specified

Description

Rule Definition

Types shall be explicitly specified.

Rationale

In some circumstances, you can omit types from the C90 standard. In those cases, the `int` type is implicitly specified. However, the omission of an explicit type can lead to confusion. For example, in the declaration `extern void foo (char c, const k);`, the type of `k` is `const int`, but you might expect `const char`.

You might be using an implicit type in:

- Object declarations
- Parameter declarations
- Member declarations
- `typedef` declarations
- Function return types

Polyspace Implementation

The rule checker flags situations where a function parameter or return type is not explicitly specified. To enable checking of this rule, use the value `c90` for the option `C standard version (-c-version)`.

Message in Report

Types shall be explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Implicit Types

```
static foo(int a);  /* Non compliant */
static void bar(void);      /* Compliant */
```

In this example, the rule is violated because the return type of `foo` is implicit.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90

See Also

`MISRA C:2012 Rule 8.2`

Introduced in R2014b

MISRA C:2012 Rule 8.10

An inline function shall be declared with the static storage class

Description

Rule Definition

An inline function shall be declared with the static storage class.

Rationale

If you call an inline function that is declared with external linkage but not defined in the same translation unit, the function might not be inlined. You might not see the reduction in execution time that you expect from inlining.

If you want to make an inline function available to several translation units, you can still define it with the `static` specifier. In this case, place the definition in a header file. Include the header file in all the files where you want the function inlined.

Polyspace Implementation

The rule checker flags definitions that contain the `inline` specifier without an accompanying `static` specifier.

Message in Report

An inline function shall be declared with the static storage class.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Inlining Functions with External Linkage

```
inline double mult(int val);
inline double mult(int val) {   /* Non compliant */
    return val * 2.0;
}

static inline double div(int val);
static inline double div(int val) {  /* Compliant */
    return val / 2.0;
}
```

In this example, the definition of `mult` is noncompliant because it is inlined without the `static` storage specifier.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C99

See Also

`MISRA C:2012 Rule 5.9`

Introduced in R2014b

MISRA C:2012 Rule 8.11

When an array with external linkage is declared, its size should be explicitly specified

Description

Rule Definition

When an array with external linkage is declared, its size should be explicitly specified.

Rationale

Although it is possible to declare an array with an incomplete type and access its elements, it is safer to state the size of the array explicitly. If you provide size information for each declaration, a code reviewer can check multiple declarations for their consistency. With size information, a static analysis tool can perform array bounds analysis without analyzing more than one unit.

Polyspace Implementation

The rule checker flags arrays declared with the `extern` specifier if the declaration does not explicitly specify the array size.

Message in Report

Size of array *`array_name`* should be explicitly stated. When an array with external linkage is declared, its size should be explicitly specified.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Array Declarations

```
extern int32_t array1[10];    /*  Compliant  */
extern int32_t array2[];      /*  Non-compliant  */
```

In this example, two arrays are declared `array1` and `array2`. `array1` has external linkage (the `extern` keyword) and a size of 10. `array2` also has external linkage, but no specified size. `array2` is noncompliant because for arrays with external linkage, you must explicitly specify a size.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.12

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique

Description

Rule Definition

Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

Rationale

An implicitly specified enumeration constant has a value one greater than its predecessor. If the first enumeration constant is implicitly specified, then its value is 0. An explicitly specified enumeration constant has the specified value.

If implicitly and explicitly specified constants are mixed within an enumeration list, it is possible for your program to replicate values. Such replications can be unintentional and can cause unexpected behavior.

Polyspace Implementation

The rule checker flags an enumeration if it has an implicitly specified enumeration constant with the same value as another enumeration constant.

Message in Report

The constant *`constant1`* has same value as the constant *`constant2`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Replication of Value in Implicitly Specified Enum Constants

```
enum color1 {red_1, blue_1, green_1};   /* Compliant */
enum color2 {red_2 = 1, blue_2 = 2, green_2 = 3};       /* Compliant */
enum color3 {red_3 = 1, blue_3, green_3};     /* Compliant */
enum color4 {red_4, blue_4, green_4 = 1};     /* Non Compliant */
enum color5 {red_5 = 2, blue_5, green_5 = 2};     /* Compliant */
enum color6 {red_6 = 2, blue_6, green_6 = 2, yellow_6};     /* Non Compliant */
```

Compliant situations:

- `color1`: All constants are implicitly specified.
- `color2`: All constants are explicitly specified.
- `color3`: Though there is a mix of implicit and explicit specification, all constants have unique values.
- `color5`: The implicitly specified constants have unique values.

Noncompliant situations:

- `color4`: The implicitly specified constant `blue_4` has the same value as `green_4`.
- `color6`: The implicitly specified constant `blue_6` has the same value as `yellow_6`.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.13

A pointer should point to a const-qualified type whenever possible

Description

Rule Definition

A pointer should point to a const-qualified type whenever possible.

Rationale

This rule ensures that you do not inadvertently use pointers to modify objects.

Polyspace Implementation

The rule checker flags a pointer to a non-`const` function parameter if the pointer does not modify the addressed object. The assumption is that the pointer is not meant to modify the object and so must point to a `const`-qualified type.

Message in Report

A pointer should point to a const-qualified type whenever possible.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Pointer That Should Point to `const`-Qualified Types

```
#include <string.h>
```

```
typedef unsigned short uint16_t;

uint16_t ptr_ex(uint16_t *p) {      /* Non-compliant */
    return *p;
}

char last_char(char * const s){     /* Non-compliant */
    return s[strlen(s) - 1u];
}

uint16_t first(uint16_t a[5]){      /* Non-compliant */
    return a[0];
}
```

This example shows three different noncompliant pointer parameters.

- In the `ptr_ex` function, `p` does not modify an object. However, the type to which `p` points is not `const`-qualified, so it is noncompliant.
- In `last_char`, the pointer `s` is `const`-qualified but the type it points to is not. This parameter is noncompliant because `s` does not modify an object.
- The function `first` does not modify the elements of the array `a`. However, the element type is not `const`-qualified, so `a` is also noncompliant.

Correction — Use `const` Keywords

One possible correction is to add `const` qualifiers to the definitions.

```
#include <string.h>

typedef unsigned short uint16_t;

uint16_t ptr_ex(const uint16_t *p){     /* Compliant */
    return *p;
}

char last_char(const char * const s){   /* Compliant */
    return s[strlen( s ) - 1u];
}

uint16_t first(const uint16_t a[5]) {   /* Compliant */
    return a[0];
}
```

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.14

The restrict type qualifier shall not be used

Description

Rule Definition

The restrict type qualifier shall not be used.

Rationale

When you use a `restrict` qualifier carefully, it improves the efficiency of code generated by a compiler. It can also improve static analysis. However, when using the `restrict` qualifier, it is difficult to make sure that the memory areas operated on by two or more pointers do not overlap.

Polyspace Implementation

The rule checker flags all uses of the `restrict` qualifier.

Message in Report

The restrict type qualifier shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of `restrict` Qualifier

```
void f(int n, int * restrict p, int * restrict q)
{
}
```

In this example, both uses of the `restrict` qualifier are flagged.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.2

Function types shall be in prototype form with named parameters

Description

Rule Definition

Function types shall be in prototype form with named parameters.

Rationale

The rule requires that you specify names and data types for all the parameters in a declaration. The parameter names provide useful information regarding the function interface. A mismatch between a declaration and definition can indicate a programming error. For instance, you mixed up parameters when defining the function. By insisting on parameter names, the rule allows a code reviewer to detect this mismatch.

Polyspace Implementation

The rule checker shows a violation if the parameters in a function declaration or definition are missing names or data types.

Message in Report

- Too many arguments to *`function_name`*.
- Too few arguments to *`function_name`*.
- Function types shall be in prototype form with named parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function Prototype Without Named Parameters

```
extern int func(int);   /* Non compliant */
extern int func2(int n);   /* Compliant */

extern int func3();   /* Non compliant */
extern int func4(void);   /* Compliant */
```

In this example, the declarations of `func` and `func3` are noncompliant because the parameters are missing or do not have names.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 8.1` | `MISRA C:2012 Rule 8.4` | `MISRA C:2012 Rule 17.3`

Introduced in R2014b

MISRA C:2012 Rule 8.3

All declarations of an object or function shall use the same names and type qualifiers

Description

Rule Definition

All declarations of an object or function shall use the same names and type qualifiers.

Rationale

Consistently using parameter names and types across declarations of the same object or function encourages stronger typing. It is easier to check that the same function interface is used across all declarations.

Polyspace Implementation

The rule checker detects situations where parameter names or data types are different between multiple declarations or the declaration and the definition. The checker considers declarations in all translation units and flags issues that are not likely to be detected by a compiler.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Definition of function *`function_name`* incompatible with its declaration.
- Global declaration of *`function_name`* function has incompatible type with its definition.
- Global declaration of *`variable_name`* variable has incompatible type with its definition.
- All declarations of an object or function shall use the same names and type qualifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Mismatch in Parameter Names

```
extern int div (int num, int den);

int div(int den, int num) { /* Non compliant */
    return(num/den);
}
```

In this example, the rule is violated because the parameter names in the declaration and definition are switched.

Mismatch in Parameter Data Types

```
typedef unsigned short width;
typedef unsigned short height;
typedef unsigned int area;

extern area calculate(width w, height h);

area calculate(width w, width h) { /* Non compliant *
    return w*h;
}
```

In this example, the rule is violated because the second argument of the `calculate` function has data type:

- `height` in the declaration.
- `width` in the definition.

The rule is violated even though the underlying type of `height` and `width` are identical.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 8.4`

Introduced in R2014b

MISRA C:2012 Rule 8.4

A compatible declaration shall be visible when an object or function with external linkage is defined

Description

Rule Definition

A compatible declaration shall be visible when an object or function with external linkage is defined.

Rationale

If a declaration is visible when an object or function is defined, it allows the compiler to check that the declaration and the definition are compatible.

This rule with `MISRA C:2012 Rule 8.5` enforces the practice of declaring an object (or function) in a header file and including the header file in source files that define or use the object (or function).

Polyspace Implementation

The rule checker detects situations where:

- An object or function is defined without a previous declaration.
- There is a data type mismatch between the object or function declaration and definition. Such a mismatch also causes a compilation error.

Message in Report

- Global definition of *`variable_name`* variable has no previous declaration.
- Function *`function_name`* has no visible compatible prototype at definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Definition Without Previous Declaration

Header file:

```
/* file.h */
extern int var2;
void func2(void);
```

Source file:

```
/* file.c */
#include "file.h"

int var1 = 0;    /* Non compliant */
int var2 = 0;    /* Compliant */

void func1(void) {   /* Non compliant */
}

void func2(void) {   /* Compliant */
}
```

In this example, the definitions of `var1` and `func1` are noncompliant because they are not preceded by declarations.

Mismatch in Parameter Data Types

```
void func(int param1, int param2);

void func(int param1, unsigned int param2) { /* Non compliant */
}
```

In this example, the definition of `func` has a different parameter type from its declaration. The mismatch also causes a compilation error.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 8.2` | `MISRA C:2012 Rule 8.3` | `MISRA C:2012 Rule 8.5` | `MISRA C:2012 Rule 17.3`

Introduced in R2014b

MISRA C:2012 Rule 8.5

An external object or function shall be declared once in one and only one file

Description

Rule Definition

An external object or function shall be declared once in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Polyspace Implementation

The rule checker checks only explicit `extern` declarations (tentative definitions are ignored). The checker flags variables or functions declared `extern` in a non-header file.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Object *`object_name`* has external declarations in multiple files.
- Function *`function_name`* has external declarations in multiple files.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Extern Declaration in Non-Header File

Header file:

```
/* file.h */
extern int var;
extern void func1(void);   /* Compliant */
```

Source file:

```
/* file.c */
#include "file.h"

extern void func2(void);   /* Non compliant */

/* Definitions */
int var = 0;
void func1(void) {}
```

In this example, the declaration of external function `func2` is noncompliant because it occurs in a non-header file. The other external object and function declarations occur in a header file and comply with this rule.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 8.4`

Introduced in R2014b

MISRA C:2012 Rule 8.6

An identifier with external linkage shall have exactly one external definition

Description

Rule Definition

An identifier with external linkage shall have exactly one external definition.

Rationale

If you use an identifier for which multiple definitions exist in different files or no definition exists, the behavior is undefined.

Multiple definitions in different files are not permitted by this rule even if the definitions are the same.

Polyspace Implementation

The checker flags multiple definitions only if the definitions occur in different files.

The checker does not consider tentative definitions as definitions. For instance, the following code does not violate the rule:

```
int val;
int val=1;
```

The checker does not show a violation if a function is not defined at all but declared with external linkage and called in the source code.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Forbidden multiple definitions for function *function_name*.
- Forbidden multiple tentative definitions for object *object_name*.
- Global variable *variable_name* multiply defined.
- Function *function_name* multiply defined.
- Global variable has multiple tentative definitions.
- Undefined global variable *variable_name*.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Variable Multiply Defined

First source file:

```
extern int var = 1;
```

Second source file:

```
int var = 0;   /* Non compliant */
```

In this example, the global variable `var` is multiply defined. Unless explicitly specified with the `static` qualifier, the variables have external linkage.

Function Multiply Defined

Header file:

```
/* file.h */
int func(int param);
```

First source file:

```
/* file1.c */
#include "file.h"

int func(int param) {
    return param+1;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

int func(int param) {  /* Non compliant */
    return param-1;
}
```

In this example, the function `func` is multiply defined. Unless explicitly specified with the `static` qualifier, the functions have external linkage.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.7

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

Description

Rule Definition

Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

Rationale

Compliance with this rule avoids confusion between your identifier and an identical identifier in another translation unit or library. If you restrict or reduce the visibility of an object by giving it internal linkage or no linkage, you or someone else is less likely to access the object inadvertently.

Polyspace Implementation

The rule checker flags:

- Objects that are defined at file scope without the `static` specifier but used only in one file.
- Functions that are defined without the `static` specifier but called only in one file.

If you intend to use the object or function in one file only, declare it static.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

- Variable *`variable_name`* should have internal linkage.

- Function *function_name* should have internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Variable with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
```

First source file:

```
/* file1.c */
#include "file.h"

int var;    /* Compliant */
int var2;   /* Non compliant */
static int var3; /* Compliant */

void reset(void);

void reset(void) {
    var = 0;
    var2 = 0;
    var3 = 0;
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment(int var2);

void increment(int var2) {
```

```
    var++;
    var2++;
}
```

In this example:

- The declaration of `var` is compliant because `var` is declared with external linkage and used in multiple files.
- The declaration of `var2` is noncompliant because `var2` is declared with external linkage but used in one file only.

  It might appear that `var2` is defined in both files. However, in the second file, `var2` is a parameter with no linkage and is not the same as the `var2` in the first file.
- The declaration of `var3` is compliant because `var3` is declared with internal linkage (with the `static` specifier) and used in one file only.

Function with External Linkage Used in One File

Header file:

```
/* file.h */
extern int var;
extern void increment1 (void);
```

First source file:

```
/* file1.c */
#include "file.h"

int var;

void increment2(void);
static void increment3(void);
void func(void);

void increment2(void) { /* Non compliant */
    var+=2;
}

static void increment3(void) { /* Compliant */
    var+=3;
}
```

```
void func(void) {
    increment1();
    increment2();
    increment3();
}
```

Second source file:

```
/* file2.c */
#include "file.h"

void increment1(void) { /* Compliant */
    var++;
}
```

In this example:

- The definition of `increment1` is compliant because `increment1` is defined with external linkage and called in a different file.
- The declaration of `increment2` is noncompliant because `increment2` is defined with external linkage but called in the same file and nowhere else.
- The declaration of `increment3` is compliant because `increment3` is defined with internal linkage (with the `static` specifier) and called in the same file and nowhere else.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.8

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Description

Rule Definition

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Rationale

If you do not use the `static` specifier consistently in all declarations of objects with internal linkage, you might declare the same object with external and internal linkage.

In this situation, the linkage follows the earlier specification that is visible (C99 Standard, Section 6.2.2). For instance, if the earlier specification indicates internal linkage, the object has internal linkage even though the latter specification indicates external linkage. If you notice the latter specification alone, you might expect otherwise.

Polyspace Implementation

The rule checker detects situations where:

- The same object is declared multiple times with different storage specifiers.
- The same function is declared and defined with different storage specifiers.

Message in Report

The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Linkage Conflict Between Variable Declarations

```
static int foo = 0;
extern int foo;         /* Non-compliant */

extern int hhh;
static int hhh;         /* Non-compliant */
```

In this example, the first line defines `foo` with internal linkage. The first line is compliant because the example uses the `static` keyword. The second line does not use `static` in the declaration, so the declaration is noncompliant. By comparison, the third line declares `hhh` with an `extern` keyword creating external linkage. The fourth line declares `hhh` with internal linkage, but this declaration conflicts with the first declaration of `hhh`.

Correction — Consistent `static` and `extern` Use

One possible correction is to use `static` and `extern` consistently:

```
static int foo = 0;
static int foo;

extern int hhh;
extern int hhh;
```

Linkage Conflict Between Function Declaration and Definition

```
static int fee(void);  /* Compliant - declaration: internal linkage */
int fee(void){         /* Non-compliant */
  return 1;
}

static int ggg(void);  /* Compliant - declaration: internal linkage */
extern int ggg(void){  /* Non-compliant */
```

```
  return 1 + x;
}
```

This example shows two internal linkage violations. Because `fee` and `ggg` have internal linkage, you must use a `static` class specifier to be compliant with MISRA.

Check Information

Group: Declarations and Definitions
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 8.9

An object should be defined at block scope if its identifier only appears in a single function

Description

Rule Definition

An object should be defined at block scope if its identifier only appears in a single function.

Rationale

If you define an object at block scope, you or someone else is less likely to access the object inadvertently outside the block.

Polyspace Implementation

The rule checker flags `static` objects that are accessed in one function only but declared at file scope.

Message in Report

An object should be defined at block scope if its identifier only appears in a single function.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Object Declared at File Scope but Used in One Function

```
static int ctr;   /* Non compliant */

int checkStatus(void);
void incrementCount(void);

void incrementCount(void) {
    ctr=0;
    while(1) {
        if(checkStatus())
            ctr++;
    }
}
```

In this example, the declaration of `ctr` is noncompliant because it is declared at file scope but used only in the function `incrementCount`. Declare `ctr` in the body of `incrementCount` to be MISRA C-compliant.

Check Information

Group: Declarations and Definitions
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 9.1

The value of an object with automatic storage duration shall not be read before it has been set

Description

Message in Report:

Rule Definition

The value of an object with automatic storage duration shall not be read before it has been set.

Rationale

A variable with an automatic storage duration is allocated memory at the beginning of an enclosing code block and deallocated at the end. All non-global variables have this storage duration, except those declared `static` or `extern`.

Variables with automatic storage duration are not automatically initialized and have indeterminate values. Therefore, you must not read such a variable before you have set its value through a write operation.

Polyspace Implementation

The Polyspace analysis checks some of the violations as non-initialized variables. For more information, see `Non-initialized local variable`.

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results. In Code Prover, you can also see a difference in results based on your choice for the option `Verification level (-to)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

The value of an object with automatic storage duration shall not be read before it has been set.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Initialization
Category: Mandatory
AGC Category: Mandatory
Language: C90, C99

See Also

`MISRA C:2012 Rule 15.1` | `MISRA C:2012 Rule 15.3`

Introduced in R2014b

MISRA C:2012 Rule 9.2

The initializer for an aggregate or union shall be enclosed in braces

Description

Rule Definition

The initializer for an aggregate or union shall be enclosed in braces.

Rationale

The rule applies to both objects and subobjects. For example, when initializing a structure that contains an array, the values assigned to the structure must be enclosed in braces. Within these braces, the values assigned to the array must be enclosed in another pair of braces.

Enclosing initializers in braces improves clarity of code that contains complex data structures such as multidimensional arrays and arrays of structures.

Tip To avoid nested braces for subobjects, use the syntax `{0}`, which sets all values to zero.

Message in Report

The initializer for an aggregate or union shall be enclosed in braces.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Initialization of Two-dimensional Arrays

```
void initialize(void) {
    int x[4][2] = {{0,0},{1,0},{0,1},{1,1}}; /* Compliant */
    int y[4][2] = {{0},{1,0},{0,1},{1,1}};   /* Compliant */
    int z[4][2] = {0};                       /* Compliant */
    int w[4][2] = {0,0,1,0,0,1,1,1};         /* Non-compliant */
}
```

In this example, the rule is not violated when:

- Initializers for each row of the array are enclosed in braces.
- The syntax `{0}` initializes all elements to zero.

The rule is violated when a separate pair of braces is not used to enclose the initializers for each row.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 9.3

Arrays shall not be partially initialized

Description

Rule Definition

Arrays shall not be partially initialized.

Rationale

Providing an explicit initialization for each array element makes it clear that every element has been considered.

Message in Report

Arrays shall not be partially initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Partial and Complete Initializations

```
void func(void) {
    int x[3] = {0,1,2};              /* Compliant */
    int y[3] = {0,1};                /* Non-compliant */
    int z[3] = {0};                  /* Compliant - exception */
    int a[30] = {[1] = 1,[15]=1};    /* Compliant - exception */
    int b[30] = {{1} = 1, 1};        /* Non-compliant */
```

```
    char c[20] = "Hello World";       /* Compliant - exception */
}
```

In this example, the rule is not violated when each array element is explicitly initialized.

The rule is violated when some elements of the array are implicitly initialized. Exceptions include the following:

- The initializer has the form `{0}`, which initializes all elements to zero.
- The array initializer consists *only* of designated initializers. Typically, you use this approach for sparse initialization.
- The array is initialized using a string literal.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 9.4

An element of an object shall not be initialized more than once

Description

Rule Definition

An element of an object shall not be initialized more than once.

Rationale

Designated initializers allow explicitly initializing elements of objects such as arrays in any order. However, using designated initializers, one can inadvertently initialize the same element twice and therefore overwrite the first initialization.

Message in Report

An element of an object shall not be initialized more than once.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Array Initialization Using Designated Initializers

```
void func(void) {
    int a[5] = {-2,-1,0,1,2};                /* Compliant */
    int b[5] = {[0]=-2, [1]=-1, [2]=0, [3]=1, [4]=2};
                                             /* Compliant */
    int c[5] = {[0]=-2, [1]=-1, [1]=0, [3]=1, [4]=2};
```

```
                                              /* Non-compliant */
}
```

In this example, the rule is violated when the array element `c[1]` is initialized twice using a designated initializer.

Structure Initialization Using Designated Initializers

```
struct myStruct {
    int a;
    int b;
    int c;
    int d;
};

void func(void) {
    struct myStruct struct1 = {-4,-2,2,4};   /* Compliant */
    struct myStruct struct2 = {.a=-4, .b=-2, .c=2, .d=4};
                                             /* Compliant */
    struct myStruct struct3 = {.a=-4, .b=-2, .b=2, .d=4};
                                             /* Non-compliant */
}
```

In this example, the rule is violated when `struct3.b` is initialized twice using a designated initializer.

Check Information

Group: Initialization
Category: Required
AGC Category: Required
Language: C99

See Also

Introduced in R2014b

MISRA C:2012 Rule 9.5

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

Description

Rule Definition

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of the elements that are initialized. When using long designated initializers, it might not be immediately apparent which element has the highest index.

Message in Report

Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Using Designated Initializers Without Specifying Array Size

```
int a[5] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};        /* Compliant */
int b[] = {[0]= 1, [2] = 1, [4]= 1, [1] = 1};         /* Non-compliant */
```

```
int c[] = {[0]= 1, [1] = 1, [2]= 1, [3]=0, [4] = 1};  /* Non-compliant */

void display(int);

void main() {
    func(a,5);
    func(b,5);
    func(c,5);
}

void func(int* arr, int size) {
    for(int i=0; i<size; i++)
        display(arr[i]);
}
```

In this example, the rule is violated when the arrays `b` and `c` are initialized using designated initializers but the array size is not specified.

Check Information

Group: Initialization
Category: Required
AGC Category: Readability
Language: C99

See Also

Introduced in R2014b

MISRA C:2012 Dir 1.1

Any implementation-defined behavior on which the output of the program depends shall be documented and understood

Description

Directive Definition

Any implementation-defined behavior on which the output of the program depends shall be documented and understood.

Rationale

A code construct has implementation-defined behavior if the C standard allows compilers to choose their own specifications for the construct. The full list of implementation-defined behavior is available in Annex J.3 of the standard ISO/IEC 9899:1999 (C99) and in Annex G.3 of the standard ISO/IEC 9899:1990 (C90).

If you understand and document all implementation-defined behavior, you can be assured that all output of your program is intentional and not produced by chance.

Polyspace Implementation

The analysis detects the following possibilities of implementation-defined behavior in C99 and their counterparts in C90. If you know the behavior of your compiler implementation, justify the analysis result with appropriate comments. To justify a result, assign one of these statuses: `Justified`, `No action planned`, or `Not a defect`.

Tip To mass-justify all results that indicate the same implementation-defined behavior, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps

J.3.2: Environment	An alternative manner in which `main` function may be defined.	The analysis flags `main` with arguments and return types other than:
		`int main(void) { ... }`
		or
		`int main(int argc, char *argv[]) { ... }`
		See section 5.1.2.2.1 of the C99 Standard.

J.3.2: Environment	The set of environment names and the method for altering the environment list used by the `getenv` function.	The analysis flags uses of the `getenv` function. For this function, you need to know the list of environment variables and how the list is modified. See section 7.20.4.5 of the C99 Standard.
J.3.6: Floating Point	The rounding behaviors characterized by non-standard values of `FLT_ROUNDS`.	The analysis flags the include of `float.h` if values of `FLT_ROUNDS` are outside the set, {-1, 0, 1, 2, 3}. Only the values in this set lead to well-defined rounding behavior. See section 5.2.4.2.2 of the C99 Standard.
J.3.6: Floating Point	The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD`.	The analysis flags the include of `float.h` if values of `FLT_EVAL_METHOD` are outside the set, {-1, 0, 1, 2}. Only the values in this set lead to well-defined behavior for floating-point operations. See section 5.2.4.2.2 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.6: Floating Point	The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value.	The analysis flags conversions from integer to floating-point data types of smaller size (for example, 64-bit `int` to 32-bit `float`). See section 6.3.1.4 of the C99 Standard.
J.3.6: Floating Point	The direction of rounding when a floating-point number is converted to a narrower floating-point number.	The analysis flags these conversions: • `double` to `float` • `long double` to `double` or `float` See section 6.3.1.5 of the C99 Standard.

J.3.6: Floating Point	The default state for the `FENV_ACCESS` pragma.	The analysis flags use of the pragma other than:
		`#pragma STDC FENV_ACCESS ON`
		or
		`#pragma STDC FENV_ACCESS OFF`
		
		See section 7.6.1 of the C99 Standard.

J.3.6: Floating Point	The default state for the `FP_CONTRACT` pragma.	The analysis flags use of the pragma other than:
		`#pragma STDC FP_CONTRACT ON`
		or
		`#pragma STDC FP_CONTRACT OFF`
		
		See section 7.12.2 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.11: Preprocessing Directives	The behavior on each recognized non-STDC #pragma directive.	The analysis flags the pragma usage: `#pragma` *`pp-tokens`* where the processing token `STDC` does not immediately follow`pragma`. For instance: `#pragma FENV_ACCESS ON` See section 6.10.6 of the C99 Standard.
J.3.12: Library Functions	Whether the `feraiseexcept` function raises the ‘‘inexact’’ floating-point exception in addition to the ‘‘overflow’’ or ‘‘underflow’’ floating-point exception.	The analysis flags calls to the `feraiseexcept` function. See section 7.6.2.3 of the C99 Standard.
J.3.12: Library Functions	Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function.	The analysis flags calls to the `setlocale` function when its second argument is not `"C"` or `""`. See section 7.11.1.1 of the C99 Standard.
J.3.12: Library Functions	The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2.	The analysis flags the include of `math.h` if `FLT_EVAL_METHOD` has values outside the set {0,1,2}. See section 7.12 of the C99 Standard.

C99 Standard Annex Ref	Behavior to Be Documented	How Polyspace Helps
J.3.12: Library Functions	The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient.	The analysis flags calls to the `remquo`, `remquof` and `remquol` function. See section 7.12.10.3 of the C99 Standard.
J.3.12: Library Functions	The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function.	The analysis flags calls to the `abort`, `exit`, or `_Exit` function. See sections 7.20.4.1, 7.20.4.3 or 7.20.4.4 of the C99 Standard.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: The implementation
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2017b

MISRA C:2012 Dir 2.1

All source files shall compile without any compilation errors

Description

Directive Definition

All source files shall compile without any compilation errors.

Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program can produce unexpected behavior.

Polyspace Implementation

The software raises a violation of this directive if it finds a compilation error. Because Code Prover is more strict about compilation errors compared to Bug Finder, the coding rules checking in the two products can produce different results for this directive.

Message in Report

All source files shall compile without any compilation errors.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Compilation and build

Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Dir 4.1

Run-time failures shall be minimized

Description

Directive Definition

Run-time failures shall be minimized.

Rationale

Some areas to concentrate on are:

- Arithmetic errors
- Pointer arithmetic
- Array bound errors
- Function parameters
- Pointer dereferencing
- Dynamic memory

Polyspace Implementation

This directive is checked through the Polyspace analysis. For more information, see:

- "Defects" (Polyspace Bug Finder Access)
- "Run-Time Checks".

Polyspace Bug Finder and Polyspace Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

Run-time failures shall be minimized.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.11` | `MISRA C:2012 Rule 1.3` | `MISRA C:2012 Rule 18.1` | `MISRA C:2012 Rule 18.2` | `MISRA C:2012 Rule 18.3`

Introduced in R2014b

MISRA C:2012 Dir 4.10

Precautions shall be taken in order to prevent the contents of a header file being included more than once

Description

Directive Definition

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Rationale

When a translation unit contains a complex hierarchy of nested header files, it is possible for a particular header file to be included more than once, leading to confusion. If this multiple inclusion produces multiple or conflicting definitions, then your program can have undefined or erroneous behavior.

For instance, suppose that a header file contains:

```
#ifdef _WIN64
   int env_var;
#elseif
   long int env_var;
#endif
```

If the header file is contained in two inclusion paths, one that defines the macro `_WIN64` and another that undefines it, you can have conflicting definitions of `env_var`.

Polyspace Implementation

If you include a header file whose contents are not guarded from multiple inclusion, the analysis raises a violation of this directive. The violation is shown at the beginning of the header file.

You can guard the contents of a header file from multiple inclusion by using one of the following methods:

```
<start-of-file>
#ifndef <control macro>
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

or

```
<start-of-file>
#ifdef <control macro>
#error ...
#else
#define <control macro>
    /* Contents of file */
#endif
<end-of-file>
```

Unless you use one of these methods, Polyspace flags the header file inclusion as noncompliant.

Message in Report

Precautions shall be taken in order to prevent the contents of a header file being included more than once.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Code After Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func(void);
```

```
#endif
void func2(void);
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func2(void)` is outside the guard.

Note You can have comments outside the macro guard.

Code Before Macro Guard

```
void func(void);
#ifndef  __MY_MACRO__
#define __MY_MACRO__
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro guard does not cover the entire content of the header file. The line `void func(void)` is outside the guard.

Note You can have comments outside the macro guard.

Mismatch in Macro Guard

```
#ifndef  __MY_MACRO__
#define __MY_MARCO__
    void func(void);
    void func2(void);
#endif
```

If a header file contains this code, it is noncompliant because the macro name in the `#ifndef` statement is different from the name in the following `#define` statement.

Check Information

Group: Code Design

Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Dir 4.11

The validity of values passed to library functions shall be checked

Description

Directive Definition

The validity of values passed to library functions shall be checked.

Rationale

Many Standard C functions do not check the validity of parameters passed to them. Even if checks are performed by a compiler, there is no guarantee that the checks are adequate. For example, you should not pass negative numbers to `sqrt` or `log`.

Polyspace Implementation

Polyspace raises a violation result for library function arguments if the following are all true:

- Argument is a local variable.
- Local variable is not tested between last assignment and call to the library function.
- Corresponding parameter of the library function has a restricted input domain.
- Library function is one of the following common mathematical functions:
  - `sqrt`
  - `tan`
  - `pow`
  - `log`
  - `log10`
  - `fmod`
  - `acos`

- `asin`
- `acosh`
- `atanh`
- or `atan2`

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

Tip To mass-justify all results related to the same library function, use the **Detail** column on the **Results List** pane. Click the column header so that all results with the same entry are grouped together. Select the first result and then select the last result while holding the `Shift` key. Assign a status to one of the results. If you do not see the **Detail** column, right-click any other column header and enable this column.

Message in Report

The validity of values passed to library functions shall be checked

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Dir 4.1`

Introduced in R2014b

MISRA C:2012 Dir 4.13

Functions which are designed to provide operations on a resource should be called in an appropriate sequence

Description

Directive Definition

Functions which are designed to provide operations on a resource should be called in an appropriate sequence.

Rationale

You typically use functions operating on a resource in the following way:

1. You allocate the resource.

   For example, you open a file or critical section.
2. You use the resource.

   For example, you read from the file or perform operations in the critical section.
3. You deallocate the resource.

   For example, you close the file or critical section.

For your functions to operate as you expect, perform the steps in sequence. For instance, if you call a resource allocation function on a certain execution path, you must call a deallocation function on that path.

Polyspace Implementation

Polyspace Bug Finder detects a violation of this rule if you specify multitasking options and your code contains one of these defects:

- : A task calls an unlock function before calling the corresponding lock function.

- : A task calls a lock function but ends without a call to the corresponding unlock function.
- : A task calls a lock function twice without an intermediate call to an unlock function.
- : A task calls an unlock function twice without an intermediate call to a lock function.

For more information on the multitasking options, see .

Message in Report

Functions which are designed to provide operations on a resource should be called in an appropriate sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Multitasking: Lock Function That Is Missing Unlock Function

```
typedef signed int int32_t;
typedef signed short int16_t;

typedef struct tag_mutex_t {
    int32_t value;
} mutex_t;


extern mutex_t mutex_lock   ( void );
extern void  mutex_unlock (  mutex_t m );

extern int16_t x;
void func(void);

void task1(void) {
     func();
}
```

```
void task2(void) {
     func();
}

void func ( void ) {
    mutex_t m = mutex_lock ( );  /* Non-compliant */

    if ( x > 0 )  {
        mutex_unlock ( m );
    }  else  {
        /* Mutex not unlocked on this path */
    }
}
```

In this example, the rule is violated when:

- You specify that the functions `mutex_lock` and `mutex_unlock` are paired.

  `mutex_lock` begins a critical section and `mutex_unlock` ends it.
- The function `mutex_lock` is called. However, if `x <= 0`, the function `mutex_unlock` is not called.

To enable detection of this rule violation, you must specify these analysis options.

Option	Specification	
Configure multitasking manually	☑	
Entry points	task1 task2	
Critical section details	**Starting routine**	**Ending routine**
	mutex_lock	mutex_unlock

For more information on the options, see:

-
-

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Dir 4.14

The validity of values received from external sources shall be checked

Description

Directive Definition

The validity of values received from external sources shall be checked.

Rationale

The values originating from external sources can be invalid because of errors or deliberate modification by attackers. Before using the data, you must check the data for validity.

For instance:

- Before using an external input as array index, you must check if it can potentially cause an array bounds error.
- Before using a variable to control a loop, you must check if it can potentially result in an infinite loop.

Message in Report

The validity of values received from external sources shall be checked.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Validity of External Values Not Checked

```
#include <stdio.h>

void f1(char from_user[])
{
        char input [128];
        (void) sscanf (from_user, "%128c", input);
        (void) sprintf ("%s", input);
}
```

In this example, the `sscanf` statement is noncompliant as there is no check to ensure that the user input is null terminated. The subsequent `sprintf` statement that outputs the string can potentially lead to an array bounds error (buffer overrun).

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2017a

MISRA C:2012 Dir 4.3

Assembly language shall be encapsulated and isolated

Description

Directive Definition

Assembly language shall be encapsulated and isolated.

Rationale

Encapsulating assembly language is beneficial because:

- It improves readability.
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear.
- All uses of assembly language for a given purpose can share encapsulation, which improves maintainability.
- You can easily substitute the assembly language for a different target or for purposes of static analysis.

Polyspace Implementation

Polyspace does not raise a warning on assembly language code encapsulated in the following:

- `asm` functions or `asm` pragmas
- Macros

Message in Report

Assembly language shall be encapsulated and isolated

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Assembly Language Code in C Function

```
enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);

void taskHandler(void) {
    isTaskActive = FALSE;
    // Software interrupt for task switching
    asm volatile
    (
        "SWI &02"     /* Service #1: calculate run-time */
    );
    return;
}
```

In this example, the rule violation occurs because the assembly language code is embedded directly in a C function `taskHandler` that contains other C language statements.

Correction: Encapsulate Assembly Code in Macro

One possible correction is to encapsulate the assembly language code in a macro and invoke the macro in the function `taskHandler`.

```
#define  RUN_TIME_CALC \
asm volatile \
    ( \
        "SWI &02"     /* Service #1: calculate run-Time */ \
    )\

enum boolVal {TRUE, FALSE};
enum boolVal isTaskActive;
void taskHandler(void);
```

```
void taskHandler(void) {
    isTaskActive = FALSE;
    RUN_TIME_CALC;
    return;
}
```

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

`MISRA C:2012 Rule 1.2`

Introduced in R2014b

MISRA C:2012 Dir 4.5

Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Description

Directive Definition

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Rationale

What "unambiguous" means depends on the alphabet and language in which source code is written. When you use identifiers that are typographically close, you can confuse between them.

For the Latin alphabet as used in English words, at a minimum, the identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Message in Report

Identifiers in the same name space with overlapping visibility should be typographically unambiguous.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */

    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */

    int id5_zerO;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Code design
Category: Advisory
AGC Category: Readability
Language: C90, C99

See Also

Introduced in R2015b

MISRA C:2012 Dir 4.6

`typedefs` that indicate size and signedness should be used in place of the basic numerical types

Description

Directive Definition

`typedefs` that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker flags use of basic data types in variable or function declarations and definitions. The rule enforces use of `typedefs` instead.

The rule checker does not flag the use of basic types in the `typedef` statements themselves.

Message in Report

Typedefs that indicate size and signedness should be used in place of the basic numerical types

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Direct Use of Basic Types in Definitions

```
typedef unsigned int uint32_t;

int x = 0;        /* Non compliant */
uint32_t y = 0;  /* Compliant */
```

In this example, the declaration of `x` is noncompliant because it uses a basic type directly.

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2014b

MISRA C:2012 Dir 4.7

If a function returns error information, then that error information shall be tested

Description

Directive Definition

If a function returns error information, then that error information shall be tested.

Rationale

Typically a function indicates whether an error occurred during execution, via a special return value or by another means.

If a function provides a mechanism to determine errors, before you use the function return value, you must check for such errors.

Polyspace Implementation

The checking of this directive follows the same specifications as the defect checker .

This directive is only partially supported.

Message in Report

If a function returns error information, then that error information shall be tested.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Code design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2017a

MISRA C:2012 Dir 4.8

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden

Description

Rule Definition

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Rationale

If a pointer to a structure or union is not dereferenced in a file, the implementation details of the structure or union need not be available in the translation unit for the file. You can hide the implementation details such as structure members and protect them from unintentional changes.

Define an opaque type that can be referenced via pointers but whose contents cannot be accessed.

Polyspace Implementation

If a structure or union is defined in a file or a header file included in the file, a pointer to this structure or union declared but the pointer never dereferenced in the file, the checker flags a coding rule violation. The structure or union definition should not be visible to this file.

If you see a violation of this rule on a structure definition, identify if you have defined a pointer to the structure in the same file or in a header file included in the file. Then check if you dereference the pointer anywhere in the file. If you do not dereference the pointer, the structure definition should be hidden from this file and included header files.

Message in Report

If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Object Implementation Revealed

`file.h`: Contains structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct  {
  int a;
} myStruct;

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

myStruct* getObj(void);
void useObj(myStruct*);

void func() {
  myStruct *sPtr = getObj();
  useObj(sPtr);
}
```

In this example, the pointer to the type `myStruct` is not dereferenced. The pointer is simply obtained from the `getObj` function and passed to the `useObj` function.

The implementation of `myStruct` is visible in the translation unit consisting of `file.c` and `file.h`.

Correction — Define Opaque Type

One possible correction is to define an opaque data type in the header file `file.h`. The opaque data type `ptrMyStruct` points to the `myStruct` structure without revealing what the structure contains. The structure `myStruct` itself can be defined in a separate translation unit, in this case, consisting of the file `file2.c`. The common header file `file.h` must be included in both `file.c` and `file2.c` for linking the structure definition to the opaque type definition.

`file.h`: Does not contain structure implementation.

```
#ifndef TYPE_GUARD
#define TYPE_GUARD

typedef struct myStruct *ptrMyStruct;

ptrMyStruct getObj(void);
void useObj(ptrMyStruct);

#endif
```

`file.c`: Includes `file.h` but does not dereference structure.

```
#include "file.h"

void func() {
  ptrMyStruct sPtr = getObj();
  useObj(sPtr);
}
```

`file2.c`: Includes `file.h` and dereferences structure.

```
#include "file.h"

struct myStruct {
  int a;
};

void useObj(ptrMyStruct ptr) {
    (ptr->a)++;
}
```

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

Introduced in R2018a

MISRA C:2012 Dir 4.9

A function should be used in preference to a function-like macro where they are interchangeable

Description

Directive Definition

A function should be used in preference to a function-like macro where they are interchangeable.

Rationale

In most circumstances, use functions instead of macros. Functions perform argument type-checking and evaluate their arguments once, avoiding problems with potential multiple side effects.

Polyspace Implementation

Polyspace considers all function-like macro definitions.

Message in Report

A function should be used in preference to a function-like macro where they are interchangeable

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Code design
Category: Advisory
AGC Category: Advisory
Language: C90, C99

See Also

`MISRA C:2012 Rule 13.2` | `MISRA C:2012 Rule 20.7`

Introduced in R2014b

MISRA C:2012 Dir 4.12

Dynamic memory allocation shall not be used

Description

Rule Definition

Dynamic memory allocation shall not be used.

Rationale

Using dynamic memory allocation and deallocation routines provided by the Standard Library or third-party libraries can cause undefined behavior. For instance:

- You use `free` to deallocate memory that you did not allocate with `malloc`, `calloc`, or `realloc`.
- You use a pointer that points to a freed memory location.
- You access allocated memory that has no value stored into it.

Dynamic memory allocation and deallocation routines from third-party libraries are likely to exhibit similar undefined behavior.

If you choose to use dynamic memory allocation and deallocation routines, ensure that your program behavior is predictable. For example, ensure that you safely handle allocation failure due to insufficient memory.

Troubleshooting

If you expect a rule violation but do not see it, refer to the documentation of Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of malloc, calloc, realloc and free

```
#include <stdlib.h>

static int foo(void);

typedef struct struct_1 {
    int a;
    char c;
} S_1;

static int foo(void) {

    S_1 * ad_1;
    int  * ad_2;
    int  * ad_3;

    ad_1 = (S_1*)calloc(100U, sizeof(S_1));        /* Non-compliant */
    ad_2 = malloc(100U * sizeof(int));             /* Non-compliant */
    ad_3 = realloc(ad_3, 60U * sizeof(long));      /* Non-compliant */

    free(ad_1);                                    /* Non-compliant */
    free(ad_2);                                    /* Non-compliant */
    free(ad_3);                                    /* Non-compliant */

    return 1;
}
```

In this example, the rule is violated when the functions `malloc`, `calloc`, `realloc` and `free` are used.

Check Information

Group: Code Design
Category: Required
AGC Category: Required
Language: C90, C99

See Also

Introduced in R2019b

3

MISRA C++: 2008

MISRA C++:2008 Rule 0-1-1

A project shall not contain unreachable code

Description

Rule Definition

A project shall not contain unreachable code.

Rationale

This rule flags situations where a group of statements is unreachable because of syntactic reasons. For instance, code following a `return` statement are always unreachable.

Unreachable code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

A project shall not contain unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unreachable statements

```
int func(int arg) {
 int temp = 0;
 switch(arg) {
     temp = arg; // Noncompliant
     case 1:
     {
         break;
     }
     default:
     {
         break;
     }
 }
 return arg;
 arg++; // Noncompliant
}
```

These statements are unreachable:

- Statements inside a `switch` statement that do not belong to a `case` or `default` block.
- Statements after a `return` statement.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 0-1-2

A project shall not contain infeasible paths

Description

Rule Definition

A project shall not contain infeasible paths.

Rationale

This rule flags situations where a group of statements is redundant because of nonsyntactic reasons. For instance, an `if` condition is always true or false. Code that is unreachable from syntactic reasons are flagged by rule 0-1-1.

Unreachable or redundant code involve unnecessary maintenance and can often indicate programming errors.

Polyspace Implementation

Bug Finder and Code Prover check this rule differently. The analysis can produce different results.

- Bug Finder checks for this rule through the `Dead code` and `Useless if` checkers..
- Code Prover does not use run-time checks to detect violations of this rule. Instead, Code Prover detects the violations at compile time.

Message in Report

A project shall not contain infeasible paths.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Boolean Operations with Invariant Results

```
void func (unsigned int arg) {
 if (arg >= 0U) //Noncompliant
     arg  = 1U;
 if (arg < 0U) //Noncompliant
     arg = 1U;
}
```

An `unsigned int` variable is nonnegative. Both `if` conditions involving the variable are always true or always false and are therefore redundant.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 0-1-3

A project shall not contain unused variables

Description

Rule Definition

A project shall not contain unused variables.

Polyspace Implementation

The checker flags local or global variables that are declared or defined but not used anywhere in the source files. This specification also applies to members of structures and classes.

Message in Report

A project shall not contain unused variables.

Variable is never used or used only in unreachable code.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Named Bit Field for Padding

```
#include <iostream>
struct S {
    unsigned char b1 : 3;
```

```
    unsigned char pad: 1;  //Noncompliant
    unsigned char b2 : 4;
};
void init(struct S S_obj)
{
    S_obj.b1 = 0;
    S_obj.b2 = 0;
}
```

In this example, the bit field `pad` is used for padding the structure. Therefore, the field is never read or written and causes a violation of this rule. To avoid the violation, use an unnamed field for padding.

```
struct S {
    unsigned char b1 : 3;
    unsigned char : 1;
    unsigned char b2 : 4;
};
```

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 0-1-5

A project shall not contain unused type declarations

Description

Rule Definition

A project shall not contain unused type declarations.

Rationale

If a type is declared but not used, when reviewing the code later, it is unclear if the type is redundant or left unused by mistake.

Unused types can indicate coding errors. For instance, you declared a enumerated data type for some specialized data but used an integer type for the data.

Message in Report

A project shall not contain unused type declarations.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused enum Declaration

```
enum switchValue {low, medium, high}; //Noncompliant

void operate(int userInput) {
```

```
    switch(userInput) {
        case 0: // Turn on low setting
                 break;
        case 1: // Turn on medium setting
                 break;
        case 2: // Turn on high setting
                 break;
        default: // Return error
    }
}
```

In this example, the enumerated type `switchValue` is not used. Perhaps the intention was to use the type as `switch` input like this.

```
enum switchValue {low, medium, high}; //Compliant

void operate(switchValue userInput) {
    switch(userInput) {
        case low: // Turn on low setting
                   break;
        case medium: // Turn on medium setting
                     break;
        case high: // Turn on high setting
                   break;
        default: // Return error
    }
}
```

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 0-1-7

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used

Description

Rule Definition

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.

Rationale

The unused return value might indicate a coding error or oversight.

Overloaded operators are excluded from this rule because their usage must emulate built-in operators which might not use their return value.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

The value returned by a function having a non- void return type that is not an overloaded operator shall always be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Return Value Not Used

```
#include <iostream>
#include <new>

int assignMemory(int * ptr){
     int res = 1;
     ptr = new (std::nothrow) int;
     if(ptr==NULL) {
         res = 0;
     }
     return res;
}

void main() {
    int val;
    int status;

    assignMemory(&val);   //Noncompliant
    status = assignMemory(&val); //Compliant
    (void)assignMemory(&val); //Compliant

}
```

The first call to the function `assignMemory` is noncompliant because the return value is not used. The second and third calls use the return value. The return value from the second call is assigned to a local variable.

The return value from the third call is cast to `void`. Casting to `void` indicates deliberate non-use of the return value and cannot be a coding oversight.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 0-1-9

There shall be no dead code

Description

Rule Definition

There shall be no dead code.

Rationale

If an operation is reachable but removing the operation does not affect program behavior, the operation constitutes dead code. For instance, suppose that a variable is never read following a write operation. The write operation is redundant.

The presence of dead code can indicate an error in the program logic. Because a compiler can remove dead code, its presence can cause confusion for code reviewers.

Message in Report

There shall be no dead code.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Redundant Operations

```
#define ULIM 10000
```

```
int func(int arg) {
    int res;
    res = arg*arg + arg;
    if (res > ULIM)
        res = 0; //Noncompliant
    return arg;
}
```

In this example, the operations involving `res` are redundant because the function `func` returns its argument `arg`. All operations involving `res` can be removed without changing the effect of the function.

The checker flags the last write operation on `res` because the variable is never read after that point. The dead code can indicate an unintended coding error. For instance, you intended to return the value of `res` instead of `arg`.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2016b

MISRA C++:2008 Rule 0-1-10

Every defined function shall be called at least once

Description

Rule Definition

Every defined function shall be called at least once.

Rationale

If a function with a definition is not called, it might indicate a serious coding error. For instance, the function call is unreachable or a different function is called unintentionally.

Polyspace Implementation

The checker detects situations where a static function is defined but not called at all in its translation unit.

Message in Report

Every defined function shall be called at least once. The static function *funcName* is not called.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Uncalled Static Function

```
static void func1() {
}

static void func2() { //Noncompliant
}

void func3();

int main() {
    func1();
    return 0;
}
```

The `static` function `func2` is defined but not called.

The function `func3` is not called either, however, it is only declared and not defined. The absence of a call to `func3` does not violate the rule.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 0-1-11

There shall be no unused parameters (named or unnamed) in nonvirtual functions

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in nonvirtual functions.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Message in Report

There shall be no unused parameters (named or unnamed) in non-virtual functions.

Function *`funcName`* has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Parameters

```
typedef int (*callbackFn) (int a, int b);
```

```
int callback_1 (int a, int b) { //Compliant
    return a+b;
}

int callback_2 (int a, int b) { //Noncompliant
    return a;
}

int callback_3 (int, int b) { //Compliant - flagged by Polyspace
    return b;
}

int getCallbackNumber();
int getInput();

void main() {
    callbackFn ptrFn;
    int n = getCallbackNumber();
    int x = getInput(), y = getInput();
    switch(n) {
        case 0: ptrFn = &callback_1; break;
        case 1: ptrFn = &callback_2; break;
        default: ptrFn = &callback_3; break;
    }

    (*ptrFn)(x,y);
}
```

In this example, the three functions `callback_1`, `callback_2` and `callback_3` are used as callback functions. One of the three functions is called via a function pointer depending on a value obtained at run time.

- Function `callback_1` uses all its parameters and does not violate the rule.
- Function `callback_2` does not use its parameter `a` and violates this rule.
- Function `callback_3` also does not use its first parameter but it does not violate the rule because the parameter is unnamed. However, Polyspace flags the unused parameter as a rule violation. If you see a violation of this kind, justify the violation with comments. See .

Check Information

Group: Language Independent Issues

Category: Required

See Also

Introduced in R2016b

MISRA C++:2008 Rule 0-1-12

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it

Description

Rule Definition

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused parameters often indicate later design changes. You perhaps removed all uses of a specific parameter but forgot to remove the parameter from the parameter list.

Unused parameters constitute an unnecessary overhead. You can also inadvertently call the function with a different number of arguments causing a parameter mismatch.

Polyspace Implementation

Polyspace checks for unused parameters in virtual functions within single translation units.

For instance, if a base class contains a virtual method with an unused parameter but the derived class implementation of the method uses that parameter, the rule is not violated. However, if the base class and derived class are defined in different files, the checker, which operates file by file, flags a violation of this rule on the base class.

Message in Report

There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.

Function `funcName` has unused parameters.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Unused Parameter in Virtual Function

```
class base {
    public:
       virtual void assignVal (int arg1, int arg2) = 0; //Noncompliant
       virtual void assignAnotherVal (int arg1, int arg2) = 0;
};

class derived1: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg1 = 1;
       }
};

class derived2: public base {
    public:
       virtual void assignVal (int arg1, int arg2) {
           arg1 = 0;
       }
       virtual void assignAnotherVal (int arg1, int arg2) {
           arg2 = 1;
       }
};
```

In this example, the second parameter of the virtual method `assignVal` is not used in any of the derived class implementations of the method.

On the other hand, the implementation of the virtual method `assignAnotherVal` in derived class `derived1` uses the first parameter of the method. The implementation in derived2 uses the second parameter. Both parameters of assignAnotherVal are used and therefore the virtual method does not violate the rule.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2016b

MISRA C++:2008 Rule 0-2-1

An object shall not be assigned to an overlapping object

Description

Rule Definition

An object shall not be assigned to an overlapping object.

Rationale

When you assign an object to another object with overlapping memory, the behavior is undefined.

The exceptions are:

- You assign an object to another object with exactly overlapping memory and compatible type.
- You copy one object to another with `memmove`.

Message in Report

An object shall not be assigned to an overlapping object.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Assignment of Union Members

```
void func (void) {
    union {
        short i;
        int j;
    } a = {0}, b = {1};

    a.j = a.i;   //Noncompliant
    a = b;       //Compliant
}
```

In this example, the rule is violated when `a.i` is assigned to `a.j` because the two variables have overlapping regions of memory.

Check Information

Group: Language Independent Issues
Category: Required

See Also

Introduced in R2016b

MISRA C++:2008 Rule 1-0-1

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1"

Description

Rule Definition

All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".

Polyspace Implementation

The checker reports compilation errors as detected by a compiler that strictly adheres to the C++03 Standard (ISO/IEC 14882:2003).

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

The message has two parts:

- Rule statement:

  All code shall conform to ISO/IEC 14882:2003 "The C++ Standard Incorporating Technical Corrigendum 1".
- Compilation error message such as:

  Expected a ;

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: General
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-3-1

Trigraphs shall not be used

Description

Rule Definition

Trigraphs shall not be used.

Rationale

You denote trigraphs with two question marks followed by a specific third character (for instance,`'??-'` represents a `'~'` (tilde) character and `'??)'` represents a `']'`). These trigraphs can cause accidental confusion with other uses of two question marks.

For instance, the string

```
"(Date should be in the form ??-??-??)"
```

is transformed to

```
"(Date should be in the form ~~]"
```

but this transformation might not be intended.

Message in Report

Trigraphs shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-5-1

Digraphs should not be used

Description

Rule Definition

Digraphs should not be used.

Rationale

Digraphs are a sequence of two characters that are supposed to be treated as a single character. The checker flags use of these digraphs:

- `<%`, indicating `[`
- `%>`, indicating `]`
- `<:`, indicating `{`
- `:>`, indicating `}`
- `%:`, indicating #
- `%:%:`

When developing or reviewing code with digraphs, the developer or reviewer can incorrectly consider the digraph as a sequence of separate characters.

Message in Report

Digraphs should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Lexical Conventions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-7-1

The character sequence /* shall not be used within a C-style comment

Description

Rule Definition

The character sequence / shall not be used within a C-style comment.*

Rationale

If your code contains a `/*` in a `/* */` comment, it typically means that you have inadvertently commented out code. See the example that follows.

Polyspace Implementation

You cannot justify a violation of this rule using source code annotations.

Message in Report

The character sequence /* shall not be used within a C-style comment.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of `/*` in `/* */` Comment

```
void foo() {
    /* Initializer functions
```

```
     setup();
    /* Step functions */
}
```

In this example, the call to `setup()` is commented out because the ending `*/` is omitted, perhaps inadvertently. The checker flags this issue by highlighting the `/*` in the `/* */` comment.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-1

Different identifiers shall be typographically unambiguous

Description

Rule Definition

Different identifiers shall be typographically unambiguous.

Rationale

When you use identifiers that are typographically close, you can confuse between them.

The identifiers should not differ by:

- The interchange of a lowercase letter with its uppercase equivalent.
- The presence or absence of the underscore character.
- The interchange of the letter `O` and the digit `0`.
- The interchange of the letter `I` and the digit `1`.
- The interchange of the letter `I` and the letter `l`.
- The interchange of the letter `S` and the digit `5`.
- The interchange of the letter `Z` and the digit `2`.
- The interchange of the letter `n` and the letter `h`.
- The interchange of the letter `B` and the digit `8`.
- The interchange of the letters `rn` and the letter `m`.

Polyspace Implementation

The rule checker does not consider the fully qualified names of variables when checking this rule.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

Different identifiers shall be typographically unambiguous.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Typographically Ambiguous Identifiers

```
void func(void) {
    int id1_numval;
    int id1_num_val;  /* Non-compliant */

    int id2_numval;
    int id2_numVal;   /* Non-compliant */

    int id3_lvalue;
    int id3_Ivalue;   /* Non-compliant */

    int id4_xyz;
    int id4_xy2;      /* Non-compliant */

    int id5_zerO;
    int id5_zer0;     /* Non-compliant */

    int id6_rn;
    int id6_m;        /* Non-compliant */
}
```

In this example, the rule is violated when identifiers that can be confused for each other are used.

Check Information

Group: Lexical Conventions

Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-2

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope

Description

Rule Definition

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

The rule flags situations where the same identifier name is used in two variable declarations, one in an outer scope and the other in an inner scope.

```
int var;
...
{
...
  int var;
...
}
```

All uses of the name in the inner scope refers to the variable declared in the inner scope. However, a developer or code reviewer can incorrectly assume that the usage refers to the variable declared in the outer scope.

Polyspace Implementation

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

The rule checker does not flag situations where the same identifier name is used in different logical scopes:

- The same name is used for a class data member and a variable outside the class.
- The same name is used for a method in a base and derived class.

Message in Report

Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Local Variable Hiding Global Variable

```
int varInit = 1;

void doSomething(void);

void step(void) {
    int varInit = 0; //Noncompliant
    if(varInit)
       doSomething();
}
```

In this example, `varInit` defined in `func` hides the global variable `varInit`. The `if` condition refers to the local `varInit` and the block is unreachable, but you might expect otherwise.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-3

A typedef name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A typedef name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

A typedef name (including qualification, if any) shall be a unique identifier.

Identifier *`typeName`* should not be reused.

Already used as typedef name (*`fileName`* *`lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Typedef Name Conflicting with Other Identifiers

```
namespace NS1 {
    typedef int WIDTH;
}

namespace NS2 {
    float WIDTH; //Compliant
}

void f1() {
    typedef int TYPE;
}

void f2() {
    float TYPE; //Noncompliant
}
```

In this example, the declaration of `TYPE` in `f2()` conflicts with a typedef declaration in `f1()`.

The checker does not flag the redeclaration of `WIDTH` because the two declarations belong to different namespaces.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-4

A class, union or enum name (including qualification, if any) shall be a unique identifier

Description

Rule Definition

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Rationale

The rule flags identifier declarations where the identifier name is the same as a previously declared class, union or typedef name. When you use identifiers that are identical, you can confuse between them.

Polyspace Implementation

The checker does not flag situations where the conflicting names occur in different namespaces.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

A class, union or enum name (including qualification, if any) shall be a unique identifier.

Identifier *`tagName`* should not be reused.

Already used as tag name (*`fileName`* *`lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Typedef Name Conflicting with Other Identifiers

```
void f1() {
    class floatVar {};
}

void f2() {
    float floatVar; //Noncompliant
}
```

In this example, the declaration of `floatVar` in `f2()` conflicts with a class declaration in `f1()`.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-5

The identifier name of a non-member object or function with static storage duration should not be reused

Description

Rule Definition

The identifier name of a non-member object or function with static storage duration should not be reused.

Rationale

The rule flags situations where the name of an identifier with static storage duration is reused. The rule applies even if the identifiers belong to different namespaces because the reuse leaves the chance that you mistake one identifier for the other.

Polyspace Implementation

The rule checker flags redefined functions only when there is a declaration.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

The identifier name of a non-member object or function with static storage duration should not be reused.

Identifier *name* should not be reused.

Already used as static identifier with static storage duration (*fileName* *lineNumber*).

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Reuse of Identifiers in Different Namespaces

```
namespace NS1 {
    static int WIDTH;
}

namespace NS2 {
    float WIDTH; //Noncompliant
}
```

In this example, the identifier name `WIDTH` is reused in the two namespaces `NS1` and `NS2`.

Check Information

Group: Lexical Conventions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-10-6

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope

Description

Rule Definition

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Rationale

For compatibility with C, in C++, you are allowed to use the same name for a type and an object or function. However, the name reuse can cause confusion during development or code review.

Polyspace Implementation

If the identifier is a function and the function is both declared and defined, then the violation is reported only once.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Reuse of Name for Type and Object

```
struct vector{
    int x;
    int y;
    int z;
}vector; //Noncompliant
```

In this example, the name `vector` is used both for the structured data type and for an object of that type.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-13-1

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used

Description

Rule Definition

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

Rationale

Escape sequences are certain special characters represented in string and character literals. They are written with a backslash (\) followed by a character.

The C++ Standard (ISO/IEC 14882:2003, Sec. 2.13.2) defines a list of escape sequences. See Escape Sequences. Use of escape sequences (backslash followed by character) outside that list leads to undefined behavior.

Message in Report

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

`\`*`char`* is not an ISO/IEC escape sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Incorrect Escape Sequences

```
void func () {
  const char a[2] = "\k"; \\Noncompliant
  const char b[2] = "\b"; \\Compliant
}
```

In this example, `\k` is not a recognized escape sequence.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-13-2

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used

Description

Rule Definition

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Rationale

Octal constants are denoted by a leading zero. A developer or code reviewer can mistake an octal constant as a decimal constant with a redundant leading zero.

Octal escape sequences beginning with \ can also cause confusion. Inadvertently introducing an 8 or 9 in the digit sequence after \ breaks the escape sequence and introduces a new digit. A developer or code reviewer can ignore this issue and continue to treat the escape sequence as one digit.

Message in Report

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Octal Constants and Octal Escape Sequences

```
void func(void) {
  int busData[6];

  busData[0] = 100;
  busData[1] = 108;
  busData[2] = 052;      //Noncompliant
  busData[3] = 071;      //Noncompliant
  busData[4] = '\109';   //Noncompliant
  busData[5] = '\100';   //Noncompliant

}
```

The checker flags all octal constants (other than zero) and all octal escape sequences (other than `\0`).

In this example:

- The octal escape sequence contains the digit 9, which is not an octal digit. This escape sequence has implementation-defined behavior.
- The octal escape sequence `\100` represents the number 64, but the rule checker forbids this use.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-13-3

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type

Description

Rule Definition

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Rationale

The signedness of a constant is determined from:

- Value of the constant.
- Base of the constant: octal, decimal or hexadecimal.
- Size of the various types.
- Any suffixes used.

Unless you use a suffix `u` or `U`, another developer looking at your code cannot determine easily whether a constant is signed or unsigned.

Message in Report

A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Lexical Conventions

Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-13-4

Literal suffixes shall be upper case

Description

Rule Definition

Literal suffixes shall be upper case.

Rationale

Literal constants can end with the letter `l` (el). Enforcing literal suffixes to be upper case removes potential confusion between the letter `l` and the digit `1`.

For consistency, use upper case constants for other suffixes such as `U` (unsigned) and `F` (float).

Message in Report

Literal suffixes shall be upper case.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Literal Constants with Lower Case Suffix

```
const int a = 0l; //Noncompliant
const int b = 0L; //Compliant
```

In this example, both `a` and `b` are assigned the same literal constant. However, from a quick glance, one can mistakenly assume that `a` is assigned the value `01` (octal one).

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 2-13-5

Narrow and wide string literals shall not be concatenated

Description

Rule Definition

Narrow and wide string literals shall not be concatenated.

Rationale

Narrow string literals are enclosed in double quotes without a prefix. Wide string literals are enclosed in double quotes with a prefix L outside the quotes. See string literals.

Concatenation of narrow and wide string literals can lead to undefined behavior.

Message in Report

Narrow and wide string literals shall not be concatenated.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Concatenation of Narrow and Wide String Literals

```
char array[] = "Hello" "World";
wchar_t w_array[] = L"Hello" L"World";
wchar_t mixed[] = "Hello" L"World"; //Noncompliant
```

In this example, in the initialization of the array `mixed`, the narrow string literal `"Hello"` is concatenated with the wide string literal `L"World"`.

Check Information

Group: Lexical Conventions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-1-1

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule

Description

Rule Definition

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rationale

If a header file with variable or function definitions appears in multiple inclusion paths, the header file violates the One Definition Rule possibly leading to unpredictable behavior. For instance, a source file includes the header file `include.h` and another header file, which also includes `include.h`.

Polyspace Implementation

The rule checker flags variable and function definitions in header files.

Message in Report

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-1-2

Functions shall not be declared at block scope

Description

Rule Definition

Functions shall not be declared at block scope.

Rationale

It is a good practice to place all declarations at the namespace level.

Additionally, if you declare a function at block scope, it is often not clear if the statement is a function declaration or an object declaration with a call to the constructor.

Message in Report

Functions shall not be declared at block scope.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function Declarations at Block Scope

```
class A {
};

void b1() {
```

```
    void func(); //Noncompliant
    A a();   //Noncompliant
}
```

In this example, the declarations of `func` and `a` are in the block scope of `b1`.

The second function declaration can cause confusion because it is not clear if `a` is a function that returns an object of type `A` or `a` is itself an object of type `A`.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-1-3

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization

Description

Rule Definition

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Rationale

Though you can declare an incomplete array type and later complete the type, specifying the array size during the first declaration makes the subsequent array access less error-prone.

Message in Report

When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Size of array `arrayName` should be explicitly stated.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Array Size Unspecified During Declaration

```
int array[10];
extern int array2[]; //Noncompliant
int array3[]= {0,1,2};
extern int array4[10];
```

In the declaration of `array2`, the array size is unspecified.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-2-1

All declarations of an object or function shall have compatible types

Description

Rule Definition

All declarations of an object or function shall have compatible types.

Rationale

If the declarations of an object or function in two different translation units have incompatible types, the behavior is undefined.

Message in Report

All declarations of an object or function shall have compatible types.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-2-2

The One Definition Rule shall not be violated

Description

Rule Definition

The One Definition Rule shall not be violated.

Rationale

Violations of the One Definition Rule leads to undefined behavior.

Polyspace Implementation

The checker flags situations where the same function or object has multiple definitions and the definitions differ by some token.

Message in Report

The One Definition Rule shall not be violated.

Declaration of class *`className`* violates the One Definition Rule:

it conflicts with other declaration (*`fileName`* *`lineNumber`*).

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Different Tokens in Same Type Definition

This example uses two files:

- `file1.cpp`:

  ```
  struct S
  {
     int x;
     int y;
  };
  ```
- `file2.cpp`:

  ```
  struct S
  {
     int y;
     int x;
  };
  ```

In this example, both `file1.cpp` and `file2.cpp` define the structure `S`. However, the definitions switch the order of the structure fields.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-2-3

A type, object or function that is used in multiple translation units shall be declared in one and only one file

Description

Rule Definition

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Rationale

If you declare an identifier in a header file, you can include the header file in any translation unit where the identifier is defined or used. In this way, you ensure consistency between:

- The declaration and the definition.
- The declarations in different translation units.

The rule enforces the practice of declaring external objects or functions in header files.

Message in Report

A type, object or function that is used in multiple translation units shall be declared in one and only one file.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-2-4

An identifier with external linkage shall have exactly one definition

Description

Rule Definition

An identifier with external linkage shall have exactly one definition.

Rationale

If an identifier has multiple definitions or no definitions, it can lead to undefined behavior.

Message in Report

An identifier with external linkage shall have exactly one definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Multiple Definitions of Identifier

This example uses two files:

- `file1.cpp`:

  ```
  int x = 0;
  ```
- `file2.cpp`:

```
int x = 1;
```

The same identifier x is defined in both files.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-3-1

Objects or functions with external linkage shall be declared in a header file

Description

Rule Definition

Objects or functions with external linkage shall be declared in a header file.

Rationale

If you declare a function or object in a header file, it is clear that the function or object is meant to be accessed in multiple translation units. If you intend to access the function or object from a single translation unit, declare it `static` or in an unnamed namespace.

Message in Report

Objects or functions with external linkage shall be declared in a header file.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Declaration in Header File Missing

This example uses two files:

- `decls.h`:

  ```
  extern int x;
  ```

- `file.cpp`:

```
#include "decls.h"

int x = 0;
int y = 0; //Noncompliant
static int z = 0;
```

In this example, the variable `x` is declared in a header file but the variable `y` is not. The variable `z` is also not declared in a header file but it is declared with the `static` specifier and does not have external linkage.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-3-2

If a function has internal linkage then all re-declarations shall include the static storage class specifier

Description

Rule Definition

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Rationale

If a function declaration has the `static` storage class specifier, it has internal linkage. Subsequent redeclarations of the function have internal linkage even without the `static` specifier.

However, if you do not specify the `static` keyword explicitly, it is not immediately clear from a declaration whether the function has internal linkage.

Message in Report

If a function has internal linkage then all re-declarations shall include the static storage class specifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Missing `static` Specifier from Redeclaration

```
static void func1 ();
static void func2 ();

void func1() {}  //Noncompliant
static void func2() {}
```

In this example, the function `func1` is declared `static` but defined without the `static` specifier.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-4-1

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility

Description

Rule Definition

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Rationale

Defining variables with the minimum possible block scope reduces the possibility that they might later be accessed unintentionally.

For instance, if an object is meant to be accessed in one function only, declare the object local to the function.

Polyspace Implementation

The rule checker determines if an object is used in one block only. If the object is used in one block but defined outside the block, the checker raises a violation.

Message in Report

An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Global Variable in Single Function

```
static int countReset; //Noncompliant

volatile int check;

void increaseCount() {
    int count = countReset;
    while(check%2) {
     count++;
    }
}
```

In this example, the variable `countReset` is declared global used in one function only. A compliant solution declares the variable local to the function to reduce its visibility.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-9-1

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations

Description

Rule Definition

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Rationale

If a redeclaration is not token-for-token identical to the previous declaration, it is not clear from visual inspection which object or function is being redeclared.

Polyspace Implementation

The rule checker compares the current declaration with the last seen declaration.

Message in Report

The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

Variable *`varName`* is not compatible with previous declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Identical Declarations That Do Not Match Token for Token

```
typedef int* intptr;

int* map;
extern intptr map; //Noncompliant

intptr table;
extern intptr table; //Compliant
```

In this example, the variable `map` is declared twice. The second declaration uses a `typedef` which resolves to the type of the first declaration. Because of the `typedef`, the second declaration is not token-for-token identical to the first.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-9-2

typedefs that indicate size and signedness should be used in place of the basic numerical types

Description

Rule Definition

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rationale

When the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Polyspace Implementation

The rule checker does not raise violations in templates that are not instantiated.

Message in Report

typedefs that indicate size and signedness should be used in place of the basic numerical types.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Direct Use of Basic Numerical Types

```
typedef unsigned int uint32_t;

unsigned int x = 0;       //Noncompliant
uint32_t y = 0;  //Compliant
```

In this example, the declaration of `x` is noncompliant because it uses the basic type `int` directly.

Check Information

Group: Basic Concepts
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 3-9-3

The underlying bit representations of floating-point values shall not be used

Description

Rule Definition

The underlying bit representations of floating-point values shall not be used.

Rationale

The underlying bit representations of floating point values vary across compilers. If you directly use the underlying representation of floating point values, your program is not portable across implementations.

Polyspace Implementation

The rule checker flags conversions from pointers to floating point types into pointers to integer types, and vice versa.

Message in Report

The underlying bit representations of floating-point values shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Using Underlying Representation of Floating-Point Values

```
float fabs2(float f) {
    unsigned int* ptr = reinterpret_cast <unsigned int*> (&f); //Noncompliant
    *(ptr + 3) &= 0x7f;
    return f;
}
```

In this example, the `reinterpret_cast` attempts to cast a floating-point value to an integer and access the underlying bit representation of the floating point value.

Check Information

Group: Basic Concepts
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 4-5-1

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator

Description

Rule Definition

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Rationale

Operators other than the ones mentioned in the rule do not produce meaningful results with `bool` operands. Use of `bool` operands with these operators can indicate programming errors. For instance, you intended to use the logical operator `||` but used the bitwise operator `|` instead.

Message in Report

Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Compliant and Noncompliant Uses of bool Operands

```
void boolOperations() {
    bool lhs = true;
    bool rhs = false;

    int res;

    if(lhs & rhs) {} //Noncompliant
    if(lhs < rhs) {} //Noncompliant
    if(~rhs) {}      //Noncompliant
    if(lhs ^ rhs) {} //Noncompliant
    if(lhs == rhs) {} //Compliant
    if(!rhs) {}       //Compliant
    res = lhs? -1:1;  //Compliant
}
```

In this example, `bool` operands do not violate the rule when used with the `==`, `!` and the `?` operators.

Check Information

Group: Standard Conversions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 4-5-2

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=

Description

Rule Definition

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Message in Report

Expressions with type enum shall not be used as operands to built- in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Standard Conversions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 4-5-3

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator N

Description

Rule Definition

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Rationale

The C++03 Standard only requires that the characters `'0'` to `'9'` have consecutive values. Other characters do not have well-defined values. If you use these characters in operations other than the ones mentioned in the rule, you implicitly use their underlying values and might see unexpected results.

Message in Report

Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator. N

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Compliant and Noncompliant Uses of Character Operands

```
void charManipulations (char ch) {

    char initChar = 'a'; //Compliant
    char finalChar = 'z'; //Compliant

    if(ch == initChar) {} //Compliant
    if( (ch >= initChar) && (ch <= finalChar)) {} //Noncompliant
    else if( (ch >= '0') && (ch <= '9') ) {} //Compliant by exception
}
```

In this example, character operands do not violate the rule when used with the `=` and `==` operators. Character operands can also be used with relational operators as long as the comparison is performed with the digits `'0'` to `'9'`.

Check Information

Group: Standard Conversions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 4-10-1

NULL shall not be used as an integer value

Description

Rule Definition

NULL shall not be used as an integer value.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of NULL to null pointer constants. `MISRA C++:2008 Rule 4-10-2` restricts the use of the literal 0 to integers.

Polyspace Implementation

The checker flags assignment of NULL to an integer variable or binary operations involving NULL and an integer. Assignments can be direct or indirect such as passing NULL as integer argument to a function.

Message in Report

NULL shall not be used as an integer value.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Compliant and Noncompliant Uses of NULL

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(NULL); //Noncompliant
    checkPointer(NULL); //Compliant
}
```

In this example, the use of `NULL` as argument to the `checkInteger` function is noncompliant because the function expects an `int` argument.

Check Information

Group: Standard Conversions
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 4-10-2

Literal zero (0) shall not be used as the null-pointer-constant

Description

Rule Definition

Literal zero (0) shall not be used as the null-pointer-constant.

Rationale

In C++, you can use the literals 0 and NULL as both an integer and a null pointer constant. However, use of 0 as a null pointer constant or NULL as an integer can cause developer confusion.

This rule restricts the use of the literal 0 to integers. `MISRA C++:2008 Rule 4-10-1` restricts the use of NULL to null pointer constants.

Polyspace Implementation

The checker flags assignment of 0 to a pointer variable or binary operations involving 0 and a pointer. Assignments can be direct or indirect such as passing 0 as pointer argument to a function.

Message in Report

Literal zero (0) shall not be used as the null-pointer-constant.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Compliant and Noncompliant Uses of Literal 0

```
#include <cstddef>

void checkInteger(int);
void checkPointer(int *);

void main() {
    checkInteger(0); //Compliant
    checkPointer(0); //Noncompliant
}
```

In this example, the use of 0 as argument to the `checkPointer` function is noncompliant because the function expects an `int *` argument.

Check Information

Group: Standard Conversions
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 5-0-1

The value of an expression shall be the same under any order of evaluation that the standard permits

Description

Rule Definition

The value of an expression shall be the same under any order of evaluation that the standard permits.

Rationale

If an expression results in different values depending on the order of evaluation, its value becomes implementation-defined.

Polyspace Implementation

An expression can have different values under the following conditions:

- The same variable is modified more than once in the expression, or is both read and written.
- The expression allows more than one order of evaluation.

Therefore, the rule checker forbids expressions where a variable is modified more than once and can cause different results under different orders of evaluation. The rule checker also detects cases where a volatile variable is read more than once in an expression.

Message in Report

The value of an expression shall be the same under any order of evaluation that the standard permits.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Variable Modified More Than Once in Expression

```
int a[10], b[10];
#define COPY_ELEMENT(index) (a[(index)]=b[(index)])

void main () {
    int i=0, k=0;

    COPY_ELEMENT (k);         /* Compliant */
    COPY_ELEMENT (i++);       /* Non-compliant  */
}
```

In this example, the rule is violated by the statement `COPY_ELEMENT(i++)` because `i++` occurs twice and the order of evaluation of the two expressions is unspecified.

Variable Modified and Used in Multiple Function Arguments

```
void f (unsigned int param1, unsigned int param2) {}

void main () {
    unsigned int i=0;
    f ( i++, i );                /* Non-compliant */
}
```

In this example, the rule is violated because it is unspecified whether the operation `i++` occurs before or after the second argument is passed to `f`. The call `f(i++,i)` can translate to either `f(0,0)` or `f(0,1)`.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-2

Limited dependence should be placed on C++ operator precedence rules in expressions

Description

Rule Definition

Limited dependence should be placed on C++ operator precedence rules in expressions.

Rationale

Use parentheses to clearly indicate the order of evaluation.

Depending on operator precedence can cause the following issues:

- If you or another code reviewer reviews the code, the intended order of evaluation is not immediately clear.
- It is possible that the result of the evaluation does not meet your expectations. For instance:

  - In the operation `*p++`, it is possible that you expect the dereferenced value to be incremented. However, the pointer `p` is incremented before the dereference.
  - In the operation `(x == y | z)`, it is possible that you expect `x` to be compared with `y | z`. However, the == operation happens before the | operation.

Message in Report

Limited dependence should be placed on C++ operator precedence rules in expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Evaluation Order Dependent on Operator Precedence Rules

```
#include <cstdio>

void showbits(unsigned int x) {
    for(int i = (sizeof(int) * 8) - 1; i >= 0; i--) {
       (x & 1u << i) ? putchar('1') : putchar('0'); // Noncompliant
    }
    printf("\n");
}
```

In this example, the checker flags the operation `x & 1u << i` because the statement relies on operator precedence rules for the << operation to happen before the & operation. If this is the intended order, the operation can be rewritten as `x & (1u << i)`.

Check Information

Group: Expressions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-3

A cvalue expression shall not be implicitly converted to a different underlying type

Description

Rule Definition

A cvalue expression shall not be implicitly converted to a different underlying type.

Rationale

This rule ensures that the result of the expression does not overflow when converted to a different type.

Polyspace Implementation

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

The underlying data type of a cvalue expression is the widest of operand data types in the expression. For instance, if you add two variables, one of type `int8_t` (`typedef` for `char`) and another of type `int32_t` (`typedef` for `int`), the addition has underlying type `int32_t`. If you assign the sum to a variable of type `int8_t`, the rule is violated.

Message in Report

A cvalue expression shall not be implicitly converted to a different underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Implicit Conversion of Cvalue Expression

```
typedef char int8_t;
typedef signed int int32_t;

void func ( )
  {
    int32_t s32;
    int8_t s8;
    s32 = s8 + s8; //Noncompliant
    s32 = s32 + s8; //Compliant
  }
```

In this example, the rule is violated when two variables of type `int8_t` are added and the result is assigned to a variable of type `int32_t`. The underlying type of the addition does not take into account the integer promotion involved and is simply the widest of operand data types, in this case, `int8_t`.

The rule is not violated if one of the operands has type `int32_t` and the result is assigned to a variable of type `int32_t`. In this case, the underlying data type of the addition is the same as the type of the variable to which the result is assigned.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-4

An implicit integral conversion shall not change the signedness of the underlying type

Description

Rule Definition

An implicit integral conversion shall not change the signedness of the underlying type.

Rationale

Some conversions from signed to unsigned data types can lead to implementation-defined behavior. You can see unexpected results from the conversion.

Polyspace Implementation

The checker flags implicit conversions from a signed to an unsigned integer data type or vice versa.

The checker assumes that `ptrdiff_t` is a signed integer.

Message in Report

An implicit integral conversion shall not change the signedness of the underlying type.

Implicit conversion of one of the binary + operands whose underlying types are *`typename_1`* and *`typename_2`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Implicit Conversions that Change Signedness

```
typedef char int8_t;
typedef unsigned char uint8_t;

void func()
  {
    int8_t s8;
    uint8_t u8;

    s8 = u8; //Noncompliant
    u8 = s8 + u8; //Noncompliant
    u8 = static_cast< uint8_t > ( s8 ) + u8; //Compliant
}
```

In this example, the rule is violated when a variable with a variable with signed data type is implicitly converted to a variable with unsigned data type or vice versa. If the conversion is explicit, as in the preceding example, the rule violation does not occur.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-5

There shall be no implicit floating-integral conversions

Description

Rule Definition

There shall be no implicit floating-integral conversions.

Rationale

If you convert from a floating point to an integer type, you lose information. Unless you explicitly cast from floating point to an integer type, it is not clear whether the loss of information is intended. Additionally, if the floating-point value cannot be represented in the integer type, the behavior is undefined.

Conversion from an integer to floating-point type can result in an inexact representation of the value. The error from conversion can accumulate over later operations and lead to unexpected results.

Polyspace Implementation

The checker flags implicit conversions between floating-point types (`float` and `double`) and integer types (`short`, `int`, etc.).

This rule takes precedence over 5-0-4 and 5-0-6 if they apply at the same time.

Message in Report

There shall be no implicit floating-integral conversions.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Conversion Between Floating Point and Integer Types

```
typedef signed int int32_t;
typedef float float32_t;

void func ( )
  {
    float32_t f32;
    int32_t   s32;
    s32 = f32;   //Noncompliant
    f32 = s32;   //Noncompliant
    f32 = static_cast< float32_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a floating-point type is *implicitly* converted to an integer type. The violation does not occur if the conversion is explicit.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-6

An implicit integral or floating-point conversion shall not reduce the size of the underlying type

Description

Rule Definition

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Rationale

A conversion that reduces the size of the underlying type can result in loss of information. Unless you explicitly cast to the narrower type, it is not clear whether the loss of information is intended.

Polyspace Implementation

The checker flags implicit conversions that reduce the size of a type.

If the conversion is to a narrower integer with a different sign, then rule 5-0-4 takes precedence over rule 5-0-6. Only rule 5-0-4 is shown.

Message in Report

An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Conversion That Reduces Size of Type

```
typedef signed short int16_t;
typedef signed int int32_t;

void func ( )
  {
    int16_t   s16;;
    int32_t   s32;
    s16 = s32;   //Noncompliant
    s16 = static_cast< int16_t > ( s32 ); //Compliant
  }
```

In this example, the rule is violated when a type is *implicitly* converted to a narrower type. The violation does not occur if the conversion is explicit.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-7

There shall be no explicit floating-integral conversions of a cvalue expression

Description

Rule Definition

There shall be no explicit floating-integral conversions of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the result of an integer division is then cast to a floating-point type.

```
short num;
short den;
float res;
res= static_cast<float> (num/den);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a floating-point division because of the later cast.

Message in Report

There shall be no explicit floating-integral conversions of a cvalue expression.

Complex expression of underlying type *`typeBeforeConversion`* may only be cast to narrower integer type of same signedness, however the destination type is *`typeAfterconversion`*.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Conversion of Division Result from Integer to Floating Point

```
void func() {
     short num;
     short den;
     short res_short;
     float res_float;

     res_float = static_cast<float> (num/den); //Noncompliant

     res_short = num/den;
     res_short = static_cast<float> (res_float); //Compliant

}
```

In this example, the first cast on the division result violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the expression is evaluated with an underlying type `float`.
- The second cast makes it clear that the expression is evaluated with the underlying type `short`. The result is then cast to the type `float`.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-8

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression). For instance, in this example, the sum of two `short` operands is cast to the wider type `int`.

```
short op1;
short op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Message in Report

An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

Complex expression of underlying type *typeBeforeConversion* may only be cast to narrower integer type of same signedness, however the destination type is *typeAfterconversion*.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Conversion of Sum to Wider Integer Type

```
void func() {
     short op1;
     short op2;
     int res;

     res = static_cast<int> (op1 + op2); //Noncompliant
     res = static_cast<int> (op1) + op2; //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int`.
- The second cast first converts one of the operands to `int` so that the sum is actually evaluated with the underlying type `int`.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-9

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression

Description

Rule Definition

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Rationale

Expressions flagged by this checker follow the detailed specifications for cvalue expressions from the MISRA C++ documentation.

If you evaluate an expression and later cast the result to a different type, the cast has no effect on the underlying type of the evaluation (the widest of operand data types in the expression).. For instance, in this example, the sum of two `unsigned int` operands is cast to the type `int`.

```
unsigned int op1;
unsigned int op2;
int res;
res= static_cast<int> (op1 + op2);
```

However, a developer or code reviewer can expect that the evaluation uses the data type to which the result is cast later. For instance, one can expect a sum with the underlying type `int` because of the later cast.

Message in Report

An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Conversion of Sum to Wider Integer Type

```
typedef int int32_t;
typedef unsigned int uint32_t;

void func() {
     uint32_t op1;
     uint32_t op2;
     int32_t res;

     res = static_cast<int32_t> (op1 + op2); //Noncompliant
     res = static_cast<int32_t> (op1) +
           static_cast<int32_t> (op2); //Compliant

}
```

In this example, the first cast on the sum violates the rule but the second cast does not.

- The first cast can lead to the incorrect expectation that the sum is evaluated with an underlying type `int32_t`.
- The second cast first converts each of the operands to `int32_t` so that the sum is actually evaluated with the underlying type `int32_t`.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-10

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand

Description

Rule Definition

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Message in Report

If the bitwise operators ~ and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-11

The plain char type shall only be used for the storage and use of character values

Description

Rule Definition

The plain char type shall only be used for the storage and use of character values.

Polyspace Implementation

The checker raises a violation when a value of signed or unsigned integer type is implicitly converted to the plain char type.

Message in Report

The plain char type shall only be used for the storage and use of character values.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2015a

MISRA C++:2008 Rule 5-0-12

Signed char and unsigned char type shall only be used for the storage and use of numeric values

Description

Rule Definition

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Message in Report

Signed char and unsigned char type shall only be used for the storage and use of numeric values.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2015a

MISRA C++:2008 Rule 5-0-13

The condition of an if-statement and the condition of an iteration- statement shall have type bool

Description

Rule Definition

The condition of an if-statement and the condition of an iteration- statement shall have type bool.

Message in Report

The condition of an if-statement and the condition of an iteration- statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-14

The first operand of a conditional-operator shall have type bool

Description

Rule Definition

The first operand of a conditional-operator shall have type bool.

Message in Report

The first operand of a conditional-operator shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-15

Array indexing shall be the only form of pointer arithmetic

Description

Rule Definition

Array indexing shall be the only form of pointer arithmetic.

Polyspace Implementation

The checker flags:

- Arithmetic operations on all pointers, for instance `p+I`, `I+p` and `p-I`, where `p` is a pointer and `I` an integer..
- Array indexing on nonarray pointers.

Message in Report

Array indexing shall be the only form of pointer arithmetic.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-17

Subtraction between pointers shall only be applied to pointers that address elements of the same array

Description

Rule Definition

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

Message in Report

Subtraction between pointers shall only be applied to pointers that address elements of the same array.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-18

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array

Description

Rule Definition

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Polyspace Implementation

Use Bug Finder for this checker. The rule checker performs the same checks as `Subtraction or comparison between pointers to different arrays`. Code Prover can fail to detect some violations.

The checker ignores casts when showing the violation on relational operator use with pointers types.

Message in Report

>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-19

The declaration of objects shall contain no more than two levels of pointer indirection

Description

Rule Definition

The declaration of objects shall contain no more than two levels of pointer indirection.

Message in Report

The declaration of objects shall contain no more than two levels of pointer indirection.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-20

Non-constant operands to a binary bitwise operator shall have the same underlying type

Description

Rule Definition

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Message in Report

Non-constant operands to a binary bitwise operator shall have the same underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-0-21

Bitwise operators shall only be applied to operands of unsigned underlying type

Description

Rule Definition

Bitwise operators shall only be applied to operands of unsigned underlying type.

Message in Report

Bitwise operators shall only be applied to operands of unsigned underlying type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-1

Each operand of a logical && or || shall be a postfix-expression

Description

Rule Definition

Each operand of a logical && or || shall be a postfix-expression.

Polyspace Implementation

During preprocessing, violations of this rule are detected on the expressions in `#if` directives.

The checker allows exceptions on associativity (a && b && c), (a || b || c).

Message in Report

Each operand of a logical && or || shall be a postfix-expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-2

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast

Description

Rule Definition

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Message in Report

A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-3

Casts from a base class to a derived class should not be performed on polymorphic types

Description

Rule Definition

Casts from a base class to a derived class should not be performed on polymorphic types.

Message in Report

Casts from a base class to a derived class should not be performed on polymorphic types.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-4

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used

Description

Rule Definition

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Message in Report

C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-5

A cast shall not remove any const or volatile qualification from the type of a pointer or reference

Description

Rule Definition

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Message in Report

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-6

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type

Description

Rule Definition

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Message in Report

A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-7

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly

Description

Rule Definition

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Polyspace Implementation

The checker flags all pointer conversions including between a pointer to a `struct` object and a pointer to the first member of the same `struct` type.

Indirect conversions from a pointer to non-pointer type are not detected.

Message in Report

An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-8

An object with integer type or pointer to void type shall not be converted to an object with pointer type

Description

Rule Definition

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Polyspace Implementation

The checker allows an exception on zero constants.

Objects with pointer type include objects with pointer-to-function type.

Message in Report

An object with integer type or pointer to void type shall not be converted to an object with pointer type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-9

A cast should not convert a pointer type to an integral type

Description

Rule Definition

A cast should not convert a pointer type to an integral type.

Message in Report

A cast should not convert a pointer type to an integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-10

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression

Description

Rule Definition

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Message in Report

The increment (++) and decrement (--) operators should not be mixed with other operators in an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-11

The comma operator, && operator and the || operator shall not be overloaded

Description

Rule Definition

The comma operator, && operator and the || operator shall not be overloaded.

Message in Report

The comma operator, && operator and the || operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-2-12

An identifier with array type passed as a function argument shall not decay to a pointer

Description

Rule Definition

An identifier with array type passed as a function argument shall not decay to a pointer.

Message in Report

An identifier with array type passed as a function argument shall not decay to a pointer.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-3-1

Each operand of the ! operator, the logical && or the logical || operators shall have type bool

Description

Rule Definition

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Message in Report

Each operand of the ! operator, the logical && or the logical || operators shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-3-2

The unary minus operator shall not be applied to an expression whose underlying type is unsigned

Description

Rule Definition

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Message in Report

The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-3-3

The unary & operator shall not be overloaded

Description

Rule Definition

The unary & operator shall not be overloaded.

Message in Report

The unary & operator shall not be overloaded.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-3-4

Evaluation of the operand to the sizeof operator shall not contain side effects

Description

Rule Definition

Evaluation of the operand to the sizeof operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls

Message in Report

Evaluation of the operand to the sizeof operator shall not contain side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-8-1

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand

Description

Rule Definition

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Message in Report

The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-14-1

The right hand operand of a logical && or || operator shall not contain side effects

Description

Rule Definition

The right hand operand of a logical && or || operator shall not contain side effects.

Polyspace Implementation

The checker does not show a warning on volatile accesses and function calls.

Message in Report

The right hand operand of a logical && or || operator shall not contain side effects.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-18-1

The comma operator shall not be used

Description

Rule Definition

The comma operator shall not be used.

Message in Report

The comma operator shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 5-19-1

Evaluation of constant unsigned integer expressions should not lead to wrap-around

Description

Rule Definition

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Message in Report

Evaluation of constant unsigned integer expressions should not lead to wrap-around.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Expressions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-2-1

Assignment operators shall not be used in sub-expressions

Description

Rule Definition

Assignment operators shall not be used in sub-expressions.

Message in Report

Assignment operators shall not be used in sub-expressions.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-2-2

Floating-point expressions shall not be directly or indirectly tested for equality or inequality

Description

Rule Definition

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Polyspace Implementation

The checker detects the use of == or != with floating-point variables or expressions. The checker does not detect indirectly testing of equality, for instance, using the <= operator.

Message in Report

Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-2-3

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character

Description

Rule Definition

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.

Polyspace Implementation

The checker considers a null statement as a line where the first character excluding comments is a semicolon. The checker flags situations where:

- Comments appear before the semicolon.

  For instance:

  ```
  /* wait for pin */ ;
  ```
- Comments appear immediately after the semicolon without a white space in between.

  For instance:

  ```
  ;// wait for pin
  ```

The checker also shows a violation when a second statement appears on the same line following the null statement.

For instance:

```
; count++;
```

Message in Report

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white - space character.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-3-1

The statement forming the body of a switch, while, do while or for statement shall be a compound statement

Description

Rule Definition

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Message in Report

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-1

An if (condition) construct shall be followed by a compound statement The else keyword shall be followed by either a compound statement, or another if statement

Description

Rule Definition

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Message in Report

An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-2

All if else if constructs shall be terminated with an else clause

Description

Rule Definition

All if … else if constructs shall be terminated with an else clause.

Message in Report

All if … else if constructs shall be terminated with an else clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-3

A switch statement shall be a well-formed switch statement

Description

Rule Definition

A switch statement shall be a well-formed switch statement.

Polyspace Implementation

The checker flags these situations:

- A statement occurs between the `switch` statement and the first `case` statement.

  For instance:

  ```
  switch(ch) {
    int temp;
    case 1:
      break;
    default:
      break;
  }
  ```
- A label or a jump statement such as `goto` or `return` occurs in the `switch` block.
- A variable is declared in a `case` statement (outside any block).

  For instance:

  ```
  switch(ch) {
    case 1:
      int temp;
      break;
    default:
      break;
  }
  ```

Message in Report

A switch statement shall be a well-formed switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-4

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement

Description

Rule Definition

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Message in Report

A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-5

An unconditional throw or break statement shall terminate every non - empty switch-clause

Description

Rule Definition

An unconditional throw or break statement shall terminate every non - empty switch-clause.

Message in Report

An unconditional throw or break statement shall terminate every non - empty switch-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-6

The final clause of a switch statement shall be the default-clause

Description

Rule Definition

The final clause of a switch statement shall be the default-clause.

Polyspace Implementation

The checker detects `switch` statements that do not have a final `default` clause.

The checker does not raise a violation if the `switch` variable is an `enum` with finite number of values and you have a `case` clause for each value. For instance:

```
enum Colours { RED, BLUE, GREEN } colour;

switch ( colour ) {
      case RED:
        break;
      case BLUE:
        break;
      case GREEN:
        break;
}
```

Message in Report

The final clause of a switch statement shall be the default-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-7

The condition of a switch statement shall not have bool type

Description

Rule Definition

The condition of a switch statement shall not have bool type.

Message in Report

The condition of a switch statement shall not have bool type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-4-8

Every switch statement shall have at least one case-clause

Description

Rule Definition

Every switch statement shall have at least one case-clause.

Message in Report

Every switch statement shall have at least one case-clause.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-1

A for loop shall contain a single loop-counter which shall not have floating type

Description

Rule Definition

A for loop shall contain a single loop-counter which shall not have floating type.

Polyspace Implementation

The checker flags these situations:

- The `for` loop index has a floating point type.
- More than one loop counter is incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0, j=0; i<10 && j < 10;i++, j++) {}
  ```
- A loop counter is not incremented in the `for` loop increment statement.

  For instance:

  ```
  for(i=0; i<10;) {}
  ```

  Even if you increment the loop counter in the loop body, the checker still raises a violation. According to the MISRA C++ specifications, a loop counter is one that is initialized in or prior to the loop expression, acts as an operand to a relational operator in the loop expression and *is modified in the loop expression*. If the increment statement in the loop expression is missing, the checker cannot find the loop counter modification and considers as if a loop counter is not present.

Message in Report

A for loop shall contain a single loop-counter which shall not have floating type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-2

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=

Description

Rule Definition

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Message in Report

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-3

The loop-counter shall not be modified within condition or statement

Description

Rule Definition

The loop-counter shall not be modified within condition or statement.

Rationale

The `for` loop has a specific syntax for modifying the loop counter. A code reviewer expects modification using that syntax. Modifying the loop counter elsewhere can make the code harder to review.

Polyspace Implementation

The checker flags modification of a `for` loop counter in the loop body or the loop condition (the condition that is checked to see if the loop must be terminated).

Message in Report

The loop-counter shall not be modified within condition or statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-4

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop

Description

Rule Definition

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

Message in Report

The loop-counter shall be modified by one of: --, ++, -=n, or +=n ; where n remains constant for the duration of the loop.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-5

A loop-control-variable other than the loop-counter shall not be modified within condition or expression

Description

Rule Definition

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Message in Report

A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-5-6

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool

Description

Rule Definition

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Message in Report

A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-6-1

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement

Description

Rule Definition

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Message in Report

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-6-2

The goto statement shall jump to a label declared later in the same function body

Description

Rule Definition

The goto statement shall jump to a label declared later in the same function body.

Message in Report

The goto statement shall jump to a label declared later in the same function body.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-6-3

The continue statement shall only be used within a well-formed for loop

Description

Rule Definition

The continue statement shall only be used within a well-formed for loop.

Polyspace Implementation

The checker flags the use of `continue` statements in:

- `for` loops that are not well-formed, that is, loops that violate rules 6-5-x.
- `while` loops.

Message in Report

The continue statement shall only be used within a well-formed for loop.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-6-4

For any iteration statement there shall be no more than one break or goto statement used for loop termination

Description

Rule Definition

For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Message in Report

For any iteration statement there shall be no more than one break or goto statement used for loop termination.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 6-6-5

A function shall have a single point of exit at the end of the function

Description

Rule Definition

A function shall have a single point of exit at the end of the function.

Rationale

This rule requires that a `return` statement must occur as the last statement in the function body. Otherwise, the following issues can occur:

- Code following a `return` statement can be unintentionally omitted.
- If a function that modifies some of its arguments has early `return` statements, when reading the code, it is not immediately clear which modifications actually occur.

Polyspace Implementation

The checker flags these situations:

- A function has more than one `return` statement.
- A non-`void` function has one `return` statement only but the `return` statement is not the last statement in the function.

A `void` function need not have a `return` statement. If a `return` statement exists, it need not be the last statement in the function.

Message in Report

A function shall have a single point of exit at the end of the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Statements
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-1-1

A variable which is not modified shall be const qualified

Description

Rule Definition

A variable which is not modified shall be const qualified.

Polyspace Implementation

The checker flags function parameters or local variables that are not const-qualified but never modified in the function body. Function parameters of integer, float, enum and boolean types are not flagged.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. These variables are not flagged.

Message in Report

A variable which is not modified shall be const qualified.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 7-1-2

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified

Description

Rule Definition

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Polyspace Implementation

The checker flags pointers where the underlying object is not const-qualified but never modified in the function body.

If a variable is passed to another function by reference or pointers, the checker assumes that the variable can be modified. Pointers that point to these variables are not flagged.

Message in Report

A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 7-3-1

The global namespace shall only contain main, namespace declarations and extern "C" declarations

Description

Rule Definition

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Message in Report

The global namespace shall only contain main, namespace declarations and extern "C" declarations.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-3-2

The identifier main shall not be used for a function other than the global function main

Description

Rule Definition

The identifier main shall not be used for a function other than the global function main.

Message in Report

The identifier main shall not be used for a function other than the global function main.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-3-3

There shall be no unnamed namespaces in header files

Description

Rule Definition

There shall be no unnamed namespaces in header files.

Message in Report

There shall be no unnamed namespaces in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-3-4

using-directives shall not be used

Description

Rule Definition

using-directives shall not be used.

Message in Report

using-directives shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-3-5

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier

Description

Rule Definition

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Message in Report

Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-3-6

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files

Description

Rule Definition

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Message in Report

using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-4-2

Assembler instructions shall only be introduced using the asm declaration

Description

Rule Definition

Assembler instructions shall only be introduced using the asm declaration.

Message in Report

Assembler instructions shall only be introduced using the asm declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-4-3

Assembly language shall be encapsulated and isolated

Description

Rule Definition

Assembly language shall be encapsulated and isolated.

Polyspace Implementation

The checker flags `asm` statements unless they are encapsulated in a function call.

For instance, the noncompliant `asm` statement below is in regular C code while the compliant `asm` statement is encapsulated in a call to the function `Delay`.

```
void Delay ( void )
  {
    asm( "NOP");//Compliant
  }
void fn (void)
  {
    DoSomething();
    Delay();// Assembler is encapsulated
    DoSomething();
    asm("NOP"); //Noncompliant
    DoSomething();
  }
```

Message in Report

Assembly language shall be encapsulated and isolated.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-5-1

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function

Description

Rule Definition

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Message in Report

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-5-2

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

Description

Rule Definition

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Polyspace Implementation

The checker flags situations where the address of a local variable is assigned to a pointer defined at global scope.

The checker does not raise violations of this rule if :

- A function returns the address of a local variable. This situation is covered by `MISRA C++:2008 Rule 7-5-1`.
- The address of a variable defined at block scope is assigned to a pointer that is defined with greater scope (but not global scope).

  For instance:

  ```
  void foobar ( void )
   {
     char * ptr;
     {
       char var;
       ptr = &var;
     }
   }
  ```

  Only if the pointer is defined at global scope is the issue detected. For instance, the rule checker flags the issue here:

```
char * ptr;
void foobar ( void )
  {
      char var;
      ptr = &var;
  }
```

Message in Report

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-5-3

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference

Description

Rule Definition

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Message in Report

A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 7-5-4

Functions should not call themselves, either directly or indirectly

Description

Rule Definition

Functions should not call themselves, either directly or indirectly.

Message in Report

Functions should not call themselves, either directly or indirectly.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarations
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-0-1

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively

Description

Rule Definition

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Message in Report

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-3-1

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments

Description

Rule Definition

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Message in Report

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-4-1

Functions shall not be defined using the ellipsis notation

Description

Rule Definition

Functions shall not be defined using the ellipsis notation.

Message in Report

Functions shall not be defined using the ellipsis notation.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-4-2

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration

Description

Rule Definition

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Polyspace Implementation

The checker detects mismatch in parameter names between:

- A function declaration and the corresponding definition.
- Two declarations of a function, provided they occur in the same file.

  If the declarations occur in different files, the checker does not raise a violation for mismatch in parameter names. Redeclarations in different files are forbidden by `MISRA C++:2008 Rule 3-2-3`.

Message in Report

The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-4-3

All exit paths from a function with non- void return type shall have an explicit return statement with an expression

Description

Rule Definition

All exit paths from a function with non- void return type shall have an explicit return statement with an expression.

Message in Report

All exit paths from a function with non- void return type shall have an explicit return statement with an expression.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-4-4

A function identifier shall either be used to call the function or it shall be preceded by &

Description

Rule Definition

A function identifier shall either be used to call the function or it shall be preceded by &.

Message in Report

A function identifier shall either be used to call the function or it shall be preceded by &.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-5-1

All variables shall have a defined value before they are used

Description

Rule Definition

All variables shall have a defined value before they are used.

Message in Report

All variables shall have a defined value before they are used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-5-2

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures

Description

Rule Definition

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.

Message in Report

Braces shall be used to indicate and match the structure in the non- zero initialization of arrays and structures.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 8-5-3

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized

Description

Rule Definition

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Message in Report

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Declarators
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-3-1

const member functions shall not return non-const pointers or references to class-data

Description

Rule Definition

const member functions shall not return non-const pointers or references to class-data.

Polyspace Implementation

The checker flags a rule violation only if a `const` member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Message in Report

const member functions shall not return non-const pointers or references to class-data.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-3-2

Member functions shall not return non-const handles to class-data

Description

Rule Definition

Member functions shall not return non-const handles to class-data.

Polyspace Implementation

The checker flags a rule violation only if a member function returns a non-`const` pointer or reference to a nonstatic data member. The rule does not apply to static data members.

Message in Report

Member functions shall not return non-const handles to class-data.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-3-3

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const

Description

Rule Definition

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Polyspace Implementation

The checker flags member functions that are not declared static but do not access a data member of the class. Such a function can be potentially declared static.

The checker flags member functions that are not declared const but do not modify a data member of the class. Such a function can be potentially declared const.

Message in Report

If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 9-5-1

Unions shall not be used

Description

Rule Definition

Unions shall not be used.

Message in Report

Unions shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-6-2

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type

Description

Rule Definition

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Message in Report

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-6-3

Bit-fields shall not have enum type

Description

Rule Definition

Bit-fields shall not have enum type.

Message in Report

Bit-fields shall not have enum type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 9-6-4

Named bit-fields with signed integer type shall have a length of more than one bit

Description

Rule Definition

Named bit-fields with signed integer type shall have a length of more than one bit.

Message in Report

Named bit-fields with signed integer type shall have a length of more than one bit.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-1-1

Classes should not be derived from virtual bases

Description

Rule Definition

Classes should not be derived from virtual bases.

Rationale

The use of virtual bases can lead to many confusing behaviors.

For instance, in an inheritance hierarchy involving a virtual base, the most derived class calls the constructor of the virtual base. Intermediate calls to the virtual base constructor are ignored.

Message in Report

Classes should not be derived from virtual bases.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of Virtual Bases

```
class Base {};
class Intermediate: public virtual Base {}; //Noncompliant
class Final: public Intermediate {};
```

In this example, the rule checker raises a violation when the `Intermediate` class is derived from the class `Base` with the `virtual` keyword.

The following behavior can be a potential source of confusion. When you create an object of type `Final`, the constructor of `Final` directly calls the constructor of `Base`. Any call to the `Base` constructor from the `Intermediate` constructor are ignored. You might see unexpected results if you do not take into account this behavior.

Check Information

Group: Derived Classes
Category: Advisory

See Also

`MISRA C++:2008 Rule 10-1-2`

Introduced in R2013b

MISRA C++:2008 Rule 10-1-2

A base class shall only be declared virtual if it is used in a diamond hierarchy

Description

Rule Definition

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Rationale

This rule is less restrictive than `MISRA C++:2008 Rule 10-1-1`. Rule 10-1-1 forbids the use of a virtual base anywhere in your code because a virtual base can lead to potentially confusing behavior.

Rule 10-1-2 allows the use of virtual bases in the one situation where they are useful, that is, as a common base class in diamond hierarchies.

For instance, the following diamond hierarchy violates rule 10-1-1 but not rule 10-1-2.

```
class Base {};
class Intermediate1: public virtual Base {};
class Intermediate2: public virtual Base {};
class Final: public Intermediate1, public Intermediate2 {};
```

Message in Report

A base class shall only be declared virtual if it is used in a diamond hierarchy.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-1-3

An accessible base class shall not be both virtual and non-virtual in the same hierarchy

Description

Rule Definition

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Rationale

The checker flags situations where the same class is inherited as a virtual base class and a non-virtual base class in the same derived class. These situations defeat the purpose of virtual inheritance and causes multiple copies of the base class sub-object in the derived class object.

Message in Report

An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Base Class Both Virtual and Non-Virtual in Same Hierarchy

```
class Base {};
class Intermediate1: virtual public Base {};
class Intermediate2: virtual public Base {};
```

```
class Intermediate3: public Base {};
class Final: public Intermediate1, Intermediate2, Intermediate3 {}; //Noncompliant
```

In this example, the class `Base` is inherited in `Final` both as a virtual and non-virtual base class. The `Final` object contains at least two copies of a `Base` sub-object.

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-2-1

All accessible entity names within a multiple inheritance hierarchy should be unique

Description

Rule Definition

All accessible entity names within a multiple inheritance hierarchy should be unique.

Polyspace Implementation

The checker flags data members from different classes with conflicting names if the same class derives from these classes. For instance:

```
class B1
  {
    public:
      int count;
      void foo ( );
  };
class B2
  {
    public:
      int count;
      void foo ( );
  };

class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++B1::count;
        B1::foo ( );
      }
  };
```

If the data member access in the derived class is ambiguous, the analysis reports this issue as a compilation error and not a coding rule violation. For instance, a compilation error occurs in the preceding example if the class D is rewritten as:

```
class D : public B1, public B2
  {
    public:
      void Bar ( )
      {
        ++count;      // Is that B1::count or B2::count?
        foo ( );      // Is that B1::foo() or B2::foo()?
      }
  };
```

The checker does not check for conflicts between entities of different kinds, for instance, member functions against data members.

Message in Report

All accessible entity names within a multiple inheritance hierarchy should be unique.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-3-1

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy

Description

Rule Definition

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Rationale

The checker flags virtual member functions that have multiple definitions on the same path in an inheritance hierarchy. If a function is defined multiple times, it can be ambiguous which implementation is used in a given function call.

Polyspace Implementation

The checker also raises a violation if a base class member function is redefined in the derived class without the `virtual` keyword.

Message in Report

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Virtual Function Redefined in Derived Class

```
class Base {
    public:
      virtual void foo() {
     }
};

class Intermediate1: public virtual Base {
    public:
      virtual void foo() { //Noncompliant
      }
};

class Intermediate2: public virtual Base {
    public:
       void bar() {
          foo();  // Calls Base::foo()
      }
};

class Final: public Intermediate1, public Intermediate2 {
};

void main() {
    Intermediate2 intermediate2Obj;
    intermediate2Obj.bar(); // Calls Base::foo()
    Final finalObj;
    finalObj.bar(); //Calls Intermediate1::foo()
                    //but you might expect Base::foo()
}
```

In this example, the `virtual` function `foo` is defined in the base class `Base` and also in the derived class `Intermediate1`.

A potential source of confusion can be the following. The class `Final` derives from `Intermediate1` and also derives from `Base` through another path using `Intermediate2`.

- When an `Intermediate2` object calls the function `bar` that calls the function `foo`, the implementation of `foo` in `Base` is called. An `Intermediate2` object does not know of the implementation in `Intermediate1`.
- However, when a `Final` object calls the same function `bar` that calls the function `foo`, the implementation of `foo` in `Intermediate1` is called because of dominance of the more derived class.

You might see unexpected results if you do not take this behavior into account.

To prevent this issue, declare a function as pure virtual in the base class. For instance, you can declare the class `Base` as follows:

```
class Base {
    public:
      virtual void foo()=0;
};

void Base::foo() {
      //You can still define Base::foo()
}
```

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-3-2

Each overriding virtual function shall be declared with the virtual keyword

Description

Rule Definition

Each overriding virtual function shall be declared with the virtual keyword.

Message in Report

Each overriding virtual function shall be declared with the virtual keyword.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 10-3-3

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual

Description

Rule Definition

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Message in Report

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Derived Classes
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 11-0-1

Member data in non- POD class types shall be private

Description

Rule Definition

Member data in non- POD class types shall be private.

Message in Report

Member data in non- POD class types shall be private.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Member Access Control
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 12-1-1

An object's dynamic type shall not be used from the body of its constructor or destructor

Description

Rule Definition

An object's dynamic type shall not be used from the body of its constructor or destructor.

Message in Report

An object's dynamic type shall not be used from the body of its constructor or destructor.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Special Member Functions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 12-1-2

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes

Description

Rule Definition

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Message in Report

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Special Member Functions
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 12-1-3

All constructors that are callable with a single argument of fundamental type shall be declared explicit

Description

Rule Definition

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Message in Report

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Special Member Functions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 12-8-1

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member

Description

Rule Definition

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.

Message in Report

A copy constructor shall only initialize its base classes and the non- static members of the class of which it is a member.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Special Member Functions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 12-8-2

The copy assignment operator shall be declared protected or private in an abstract class

Description

Rule Definition

The copy assignment operator shall be declared protected or private in an abstract class.

Message in Report

The copy assignment operator shall be declared protected or private in an abstract class.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Special Member Functions
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-5-2

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter

Description

Rule Definition

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Message in Report

A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-5-3

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter

Description

Rule Definition

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Message in Report

A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-6-1

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Description

Rule Definition

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Message in Report

In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-6-2

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit

Description

Rule Definition

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Message in Report

The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-7-3

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template

Description

Rule Definition

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Message in Report

All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-8-1

Overloaded function templates shall not be explicitly specialized

Description

Rule Definition

Overloaded function templates shall not be explicitly specialized.

Polyspace Implementation

The checker first checks within file scope to find overloads. The checker later looks for call to a specialized template function later. As a result, the checker flags all specializations of overloaded templates even if overloading occurs after the call.

Message in Report

Overloaded function templates shall not be explicitly specialized.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 14-8-2

The viable function set for a function call should either contain no function specializations, or only contain function specializations

Description

Rule Definition

The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Message in Report

The viable function set for a function call should either contain no function specializations, or only contain function specializations.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Templates
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-0-2

An exception object should not have pointer type

Description

Rule Definition

An exception object should not have pointer type.

Polyspace Implementation

The checker raises a violation if a `throw` statement throws an exception of pointer type.

The checker does not raise a violation if a NULL pointer is thrown as exception. Throwing a NULL pointer is forbidden by `MISRA C++:2008 Rule 15-1-2`.

Message in Report

An exception object should not have pointer type.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-0-3

Control shall not be transferred into a try or catch block using a goto or a switch statement

Description

Rule Definition

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Message in Report

Control shall not be transferred into a try or catch block using a goto or a switch statement.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-1-2

NULL shall not be thrown explicitly

Description

Rule Definition

NULL shall not be thrown explicitly.

Message in Report

NULL shall not be thrown explicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-1-3

An empty throw (throw;) shall only be used in the compound- statement of a catch handler

Description

Rule Definition

An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

Message in Report

An empty throw (throw;) shall only be used in the compound- statement of a catch handler.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-3-2

There should be at least one exception handler to catch all otherwise unhandled exceptions

Description

Rule Definition

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Polyspace Implementation

The checker shows a violation if there is no `try/catch` in the `main` function or the `catch` does not handle all exceptions (with ellipsis `...`). The rule is not checked if a `main` function does not exist.

The checker does not determine if an exception of an unhandled type actually propagates to `main`.

Bug Finder and Code Prover check this coding rule differently. The analyses can produce different results.

Message in Report

There should be at least one exception handler to catch all otherwise unhandled exceptions.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-3-3

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases

Description

Rule Definition

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Message in Report

Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-3-5

A class type exception shall always be caught by reference

Description

Rule Definition

A class type exception shall always be caught by reference.

Message in Report

A class type exception shall always be caught by reference.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-3-6

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Message in Report

Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-3-7

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last

Description

Rule Definition

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Message in Report

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-4-1

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids

Description

Rule Definition

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Message in Report

If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-5-1

A class destructor shall not exit with an exception

Description

Rule Definition

A class destructor shall not exit with an exception.

Polyspace Implementation

The checker flags exceptions thrown in the body of the destructor. If the destructor calls another function, the checker does not detect if that function throws an exception.

The checker does not detect these situations:

- A `catch` statement does not catch exceptions of all types that are thrown.

  The checker considers the presence of a `catch` statement corresponding to a `try` block as indication that an exception is caught.
- `throw` statements inside `catch` blocks

Message in Report

A class destructor shall not exit with an exception.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling

Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-5-2

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)

Description

Rule Definition

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Polyspace Implementation

The checker flags situations where the data type of the exception thrown does not match the exception type listed in the function specification.

For instance:

```
void goo ( ) throw ( Exception )
  {
    throw 21; // Non-compliant - int is not listed
  }
```

The checker limits detection to `throw` statements that are in the body of the function. If the function calls another function, the checker does not detect if the called function throws an exception.

The checker does not detect `throw` statements inside `catch` blocks.

Message in Report

Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 15-5-3

The terminate() function shall not be called implicitly

Description

Rule Definition

The terminate() function shall not be called implicitly.

Polyspace Implementation

The checker flags these situations when the terminate() function can be called implicitly:

- An exception escapes uncaught. This also violates `MISRA C++:2008 Rule 15-3-2`. For instance:
  - Before an exception is caught, it escapes through another function that throws an uncaught exception. For instance, a catch statement or exception handler invokes a copy constructor that throws an uncaught exception.
  - A throw expression with no operand rethrows an uncaught exception.
- A class destructor throws an exception. This also violates `MISRA C++:2008 Rule 15-5-1`.

Message in Report

The terminate() function shall not be called implicitly.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Exception Handling
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 16-0-1

#include directives in a file shall only be preceded by other preprocessor directives or comments

Description

Rule Definition

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Message in Report

#include directives in a file shall only be preceded by other preprocessor directives or comments.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-2

Macros shall only be #define 'd or #undef 'd in the global namespace

Description

Rule Definition

Macros shall only be #define 'd or #undef 'd in the global namespace.

Message in Report

Macros shall only be #define 'd or #undef 'd in the global namespace.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-3

#undef shall not be used

Description

Rule Definition

#undef shall not be used.

Message in Report

#undef shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-4

Function-like macros shall not be defined

Description

Rule Definition

Function-like macros shall not be defined.

Message in Report

Function-like macros shall not be defined.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-5

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives

Description

Rule Definition

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Message in Report

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-6

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##

Description

Rule Definition

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Message in Report

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-7

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator

Description

Rule Definition

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Message in Report

Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-0-8

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token

Description

Rule Definition

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Message in Report

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-1-1

The defined preprocessor operator shall only be used in one of the two standard forms

Description

Rule Definition

The defined preprocessor operator shall only be used in one of the two standard forms.

Message in Report

The defined preprocessor operator shall only be used in one of the two standard forms.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-1-2

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related

Description

Rule Definition

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Message in Report

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-1

The preprocessor shall only be used for file inclusion and include guards

Description

Rule Definition

The preprocessor shall only be used for file inclusion and include guards.

Polyspace Implementation

The checker flags `#ifdef` and `#define` statements in files that are not include files.

Message in Report

The preprocessor shall only be used for file inclusion and include guards.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-2

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers

Description

Rule Definition

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Polyspace Implementation

The checker flags `#define` statements where the macros expand to something other than include guards, type qualifiers or storage class specifiers such as `static`, `inline`, `volatile`, `auto`, `register` and `const`.

Message in Report

C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-3

Include guards shall be provided

Description

Rule Definition

Include guards shall be provided.

Polyspace Implementation

The checker raises a violation if a header file does not contain an include guard.

For instance, this code uses an include guard for the `#define` and `#include` statements and does not violate the rule:

```
// Contents of a header file
#ifndef FILE_H

#define FILE_H
#include "libFile.h"

#endif
```

Message in Report

Include guards shall be provided.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-4

The ', ", /* or // characters shall not occur in a header file name

Description

Rule Definition

The ', ", / or // characters shall not occur in a header file name.*

Message in Report

The ', ", /* or // characters shall not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-5

The \ character should not occur in a header file name

Description

Rule Definition

The \ character should not occur in a header file name.

Message in Report

The \ character should not occur in a header file name.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-2-6

The #include directive shall be followed by either a <filename> or "filename" sequence

Description

Rule Definition

The #include directive shall be followed by either a <filename> or "filename" sequence.

Message in Report

The #include directive shall be followed by either a <filename> or "filename" sequence.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-3-1

There shall be at most one occurrence of the # or ## operators in a single macro definition

Description

Rule Definition

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Message in Report

There shall be at most one occurrence of the # or ## operators in a single macro definition.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-3-2

The # and ## operators should not be used

Description

Rule Definition

The # and ## operators should not be used.

Message in Report

The # and ## operators should not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Advisory

See Also

Introduced in R2013b

MISRA C++:2008 Rule 16-6-1

All uses of the #pragma directive shall be documented

Description

Rule Definition

All uses of the #pragma directive shall be documented.

Polyspace Implementation

To check this rule, you must list the pragmas that are allowed in source files by using the option `Allowed pragmas (-allowed-pragmas)`. If Polyspace finds a pragma not in the allowed pragma list, a violation is raised. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Message in Report

All uses of the #pragma directive shall be documented.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Preprocessing Directives
Category: Document

See Also

Introduced in R2016b

MISRA C++:2008 Rule 17-0-1

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined

Description

Rule Definition

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Message in Report

Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Library Introduction
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 17-0-2

The names of standard library macros and objects shall not be reused

Description

Rule Definition

The names of standard library macros and objects shall not be reused.

Message in Report

The names of standard library macros and objects shall not be reused.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Library Introduction
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 17-0-3

The names of standard library functions shall not be overridden

Description

Rule Definition

The names of standard library functions shall not be overridden.

Message in Report

The names of standard library functions shall not be overridden.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Library Introduction
Category: Required

See Also

Introduced in R2018a

MISRA C++:2008 Rule 17-0-5

The setjmp macro and the longjmp function shall not be used

Description

Rule Definition

The setjmp macro and the longjmp function shall not be used.

Message in Report

The setjmp macro and the longjmp function shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Library Introduction
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-0-1

The C library shall not be used

Description

Rule Definition

The C library shall not be used.

Message in Report

The C library shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-0-2

The library functions atof, atoi and atol from library <cstdlib> shall not be used

Description

Rule Definition

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Message in Report

The library functions atof, atoi and atol from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-0-3

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used

Description

Rule Definition

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Message in Report

The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-0-4

The time handling functions of library <ctime> shall not be used

Description

Rule Definition

The time handling functions of library <ctime> shall not be used.

Message in Report

The time handling functions of library <ctime> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-0-5

The unbounded functions of library <cstring> shall not be used

Description

Rule Definition

The unbounded functions of library <cstring> shall not be used.

Message in Report

The unbounded functions of library <cstring> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-2-1

The macro offsetof shall not be used

Description

Rule Definition

The macro offsetof shall not be used.

Message in Report

The macro offsetof shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-4-1

Dynamic heap memory allocation shall not be used

Description

Rule Definition

Dynamic heap memory allocation shall not be used.

Message in Report

Dynamic heap memory allocation shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 18-7-1

The signal handling facilities of <csignal> shall not be used

Description

Rule Definition

The signal handling facilities of <csignal> shall not be used.

Rationale

Signal handling functions such as `signal` contains undefined and implementation-specific behavior.

You have to be very careful when using `signal` to avoid these behaviors.

Message in Report

The signal handling facilities of <csignal> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Check Information

Group: Language Support Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 19-3-1

The error indicator errno shall not be used

Description

Rule Definition

The error indicator errno shall not be used.

Rationale

Observing this rule encourages the good practice of not relying on `errno` to check error conditions.

Checking `errno` is not sufficient to guarantee absence of errors. Functions such as `fopen` might not set `errno` on error conditions. Often, you have to check the return value of such functions for error conditions.

Message in Report

The error indicator errno shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of `errno`

```
#include <cstdlib>
#include <cerrno>
```

```
void func (const char* str) {
    errno = 0;  // Noncompliant
    int i = atoi(str);
    if(errno != 0) { // Noncompliant
        //Handle Error
    }
}
```

The use of `errno` violates this rule. The function `atoi` is not required to set `errno` if the input string cannot be converted to an integer. Checking `errno` later does not safeguard against possible failures in conversion.

Check Information

Group: Diagnostic Library
Category: Required

See Also

Introduced in R2013b

MISRA C++:2008 Rule 27-0-1

The stream input/output library <cstdio> shall not be used

Description

Rule Definition

The stream input/output library <cstdio> shall not be used.

Rationale

Functions in `cstdio` such as `gets`, `fgetpos`, `fopen`, `ftell`, etc. have unspecified, undefined and implementation-defined behavior.

For instance:

- The `gets` function:

  ```
  char * gets ( char * buf );
  ```

  does not check if the number of characters provided at the standard input exceeds the buffer `buf`. The function can have unexpected behavior when the input exceeds the buffer.
- The `fopen` function has implementation-specific behavior related to whether it sets `errno` on errors or whether it accepts additional characters following the standard mode specifiers.

Message in Report

The stream input/output library <cstdio> shall not be used.

Troubleshooting

If you expect a rule violation but do not see it, refer to The documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Use of gets

```
#include <cstdio>

void func() {
 char array[10];
 gets(array);
}
```

The use of `gets` violates this rule.

Check Information

Group: Input/output Library
Category: Required

See Also

Introduced in R2013b

Custom Coding Rules

Group 1: Files

The custom rules 1.*x* in Polyspace enforce naming conventions for files and folders. For information on how to enable these rules, see .

Number	Rule Applied	Other details
1.1	All source file names must follow the specified pattern.	Only the base name is checked. A source file is a file that is not included.
1.2	All source folder names must follow the specified pattern.	Only the folder name is checked. A source file is a file that is not included.
1.3	All include file names must follow the specified pattern.	Only the base name is checked. An include file is a file that is included.
1.4	All include folder names must follow the specified pattern.	Only the folder name is checked. An include file is a file that is included.

Group 2: Preprocessing

The custom rules 2.*x* in Polyspace enforce naming conventions for macros. For information on how to enable these rules, see .

Number	Rule Applied	Other details
2.1	All macros must follow the specified pattern.	Macro names are checked before preprocessing.
2.2	All macro parameters must follow the specified pattern.	Macro parameters are checked before preprocessing.

Group 3: Type definitions

The custom rules 3.*x* in Polyspace enforce naming conventions for fundamental data types. For information on how to enable these rules, see .

Number	Rule Applied	Other details
3.1	All integer types must follow the specified pattern.	Applies to integer types specified by `typedef` statements. Does not apply to enumeration types. For example: `typedef signed int int32_t;`
3.2	All float types must follow the specified pattern.	Applies to float types specified by `typedef` statements. For example: `typedef float f32_t;`
3.3	All pointer types must follow the specified pattern.	Applies to pointer types specified by `typedef` statements. For example: `typedef int* p_int;`
3.4	All array types must follow the specified pattern.	Applies to array types specified by `typedef` statements. For example: `typedef int[3] a_int_3;`
3.5	All function pointer types must follow the specified pattern.	Applies to function pointer types specified by `typedef` statements. For example: `typedef void (*pf_callback) (int);`

Group 4: Structures

The custom rules 4.*x* in Polyspace enforce naming conventions for structured data types. For information on how to enable these rules, see .

Number	Rule Applied	Other details
4.1	All `struct` tags must follow the specified pattern.	
4.2	All `struct` types must follow the specified pattern.	`struct` types are aliases for previously defined structures (defined with the `typedef` or `using` keyword).
4.3	All `struct` fields must follow the specified pattern.	
4.4	All `struct` bit fields must follow the specified pattern.	

Group 5: Classes (C++)

The custom rules 5.*x* in Polyspace enforce naming conventions for classes and class members. For information on how to enable these rules, see .

Number	Rule Applied	Other details
5.1	All class names must follow the specified pattern.	
5.2	All class types must follow the specified pattern.	Class types are aliases for previously defined classes (defined with the `typedef` or `using` keyword).
5.3	All data members must follow the specified pattern.	
5.4	All function members must follow the specified pattern.	
5.5	All static data members must follow the specified pattern.	
5.6	All static function members must follow the specified pattern.	
5.7	All bitfield members must follow the specified pattern.	

Group 6: Enumerations

The custom rules 6.*x* in Polyspace enforce naming conventions for enumerations. For information on how to enable these rules, see .

Number	Rule Applied	Other details
6.1	All enumeration tags must follow the specified pattern.	
6.2	All enumeration types must follow the specified pattern.	Enumeration types are aliases for previously defined enumerations (defined with the `typedef` or `using` keyword).
6.3	All enumeration constants must follow the specified pattern.	

Group 7: Functions

The custom rules 7.*x* in Polyspace enforce naming conventions for functions and function parameters. For information on how to enable these rules, see .

Number	Rule Applied	Other details
7.1	All global functions must follow the specified pattern.	A global function is a function with external linkage.
7.2	All static functions must follow the specified pattern.	A static function is a function with internal linkage.
7.3	All function parameters must follow the specified pattern.	In C++, applies to non-member functions.

Group 8: Constants

The custom rules 8.*x* in Polyspace enforce naming conventions for constants. For information on how to enable these rules, see .

Number	Rule Applied	Other details
8.1	All global constants must follow the specified pattern.	A global constant is a constant with external linkage.
8.2	All static constants must follow the specified pattern.	A static constant is a constant with internal linkage.
8.3	All local constants must follow the specified pattern.	A local constant is a constant without linkage.
8.4	All static local constants must follow the specified pattern.	A static local constant is a constant declared static in a function.

Group 9: Variables

The custom rules 9.*x* in Polyspace enforce naming conventions for variables. For information on how to enable these rules, see .

Number	Rule Applied	Other details
9.1	All global variables must follow the specified pattern.	A global variable is a variable with external linkage.
9.2	All static variables must follow the specified pattern.	A static variable is a variable with internal linkage.
9.3	All local variables must follow the specified pattern.	A local variable is a variable without linkage.
9.4	All static local variables must follow the specified pattern.	A static local variable is a variable declared static in a function.

Group 10: Name spaces (C++)

The custom rules 10.*x* in Polyspace enforce naming conventions for namespaces. For information on how to enable these rules, see .

Number	**Rule Applied**
10.1	All names spaces must follow the specified pattern.

Group 11: Class templates (C++)

The custom rules 11.*x* in Polyspace enforce naming conventions for class templates. For information on how to enable these rules, see .

Number	Rule Applied	Other details
11.1	All class templates must follow the specified pattern.	
11.2	All class template parameters must follow the specified pattern.	

Group 12: Function templates (C++)

The custom rules 12.*x* in Polyspace enforce naming conventions for function templates. For information on how to enable these rules, see .

Number	Rule Applied	Other details
12.1	All function templates must follow the specified pattern.	Applies to non-member functions.
12.2	All function template parameters must follow the specified pattern.	Applies to non-member functions.
12.3	All function template members must follow the specified pattern.	

Group 20: Style

The custom rules 20.*x* in Polyspace enforce coding style conventions such as number of characters per line. For information on how to enable these rules, see .

Number	Rule Applied	Other details
20.1	Source line length must not exceed specified number of characters.	When configuring the checker, specify: • A number for the character limit. Use the **Pattern** column on the configuration or the `pattern=` line in the custom rules text file. • A violation message such as: `Line exceeds` *`n`* `characters.` Use the **Convention** column on the configuration or the `convention=` line in the custom rules text file.

5

Global Variables

Potentially unprotected variable

Global variables shared between multiple tasks but not protected from concurrent access by the tasks

Description

A **shared unprotected global variable** has the following properties:

- The variable is used in more than one task.
- Polyspace determines that at least one operation on the variable is not protected from interruption by operations in other tasks.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored orange on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
```

```
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is an unprotected shared variable if you specify the following multitasking options:

Option	Value
`Configure multitasking manually`	

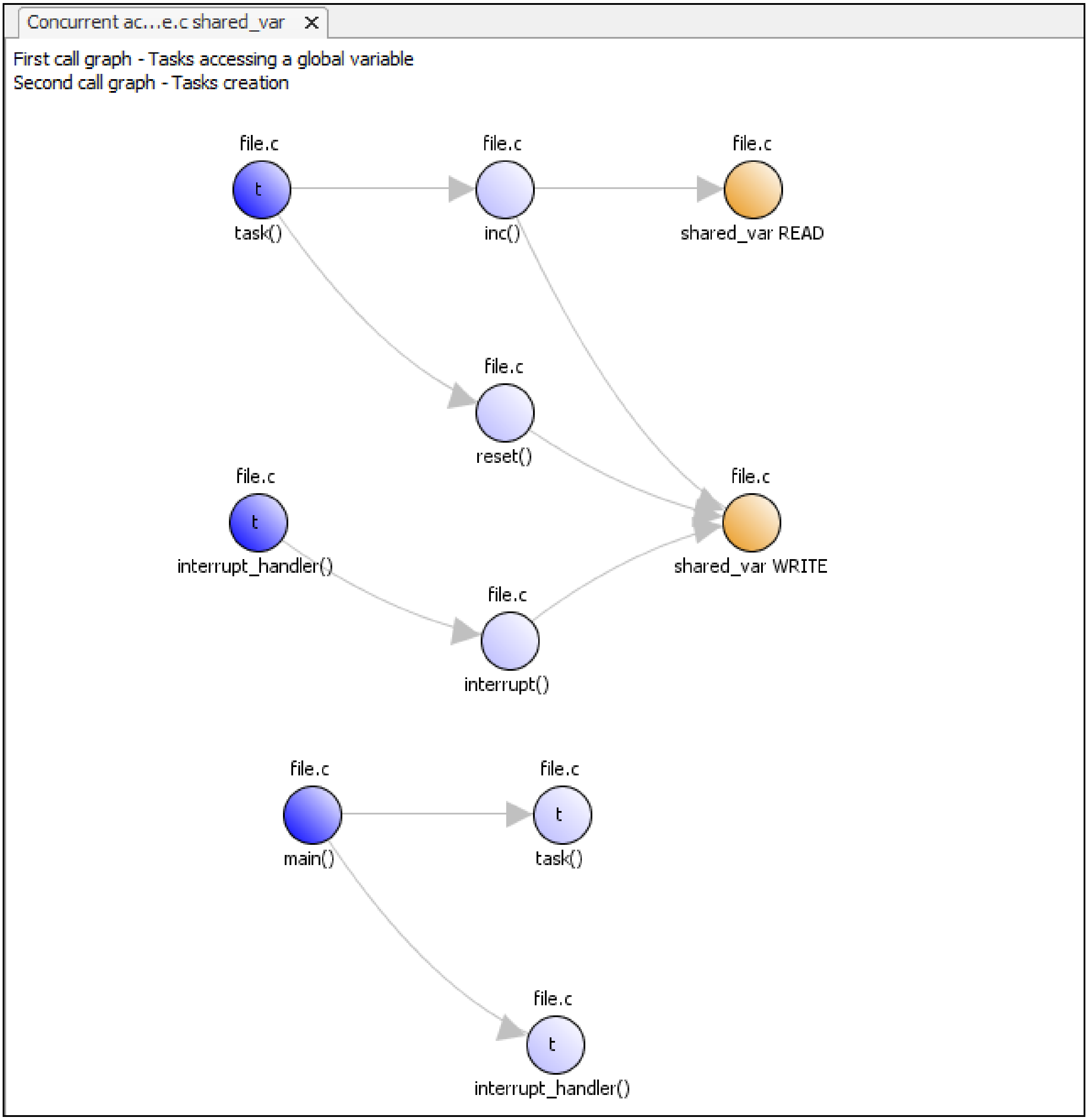
`Tasks (-entry-points)`	`task`
	
	`interrupt_handler`

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

You do not specify protection mechanisms such as critical sections.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

If you click the  (graph) icon on the **Result Details** pane, you see the two concurrent tasks (threads).


Concurrent ac…e.c shared_var
First call graph - Tasks accessing a global variable
Second call graph - Tasks creation
file.c
t
task()
file.c
inc()
file.c
shared_var READ
file.c
reset()
file.c
t
interrupt_handler()
file.c
shared_var WRITE
file.c
interrupt()
file.c
main()
file.c
t
task()
file.c
t
interrupt_handler()

The first graph shows how the tasks access the variable. For instance, the task `interrupt_handler` calls a function `interrupt` that writes to the shared variable `shared_var`.

The second graph shows how the tasks are created. In this example, both tasks are created after `main` completes. In other cases, tasks might be created within functions called from `main`.

Check Information

Language: C | C++

See Also

`Shared variable` | `Unused variable` | `Used non-shared variable`

Shared variable

Global variables shared between multiple tasks and protected from concurrent access by the tasks

Description

A **shared protected global variable** has the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusion. The calls to functions beginning and ending a critical section must be reachable.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored green on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
```

```
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following multitasking options:

Option	Value
Configure multitasking manually	☑
Tasks (-entry-points)	task interrupt_handler
Temporally exclusive tasks (-temporal-exclusions-file)	task interrupt_handler

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

On the command-line, you can use the following:

```
polyspace-code-prover
  -entry-points task,interrupt_handler
  -temporal-exclusions-file "C:\exclusions_file.txt"
```

where the file `C:\exclusions_file.txt` has the following line:

```
task interrupt_handler
```

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
```

```
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

Option	Value	
Configure multitasking manually	☑	
Tasks (-entry-points)	task interrupt_handler	
Critical section details (-critical-section-begin -critical-section-end)	**Starting routine**	**Ending routine**
	take_semaphore	give_semaphore

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

On the command-line, you can use the following:

```
 polyspace-code-prover
   -entry-points task,interrupt_handler
   -critical-section-begin take_semaphore:cs1
   -critical-section-end give_semaphore:cs1
```

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

Shared Structure Variables Protected Through Access Pattern

```
struct S {
    unsigned int var_1;
    unsigned int var_2;
};
```

```
volatile int randomVal;

struct S sharedStruct;

void task1(void) {
    while(randomVal)
        operation1();
}

void task2(void) {
    while(randomVal)
        operation2();
}

void operation1(void) {
        sharedStruct.var_1++;
}

void operation2(void) {
        sharedStruct.var_2++;
}

int main(void) {
    return 0;
}
```

In this example, `sharedStruct` is a protected shared variable if you specify the following:

Option	Value
`Configure multitasking manually`	

`Tasks (-entry-points)`	`task1`
	
	`task2`

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

On the command-line, you can use the following:

```
 polyspace-code-prover
    -entry-points task1,task2
```

The software determines that `sharedStruct` is protected because:

- `task1` operates only on `sharedStruct.var_1`.
- `task2` operates only on `sharedStruct.var_2`.

If you select the result, the **Result Details** pane indicates that the access pattern protects all operations on the variable. On the **Variable Access** pane, the row for variable `sharedStruct` lists `Access pattern` as the protection type.

Shared Variables Protected Through Design Pattern and Mutex

```
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t lock;
pthread_t id1, id2;

int var;

void * t1(void* b) {
    pthread_mutex_lock(&lock);
    var++;
    pthread_mutex_unlock(&lock);
}

void * t2(void* a) {
    pthread_mutex_lock(&lock);
    var = 1;
    pthread_mutex_unlock(&lock);
}

int main(void) {
    pthread_create(&id1, NULL, t1, NULL);
    pthread_create(&id2, NULL, t2, NULL);

    return 0;
}
```

`var` is a shared, protected variable if you specify the following options:

Option Name	Value
Enable automatic concurrency detection for Code Prover (-enable-concurrency-detection)	☑

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

On the command-line, you can use the following:

```
polyspace-code-prover
   -enable-concurrency-detection
```

In this example, if you specify the concurrency detection option, Polyspace Code Prover detects that your program uses multitasking. Two task, `lock` and `var`, share two variables. `lock` is a pthread mutex variable, which `pthread_mutex_lock` and `pthread_mutex_unlock` use to lock and unlock their mutexes. The inherent pthread design patterns protect `lock`. The **Results Details** pane and **Variable Access** pane list `Design Pattern` as the protection type.

The mutex locking and unlocking mechanisms protect `var`, the other shared variable. The **Results Details** pane and **Variable Access** pane list `Mutex` as the protection type.

Check Information

Language: C | C++

See Also

`Potentially unprotected variable` | `Unused variable` | `Used non-shared variable`

Non-shared unused global variable

Global variables declared but not used

Description

A **non-shared unused** global variable has the following properties:

- The variable is declared in the code.
- Polyspace cannot detect a read or write operation on the variable.

In your verification results, these variables are colored gray on the **Source**, **Results List** and **Variable Access** panes. On the **Source** pane, the coloring is applied to the variable only during declaration.

Note The software does not display a complete list of unused global variables. Especially, in C++ projects, unused global variables can be suppressed from display.

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
```

```
    if(flag!=0) {
        var4 =loc_var;
    }


}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Check Information

Language: C | C++

See Also

`Potentially unprotected variable` | `Shared variable` | `Used non-shared variable`

Topics

"Interpret Polyspace Code Prover Access Results"

Used non-shared variable

Global variables used in a single task

Description

A **non-shared used** global variable has the following properties:

- The variable is used only in a single task.
- Polyspace detects at least one read or write operation on the variable.

In code that is not intended for multitasking, all global variables are non-shared.

In your verification results, these variables are colored black on the **Results List** and **Variable Access** panes.

Examples

Used and Unused Global Variables

```
int var1;
int var2;
int var3;
int var4;

int input(void);

void main() {
    int loc_var = input(), flag=0;

    var1 = loc_var;
    if(0) {
        var3 = loc_var;
    }
    if(flag!=0) {
        var4 =loc_var;
    }
```

```
}
```

If you verify the above code in a C project, the software lists `var2`, `var3` and `var4` as non-shared unused variables, and `var1` as a non-shared used variable.

`var3` and `var4` are used in unreachable code and are therefore marked as unused.

Note In a C++ project, the software does not list the unused variable `var2`.

Non-shared variables in multitasking code

```
unsigned int var_1;
unsigned int var_2;
volatile int randomVal;

void task1(void) {
    while(randomVal)
        operation(1);
}

void task2(void) {
    while(randomVal)
        operation(2);
}

void operation(int i) {
    if(i==1) {
        var_1++;
    }
    else    {
        var_2++;
    }
}

int main(void) {
    return 0;
}
```

In this example, even when you specify `task1` and `task2` for the option `Tasks (-entry points)`, the software determines that `var_1` and `var_2` are non-shared. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Even though both `task1` and `task2` call the function `operation`, because of the `if` statement in `operation`, `task1` can operate only on `var_1` and `task2` only on `var_2`.

Check Information

Language: C | C++

See Also

`Potentially unprotected variable` | `Shared variable` | `Unused variable`

6

Code Metrics

Comment Density

Ratio of number of comments to number of statements

Description

The metric specifies the ratio of comments to statements expressed as a percentage.

Based on HIS specifications:

- Multi-line comments count as one comment.

  For instance, the following constitutes one comment:

  ```
  // This function implements
  // regular maintenance on an internal database
  ```
- Comments that start with the source code line do not count as comments.

  For instance, this comment does not count as a comment for the metric but counts as a statement instead:

  ```
   remove(i); // Remove employee record
  ```
- A statement typically ends with a semi-colon with some exceptions. Exceptions include semi-colons in `for` loops or structure field declarations.

  For instance, the initialization, condition and increment within parentheses in a `for` loop is counted as one statement. The following counts as one statement:

  ```
  for(i=0; i <100; i++)
  ```

  If you also declare the loop counter at initialization, it counts as two statements.

The recommended lower limit for this metric is 20. For better readability of your code, try to place at least one comment for every five statements.

Examples

Comment Density Calculation

```
struct record {
    char name[40];
    long double salary;
    int isEmployed;
};

struct record dataBase[100];

struct record fetch(void);
void remove(int);

void maintenanceRoutines() {
// This function implements
// regular maintenance on an internal database
    int i;
    struct record tempRecord;

    for(i=0; i <100; i++) {
        tempRecord = fetch(); // This function fetches a record
        // from the database
        if(tempRecord.isEmployed == 0)
            remove(i);         // Remove employee record
        //from the database
    }
}
```

In this example, the comment density is 38. The calculation is done as follows:

Code	Running Total of Comments	Running Total of Statements

`struct record {`
`    char name[40];`
`    long double salary;`
`    int isEmployed;`
`};`
	0	1

Code	Running Total of Comments	Running Total of Statements

`struct record dataBase[100];`
`struct record fetch(void);`
`void remove(int);`
	0	4

`void maintenanceRoutines() {`	0	4
`// This function implements` `// regular maintenance on an internal database`	1	4
`int i;` `struct record tempRecord;`	1	6
`for(i=0; i <100; i++) {`	1	6

`tempRecord = fetch(); // This`
`function fetches a record`
`// from the database`
	2	7

`if(tempRecord.isEmployed == 0)`
`remove(i);`
`// Remove employee record`
`//from the database`
`}`
`}`
	3	8

There are 3 comments and 8 statements. The comment density is 3/8*100 = 38.

Metric Information

Group: File
Acronym: `COMF`

See Also

Cyclomatic Complexity

Number of linearly independent paths in function body

Description

This metric calculates the number of decision points in a function and adds one to the total. A decision point is a statement that causes your program to branch into two paths.

The recommended upper limit for this metric is 10. If the cyclomatic complexity is high, the code is both difficult to read and can cause more orange checks. Therefore, try to limit the value of this metric.

Computation Details

The metric calculation uses the following rules to identify decision points:

- An `if` statement is one decision point.
- The statements `for` and `while` count as one decision point, even when no condition is evaluated, for example, in infinite loops.
- Boolean combinations `(&&, ||)` do not count as decision points.
- `case` statements do not count as decision points unless they are followed by a `break` statement. For instance, this code has a cyclomatic complexity of two:

```
switch(num) {
        case 0:
        case 1:
        case 2:
            break;
        case 3:
        case 4:
    }
```

- The calculation is done after preprocessing:
  - Macros are expanded.

- Conditional compilation is applied. The blocks hidden by preprocessing directives are ignored.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag;
    if (x <= 0)
        /* Decision point 1*/
        flag = 1;
    else
    {
        if (x < y )
            /* Decision point 2*/
            flag = 1;
        else if (x==y)
            /* Decision point 3*/
            flag = 0;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Function with ? Operator

```
int foo (int x, int y) {
    if((x <0) ||(y < 0))
        /* Decision point 1*/
        return 0;
    else
        return (x > y ? x: y);
        /* Decision point 2*/
}
```

In this example, the cyclomatic complexity of `foo` is 3. The `?` operator is the second decision point.

Function with `switch` Statement

```
#include <stdio.h>

int foo(int x,int y, int ch)
{
    int val = 0;
    switch(ch) {
    case 1:
        /* Decision point 1*/
        val = x + y;
        break;
    case 2:
        /* Decision point 2*/
        val = x - y;
        break;
    default:
        printf("Invalid choice.");
    }
    return val;
}
```

In this example, the cyclomatic complexity of `foo` is 3.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Decision point 1*/
        count = 1;
    else
        while(x>y) {
            /* Decision point 2*/
            x--;
            if(count< bound) {
                /* Decision point 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the cyclomatic complexity of `foo` is 4.

Metric Information

Group: Function
Acronym: `VG`

See Also

Estimated Function Coupling

Measure of complexity between levels of call tree

Description

This metric provides an approximate measure of complexity between different levels of the call tree. The metric is defined as:

number of call occurrences – number of function definitions + 1

If there are more function definitions than function calls, the estimated function coupling result is negative.

This metric:

- Counts function calls and function definitions in the current file only.

  It does not count function definitions in a header file included in the current file.
- Treats `static` and `inline` functions like any other function.

Examples

Same Function Called Multiple Times

```
void checkBounds(int *);
int getUnboundedValue();

int getBoundedValue(void) {
    int num = getUnboundedValue();
    checkBounds(&num);
    return num;
}

void main() {
    int input1=getBoundedValue(), input2= getBoundedValue(), prod;
    prod = input1 * input2;
```

```
    checkBounds(&prod);
}
```

In this example, there are:

- `5` call occurrences. Both `getBoundedValue` and `checkBounds` are called twice and `getUnboundedValue` is called once.
- `2` function definitions. `main` and `getBoundedValue` are defined.

Therefore, the Estimated function coupling is `5 - 2 + 1 = 4`.

Negative Estimated Function Coupling

```
int foobar(int a, int b){
    return a+b;
}

int bar(int b){
    return b+2;
}

int foo(int a){
    return a<<2;
}

int main(int x){
    foobar(x,x+2);
    return 0;
}
```

This example shows how you can get a negative estimated function coupling result. In this example, you see:

- `1` function call in `main`.
- `4` defined functions: `foobar`, `bar`, `foo`, and `main`.

Therefore, the estimated function coupling is `1 - 4 + 1 = -2`.

Metric Information

Group: File

Acronym: FCO

See Also

Higher Estimate of Local Variable Size

Total size of all local variables in function

Description

This metric provides a conservative estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables
- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming 4 bytes for `int`, the higher estimate of local variable size is 28. The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter `param`	4	8
Local variables `var_1` and `var_2`	`4+4=8`	16
Local variables defined in the `if` condition	`(4+4)+4=12` The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The sum of the two branches is 12 bytes.	28

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int` and eight bytes for `double` and four bytes for alignment, the higher estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in `Target processor type (-target)`. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4
`var_1`	4	8

Variable	Size (in Bytes)	Running Total
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` has eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:
    MySimpleClass() {};
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};
```

```
int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

  The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}

void func2() {
```

```
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function
Acronym: `LOCAL_VARS_MAX`

See Also

Introduced in R2016b

Language Scope

Language scope

Description

This metric measures the cost of maintaining or changing a function. It is calculated as:

`(N1 + N2)/(n1 + n2)`

Here:

- `N1` is the number of occurrences of operators.

  Other than identifiers (variable or function names) and literal constants, everything else counts as operators.
- `N2` is the number of occurrences of operands.
- `n1` is the number of distinct operators.
- `n2` is the number of distinct operands.

  The metric considers a literal constant with a suffix as different from the constant without the suffix. For instance, `0` and `0U` are considered different.

Tip To find `N1 + N2`, count the total number of tokens. To find `n1 + n2`, count the number of unique tokens.

The recommended upper limit for this metric is 4. For lower maintenance cost for a function, try to enforce an upper limit on this metric. For instance, if the same operand occurs many times, to change the operand name, you have to make many substitutions.

Examples

Language Scope Calculation

```
int f(int i)
{
```

```
    if (i == 1)
        return i;
    else
        return i * g(i-1);
}
```

In this example:

- N1 = 19.
- N2 = 9.
- n1 = 12.

  The distinct operators are int, (,), {, if, ==, return, else, *, -, ;, }.
- n2 = 4.

  The distinct operands are f, i, 1 and g.

The language scope of f is (17 + 9) / (12 + 4) = 1.8.

C++ Namespaces in Language Scope Calculation

```
namespace std {
  int func2() {
    return 123;
  }
};

namespace my_namespace {
  using namespace std;
  int func1(int a, int b) {
    return func2();
  }
};
```

In this example, the namespace std is implicitly associated with func2. The language scope computation treats func2() as std::func2(). Likewise, the computation treats func1() as my_namespace::func1().

For instance, the language scope value for func1 is 1.3. To break down this calculation:

- N1 + N2 = 20.

- `n1 + n2` = 15.

  The distinct operators are `int`, `::`, `(`, comma, `)`, `{`, `return`, `;`, and `}`.

  The distinct operands are `my_namespace`, `func1`, `a`, `b`, `std`, and `func2`.

Metric Information

Group: Function
Acronym: `VOCF`

See Also

Lower Estimate of Local Variable Size

Total size of local variables in function taking nested scopes into account

Description

This metric provides an optimistic estimate of the total size of local variables in a function. The metric is the sum of the following sizes in bytes:

- Size of function return value
- Sizes of function parameters
- Sizes of local variables

  Suppose that the function has variable definitions in nested scopes as follows:

```
type func (type param_1, ...) {

  {
    /* Scope 1 */
    type var_1, ...;
  }
  {
    /* Scope 2 */
    type var_2, ...;
  }
}
```

  The software computes the total variable size in each scope and uses whichever total is greatest. For instance, if a conditional statement has variable definitions, the software computes the total variable size in each branch, and then uses whichever total is greatest. If a nested scope itself has further nested scopes, the same process is repeated for the inner scopes.

  A variable defined in a nested scope is not visible outside the scope. Therefore, some compilers reuse stack space for variables defined in separate scopes. This metric provides a more accurate estimate of stack usage for such compilers. Otherwise, use the metric `Higher Estimate of Local Variable Size`. This metric adds the size of all local variables, whether or not they are defined in nested scopes.

- Additional padding introduced for memory alignment

Your actual stack usage due to local variables can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When computing this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When computing this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage due to local variables.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric also takes into account `#pragma pack` directives in your code. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

All Variables of Same Type

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, assuming four bytes for `int`, the lower estimate of local variable size is 24. The breakup of the metric is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	4	4
Parameter `param`	4	8
Local variables `var_1` and `var_2`	`4+4=8`	16
Local variables defined in the `if` condition	`max(4+4,4)= 8` The size of variables in the first branch is eight bytes. The size in the second branch is four bytes. The maximum of the two branches is eight bytes.	24

No padding is introduced for memory alignment because all the variables involved have the same type.

Variables of Different Types

```
char func(char param) {
  int var_1;
  char var_2;
  double var_3;
}
```

In this example, assuming one byte for `char`, four bytes for `int`, eight bytes for `double` and four bytes for alignment, the lower estimate of local variable size is 20. The alignment is usually the word size on your platform. In your Polyspace project, you specify the alignment through your target processor. For more information, see the Alignment column in . For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

The breakup of the size is shown in this table.

Variable	Size (in Bytes)	Running Total
Return value	1	1

Variable	Size (in Bytes)	Running Total
Additional padding introduced before `param` is stored	0 No memory alignment is required because the next variable `param` has the same size.	1
Parameter `param`	1	2
Additional padding introduced before `var_1` is stored	2 Memory must be aligned using padding because the next variable `var_1` requires four bytes. The storage must start from a memory address at a multiple of four.	4
`var_1`	4	8
Additional padding introduced before `var_2` is stored	0 No memory alignment is required because the next variable `var_2` has smaller size.	8
`var_2`	1	9
Additional padding introduced before `var_3` is stored	3 Memory must be aligned using padding because the next variable `var_3` requires eight bytes. The storage must start from a memory address at a multiple of the alignment, four bytes.	12
`var_3`	8	20

The rules for the amount of padding are:

- If the next variable stored has the same or smaller size, no padding is required.
- If the next variable has a greater size:
  - If the variable size is the same as or less than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of its size.
  - If the variable size is greater than the alignment on the platform, the amount of padding must be sufficient so that the storage address is a multiple of the alignment.

C++ Methods and Objects

```
class MySimpleClass {
  public:
    MySimpleClass() {};
    MySimpleClass(int) {};
    ~MySimpleClass() {};
};

int main() {
  MySimpleClass c;
  return 0;
}
```

In this example, the estimated local variable sizes are:

- Constructor `MySimpleClass::MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer, which is an implicit argument to the constructor. You specify the pointer size using the option `Target processor type (-target)`.
- Constructor `MySimpleClass::MySimpleClass(int)`: Eight bytes.

  The size comes from the `this` pointer and the `int` argument.
- Destructor `MySimpleClass::~MySimpleClass()`: Four bytes.

  The size comes from the `this` pointer.
- `main()`: Five bytes.

> The size comes from the `int` return value and the size of object `c`. The minimum size of an object is the alignment that you specify using the option `Target processor type (-target)`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

C++ Functions with Object Arguments

```
class MyClass {
  public:
    MyClass() {};
    MyClass(int) {};
    ~MyClass() {};
  private:
    int i[10];
};
void func1(const MyClass& c) {
}

void func2() {
  func1(4);
}
```

In this example, the estimated local variable size for `func2()` is 40 bytes. When `func2()` calls `func1()`, a temporary object of the class `MyClass` is created. The object has ten `int` variables, each with a size of four bytes.

Metric Information

Group: Function
Acronym: `LOCAL_VARS_MIN`

See Also

Introduced in R2016b

Maximum Stack Usage

Total size of local variables in function plus maximum stack usage from callees

Description

This metric provides a conservative estimate of the stack usage by a function. The metric is the sum of these sizes in bytes:

- `Higher Estimate of Local Variable Size`
- Maximum value from the stack usages of the function callees. The computation uses the maximum stack usage of each callee.

  For instance, in this example, the maximum stack usage of `func` is the same as the maximum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the greatest stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.
- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function with One Callee

```
double func(int);
double func2(int);

double func(int status) {
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 32 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), local variable `temp` counted twice (4+8=12 bytes), and return value (8 bytes).

  The metric does not take into account that the first `temp` is no longer live when the second `temp` is defined.

- `func`: 52 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 32 bytes of maximum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);


void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and local variable `temp` counted twice (4+8=12 bytes).

- `func2`: 8 bytes
- `func`: 20 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (16 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}

void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 12 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum stack usage from the two branches (8 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>

void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
```

```
  }
  va_end(ap);
}


void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}


void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The maximum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the maximum number of arguments. The call with the maximum number of arguments is the call in `call_fun_vararg1` with five arguments (one for the fixed parameter and four for the variable parameters). The maximum stack usages are:

- `fun_vararg`: 36 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the maximum number of parameters. In that call, there are four variable arguments: three `int` and one `long long int` variable (3 times 4 + 1 times 8 = 20 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).
- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes, of which 12 bytes are for the local variables and 8 bytes are for the two parameters of `fun_vararg`).

Metric Information

Group: Function
Acronym: `MAX_STACK`

See Also

`Higher Estimate of Local Variable Size` | `Minimum Stack Usage` | `Program Maximum Stack Usage`

Topics

"Determination of Program Stack Usage"

Introduced in R2017b

Minimum Stack Usage

Total size of local variables in function taking nested scopes into account plus maximum stack usage from callees

Description

This metric provides an optimistic estimate of the stack usage by a function. Unlike the metric `Maximum Stack Usage`, this metric takes nested scopes into account. For instance, if variables are defined in two mutually exclusive branches of a conditional statement, the metric considers that the stack space allocated to the variables in one branch can be reused in the other branch.

The metric is the sum of these sizes in bytes:

- `Lower Estimate of Local Variable Size`.
- Maximum value from the stack usages of the function callees. The computation uses the minimum stack usage of each callee.

  For instance, in this example, the minimum stack usage of `func` is the same as the minimum stack usage of `func1` or `func2`, *whichever is greater*.

  ```
  void func(void) {
      func1();
      func2();
  }
  ```

  If the function calls are in different branches of a conditional statement, this metric considers the branch with the least stack usage.

  The analysis does the stack size estimation later on when it has resolved which function calls actually occur. For instance, if a function call occurs in unreachable code, the stack size does not take the call into account. The analysis can also take into account calls through function pointers.

Your actual stack usage can be different from the metric value.

- Some of the variables are stored in registers instead of on the stack.

- Your compiler performs variable liveness analysis to enable certain memory optimizations. When estimating this metric, Polyspace does not consider these optimizations.
- Your compiler uses additional memory during a function call. For instance, compilers store the address to which the execution returns following the function call. When estimating this metric, Polyspace does not consider this hidden memory usage.

However, the metric provides a reasonable estimate of the stack usage.

To determine the sizes of basic types, the software uses your specifications for `Target processor type (-target)`. The metric takes into account `#pragma pack` directives in your code. For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Examples

Function with One Callee

```
double func2(int);

double func(int status) {
    double res = func2(status);
    return res;
}

double func2(int status) {
    double res;
    if(status == 0) {
      int temp;
      res = 0.0;
    }
    else {
      double temp;
      res = 1.0;
    }
    return res;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func2`: 28 bytes

  This value includes the sizes of its parameter (4 bytes), local variable `res` (8 bytes), one of the two local variables `temp` (8 bytes), and return value (8 bytes).

  The metric takes into account that the first `temp` is no longer live when the second `temp` is defined. It uses the variable `temp` with data type `double` because its size is greater.
- `func`: 48 bytes

  This value includes the sizes of its parameter, local variable `res`, and return value, a total of 20 bytes. This value includes the 28 bytes of minimum stack usage by its callee, `func2`.

Function with Multiple Callees

```
void func1(int);
void func2(void);


void func(int status) {
    func1(status);
    func2();
}

void func1(int status) {
    if(status == 0) {
      int val;
    }
    else {
      double val2;
    }
}

void func2(void) {
    double val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 12 bytes

This value includes the sizes of its parameter (4 bytes) and one of the two local variables `temp` (8 bytes). The metric takes into account that the first `temp` is no longer live when the second `temp` is defined.

- `func2`: 8 bytes
- `func`: 16 bytes

  This value includes the sizes of its parameter (4 bytes) and the maximum of stack usages of `func1` and `func2` (12 bytes).

Function with Multiple Callees in Different Branches

```
void func1(void);
void func2(void);


void func(int status) {
    if(status==0)
        func1();
    else
        func2();
}

void func1(void) {
    double val;
}

void func2(void) {
    int val;
}
```

In this example, assuming four bytes for `int` and eight bytes for `double`, the maximum stack usages are:

- `func1`: 8 bytes
- `func2`: 4 bytes
- `func`: 8 bytes

  This value includes the sizes of its parameter (4 bytes) and the minimum stack usage from the two branches (4 bytes).

Functions with Variable Number of Parameters (Variadic Functions)

```
#include <stdarg.h>

void fun_vararg(int x, ...) {
  va_list ap;
  va_start(ap, x);
  int i;
  for (i=0; i<x; i++) {
    int j = va_arg(ap, int);
  }
  va_end(ap);
}


void call_fun_vararg1(void) {
  long long int l = 0;
  fun_vararg(3, 4, 5, 6, l);
}


void call_fun_vararg2(void) {
  fun_vararg(1,0);
}
```

In this function, `fun_vararg` is a function with variable number of parameters. The minimum stack usage of `fun_vararg` takes into account the call to `fun_vararg` with the minimum number of arguments. The call with the minimum number of arguments is the call in `call_fun_vararg2` with two arguments (one for the fixed parameter and one for the variable parameter). The minimum stack usages are:

- `fun_vararg`: 20 bytes.

  This value takes into account:

  - The size of the fixed parameter `x` (4 bytes).
  - The sizes of the variable parameters from the call with the minimum number of parameters. In that call, there is only one variable argument of type `int` (4 bytes).
  - The sizes of the local variables `i`, `j` and `ap` (12 bytes). The size of the `va_list` variable uses the pointer size defined in the target (in this case, 4 bytes).

- `call_fun_vararg1`: 44 bytes.

  This value takes into account:

  - The stack size usage of `fun_vararg` with five arguments (36 bytes, of which 12 bytes are for the local variable sizes and 20 bytes are for the fixed and variable parameters of `fun_vararg`).
  - The size of local variable `l` (8 bytes).
- `call_fun_vararg2`: 20 bytes.

  Since `call_fun_vararg2` has no local variables, this value is the same as the stack size usage of `fun_vararg` with two arguments (20 bytes).

Metric Information

Group: Function
Acronym: `MIN_STACK`

See Also

`Lower Estimate of Local Variable Size` | `Maximum Stack Usage` | `Program Minimum Stack Usage`

Topics

"Determination of Program Stack Usage"

Introduced in R2017b

Number of Call Levels

Maximum depth of nesting of control flow structures

Description

This metric specifies the maximum nesting depth of control flow statements such as `if`, `switch`, `for`, or `while` in a function. A function without control-flow statements has a call level 1.

The recommended upper limit for this metric is 4. For better readability of your code, try to enforce an upper limit for this metric.

Examples

Function with Nested `if` Statements

```
int foo(int x,int y)
{
    int flag = 0;
    if (x <= 0)
        /* Call level 1*/
        flag = 1;
    else
    {
        if (x <= y )
            /* Call level 2*/
            flag = 1;
        else
            flag = -1;
    }
    return flag;
}
```

In this example, the number of call levels of `foo` is 2.

Function with Nesting of Different Control-Flow Statements

```
int foo(int x,int y, int bound)
{
    int count = 0;
    if (x <= y)
        /* Call level 1*/
        count = 1;
    else
        while(x>y) {
            /* Call level 2*/
            x--;
            if(count< bound) {
                /* Call level 3*/
                count++;
            }
        }
    return count;
}
```

In this example, the number of call levels of `foo` is 3.

Metric Information

Group: Function
Acronym: LEVEL

See Also

Number of Call Occurrences

Number of calls in function body

Description

This metric specifies the number of function calls in the body of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of call occurrences in `foo` is 4.

Function Called in a Loop

```
#include<stdio.h>

void fillArraySize10(int *arr) {
    for(int i=0; i<10; i++)
        arr[i]=getVal();
}

int getVal(void) {
    int val;
    printf("Enter a value:");
```

```
    scanf("%d", &val);
    return val;
}
```

In this example, the number of call occurrences in `fillArraySize10` is 1.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of call occurrences in `fibonacci` is 2.

Metric Information

Group: Function
Acronym: `NCALLS`

See Also

Number of Called Functions

Number of callees of a function

Description

This metric specifies the number of callees of a function.

Calls through a function pointer are not counted. Calls in unreachable code and calls to standard library functions are counted. `assert` is considered as a macro and not a function, so it is not counted.

The recommended upper limit for this metric is 7. For more self-contained code, try to enforce an upper limit on this metric.

Examples

Same Function Called Multiple Times

```
int func1(void);
int func2(void);

int foo() {
    return (func1() + func1()*func1() + 2*func2());
}
```

In this example, the number of called functions in `foo` is 2. The called functions are `func1` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
```

```
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of called functions in `fibonacci` is 1. The called function is `fibonacci` itself.

Metric Information

Group: Function
Acronym: `CALLS`

See Also

Number of Calling Functions

Number of distinct callers of a function

Description

This metric measures the number of distinct callers of a function.

Calls through a function pointer are not counted. Calls in unreachable code are counted. Even if a caller calls a function more than once, it is counted only once when this metric is calculated.

The recommended upper limit for this metric is 5. For more self-contained code, try to enforce an upper limit on this metric.

Examples

Same Function Calling a Function Multiple Times

```
#include <stdio.h>

int getVal() {
    int myVal;
    printf("Enter a value:");
    scanf("%d", &myVal);
    return myVal;
}

int func() {
    int val=getVal();
    if(val<0)
        return 0;
    else
        return val;
}

int func2() {
    int val=getVal();
```

```
    while(val<0)
        val=getVal();
    return val;
}
```

In this example, the number of calling functions for `getVal` is 2. The calling functions are `func` and `func2`.

Recursive Function

```
#include <stdio.h>

void main() {
 int count;
 printf("How many numbers ?");
 scanf("%d",&count);
 fibonacci(count);
}

int fibonacci(int num)
{
   if ( num == 0 )
      return 0;
   else if ( num == 1 )
      return 1;
   else
      return ( fibonacci(num-1) + fibonacci(num-2) );
}
```

In this example, the number of calling functions for `fibonacci` is 2. The calling functions are `main` and `fibonacci` itself.

Metric Information

Group: Function
Acronym: `CALLING`

See Also

Number of Direct Recursions

Number of instances of a function calling itself directly

Description

This metric specifies the number of direct recursions in your project.

A direct recursion is a recursion where a function calls itself in its own body. If indirect recursions do not occur, the number of direct recursions is equal to the number of recursive functions.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. To detect use of recursions, check for violations of `MISRA C:2012 Rule 17.2`.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

In this example, the number of direct recursions is 1.

Metric Information

Group: Project
Acronym: `AP_CG_DIRECT_CYCLE`

See Also

Number of Executable Lines

Number of executable lines in function body

Description

This metric measures the number of executable lines in a function body. When calculating the value of this metric, Polyspace excludes declarations without static initializers, comments, blank lines, braces or preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 9. The calculation excludes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function
Acronym: `FXLN`

See Also

Number of Files

Number of source files

Description

This metric calculates the number of source files in your project.

Metric Information

Group: Project
Acronym: `FILES`

See Also

Number of Function Parameters

Number of function arguments

Description

This metric measures the number of function arguments.

If ellipsis is used to denote variable number of arguments, when calculating this metric, the ellipsis is not counted.

The recommended upper limit for this metric is 5. For less dependency between functions and fewer side effects, try to enforce an upper limit on this metric.

Examples

Function with Fixed Arguments

```
int initializeArray(int* arr, int size) {
}
```

In this example, `initializeArray` has two parameters.

Function with Type Definition in Arguments

```
int getValueInLoc(struct {int* arr; int size;}myArray, int loc) {
}
```

In this example, `getValueInLoc` has two parameters.

Function with Variable Arguments

```
double average ( int num, ... )
{
    va_list arg;
    double sum = 0;
```

```
    va_start ( arg, num );

    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arg, double );
    }
    va_end ( arg);

    return sum / num;
}
```

In this example, `average` has one parameter. The ellipsis denoting variable number of arguments is not counted.

Metric Information

Group: Function
Acronym: `PARAM`

See Also

Number of Goto Statements

Number of `goto` statements

Description

This metric measures the number of `goto` statements in a function.

`break` and `continue` statements are not counted.

The recommended upper limit on this metric is 0. For better readability of your code, avoid `goto` statements in your code. To detect use of `goto` statements, check for violations of `MISRA C:2012 Rule 15.1`.

Examples

Function with `goto` Statements

```
#define SIZE 10
int initialize(int **arr, int loc);
void printString(char *);
void printErrorMessage(void);
void printExecutionMessage(void);

int main()
{
    int *arrayOfStrings[SIZE],len[SIZE],i;
    for ( i = 0; i < SIZE; i++ )
    {
        len[i] = initialize(arrayOfStrings,i);
    }

    for ( i = 0; i < SIZE; i++ )
    {
        if(len[i] == 0)
            goto emptyString;
        else
            goto nonEmptyString;
```

```
        loop: printExecutionMessage();
    }

emptyString:
    printErrorMessage();
    goto loop;
nonEmptyString:
    printString(arrayOfStrings[i]);
    goto loop;
}
```

In this example, the function `main` has 4 `goto` statements.

Metric Information

Group: Function
Acronym: `GOTO`

See Also

Number of Header Files

Number of included header files

Description

This metric measures the number of header files in the project. Both directly and indirectly included header files are counted.

The metric gives a slightly higher number than the actual number of header files that you use because Polyspace® internal header files and header files included by those files are also counted. For the same reason, the metric can vary slightly even if you do not explicitly include new header files or remove inclusion of header files from your code. For instance, the number of Polyspace® internal header files can vary if you change your analysis options.

Metric Information

Group: Project
Acronym: `INCLUDES`

See Also

Number of Instructions

Number of instructions per function

Description

This metric measures the number of instructions in a function body.

The recommended upper limit for this metric is 50. For more modular code, try to enforce an upper limit for this metric.

Computation Details

The metric is calculated using the following rules:

- A simple statement ending with a `;` is one instruction.

  If the statement is empty, it does not count as an instruction.
- A variable declaration counts as one instruction only if the variable is also initialized.
- Control flow statements such as `if`, `for`, `break`, `goto`, `return`, `switch`, `while`, `do-while` count as one instruction.
- The following do not count as instructions by themselves:

  - Beginning of a block of code

    For instance, the following counts as one instruction:

    ```
    {
        var = 1;
    }
    ```
  - Labels

    For instance, the following counts as two instructions. The `case` labels do not count as instructions.

    ```
    switch (1) {  // Instruction 1: switch
        case 0:
        case 1:
    ```

```
        case 2:
        default:
        break;    // Instruction 2: break
    }
```

Examples

Calculation of Number of Instructions

```
int func(int* arr, int size) {
    int i, countPos=0, countNeg=0, countZero = 0;
    for(i=0; i<size; i++) {
        if(arr[i] >0)
            countPos++;
        else if(arr[i] ==0)
            countZero++;
        else
            countNeg++;
    }
}
```

In this example, the number of instructions in `func` is 9. The instructions are:

1. `countPos=0`
2. `countNeg=0`
3. `countZero=0`
4. `for(i=0;i<size;i++) { ... }`
5. `if(arr[i] >=0)`
6. `countPos++`
7. `else if(arr[i]==0)`

   The ending `else` is counted as part of the `if-else` instruction.
8. `countZero++`
9. `countNeg++`

Note This metric is different from the number of executable lines. For instance:

- `for(i=0;i<size;i++)` has 1 instruction and 1 executable line.
- The following code has 1 instruction but 3 executable lines.

```
for(i=0;
    i<size;
    i++)
```

Metric Information

Group: Function
Acronym: `STMT`

See Also

Number of Lines

Total number of lines in a file

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace includes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Metric Information

Group: File
Acronym: `TOTAL_LINES`

See Also

Number of Lines Within Body

Number of lines in function body

Description

This metric calculates the number of lines in function body. When calculating the value of this metric, Polyspace includes declarations, comments, blank lines, braces and preprocessing directives.

If the function body contains a `#include` directive, the included file source code is also calculated as part of this metric.

This metric is not calculated for C++ templates.

Examples

Function with Declarations, Braces and Comments

```
void func(int);

int getSign(int arg) {
    int sign;
    if(arg<0) {
        sign=-1;
        func(-arg);
        /* func takes positive arguments */
    }
    else if(arg==0)
        sign=0;
    else {
        sign=1;
        func(arg);
    }
    return sign;
}
```

In this example, the number of executable lines of `getSign` is 13. The calculation includes:

- The declaration `int sign;`.
- The comment `/* ... */`.
- The two lines with braces only.

Metric Information

Group: Function
Acronym: `FLIN`

See Also

Number of Lines Without Comment

Number of lines of code excluding comments

Description

This metric calculates the number of lines in a file. When calculating the value of this metric, Polyspace excludes comments and blank lines.

This metric is calculated for source files and header files in the same folders as source files. If you want:

- The metric reported for other header files, change the default value of the option `Generate results for sources and (-generate-results-for)`.
- The metric not reported for header files at all, change the value of the option `Do not generate results for (-do-not-generate-results-for)` to `all-headers`.

For more information on analysis options, see the documentation for Polyspace Code Prover or Polyspace Code Prover Server.

Metric Information

Group: File
Acronym: `LINES_WITHOUT_CMT`

See Also

Number of Local Non-Static Variables

Total number of local variables in function

Description

This metric provides the number of local variables in a function.

The metric excludes static variables. To find number of static variables, use the metric `Number of Local Static Variables`.

Examples

Non-Structured Variables

```
int flag();

int func(int param) {
  int var_1;
  int var_2;
  if (flag()) {
      int var_3;
      int var_4;
    } else {
      int var_5;
    }
}
```

In this example, the number of local non-static variables in `func` is 5. The number does not include the function arguments and return value.

Arrays and Structured Variables

```
typedef struct myStruct{
   char  arr1[50];
   char  arr2[50];
   int   val;
```

```
} myStruct;

void func(void) {
  myStruct var;
  char localArr[50];
}
```

In this example, the number of local non-static variables in `func` is 2: the structured variable `var` and the array `localArr`.

Variables in Class Methods

```
class Rectangle {
    int width, height;
  public:
    void set (int,int);
    int area (void);
} rect;

int Rectangle::area (void) {
    int temp;
    temp = width * height;
    return(temp);
}
```

In this example, the number of local non-static variables in `Rectangle::area` is 1: the variable `temp`.

Metric Information

Group: Function
Acronym: `LOCAL_VARS`

See Also

Introduced in R2017a

Number of Local Static Variables

Total number of local static variables in function

Description

This metric provides the number of local static variables in a function.

Examples

Number of Static Variables

```
void func(void) {
  static int var_1 = 0;
  int var_2;
}
```

In this example, the number of static variables in `func` is 1. For examples of different types of variables, see `Number of Local Non-Static Variables`.

Metric Information

Group: Function
Acronym: `LOCAL_STATIC_VARS`

See Also

Introduced in R2017a

Number of Paths

Estimated static path count

Description

This metric measures the number of paths in a function.

The recommended upper limit for this metric is 80. If the number of paths is high, the code is difficult to read and can cause more orange checks. Try to limit the value of this metric.

Computation Details

The number of paths is calculated according to these rules:

- If the statements in a function do not break the control flow, the number of paths is one.

  Even an empty statement such as `;` or empty block such as `{}` counts as one path.
- The number of paths for a control flow statement is calculated as follows:
  - `if-else if-else`: The number of paths is the sum of paths calculated in the `if` block, each `else if` block, and the concluding `else` block. When the concluding `else` block is omitted, the path count is increased by 1.

    For instance, the statement `if(..) {} else if(..) {} else {}` counts as three paths. The statement `if() {}` counts as two paths, one for the `if` block and one for the omitted `else` block.
  - `switch-case`: Every `case` with `break` statement adds one to the path count. The `default` statement counts as one path, even if it is omitted.

    For instance, the statement `switch (var) { case 1: .. break; case 2: .. break; default: .. }` counts as three paths.
  - `for`, `while`, and `do-while`: The number of paths is equal to the number of paths in the loop body + 1.

    For instance, the statement `while(0) {;}` counts as two paths.

  - Ternary operators: A statement with a ternary operator such as

    ```
    result = a > b ? a : b;
    ```

    is counted as one statement that does not break the control flow. The number of paths is considered as one.

- If more than one control flow statement are present in a sequence, the number of paths is the product of the path count for each control flow statement.

  For instance, if a function has three `for` loops and two `if-else` statements, the number of paths is 2 × 2 × 2 × 2 × 2 = 32.

  If many control flow statements are present in a function, the number of paths can be large. Nested control flow statements reduce the number of paths at the cost of increasing the depth of nesting. For an example, see “Function with Nested Control Flow Statements” on page 6-69.

- The software displays specific values in cases where the metric is not calculated:

  - If `goto` statements are present in the body of the function, Polyspace cannot calculate the number of paths. The software displays a metric value of -1.
  - If the number of paths reaches an internal limit, the calculation stops. The software displays this limit as the metric value. The limit is 9223372036854775807 (indicating the hexadecimal number 0x7fffffffffffffff).

Examples

Function with One Path

```
void func(int ch) {
    switch (ch)
    {
    case 1:
    case 2:
    case 3:
    case 4:
    default:
    }
}
```

In this example, `func` has one path.

Function with Control Flow Statement Causing Multiple Paths

```
void func(int ch) {
    switch (ch)
    {
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
    case 4:
        break;
    default:
    }
}
```

In this example, `func` has five paths. Apart from the path that goes through the `cases` and `default`, each `break` causes the creation of a new path.

Function with Nested Control Flow Statements

```
void func()
{
    int i = 0, j = 0, k = 0;
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            for (k=0; k<10; k++)
            {
                if (i < 2 )
                    ;
                else
                {
                    if (i > 5)
                        ;
                    else
                        ;
                }
            }
        }
    }
}
```

In this example, `func` has six paths. The number is calculated as follows:

- The innermost `if-else` block counts as two paths.
- The outer `if-else` block counts as three paths, one path for the `if` block and the previous two paths for the `else` block.
- The innermost `for` loop counts as four paths, one path for the loop and the previous three paths for the `if-else` blocks.
- The next two outer loops add one path each.

Therefore, the number of paths in `func` is six.

Metric Information

Group: Function
Acronym: `PATH`

See Also

Number of Potentially Unprotected Shared Variables

Number of unprotected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- At least one operation on the variable is not protected from interruption by operations in other tasks.

Examples

Unprotected Shared Variables

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}
```

```
void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is an unprotected shared variable if you specify `task` and `interrupt_handler` as entry points and do not specify protection mechanisms.

The operation `shared_var = INT_MAX` can interrupt the other operations on `shared_var` and cause unpredictable behavior.

Metric Information

Group: Project
Acronym: UNPSHV

See Also

Introduced in R2018b

Number of Protected Shared Variables

Number of protected shared variables

Description

This metric measures the number of variables with the following properties:

- The variable is used in more than one task.
- All operations on the variable are protected from interruption through critical sections or temporal exclusions.

Examples

Shared Variables Protected Through Temporal Exclusion

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        reset();
        inc();
        inc();
    }
}

void interrupt() {
    shared_var = INT_MAX;
```

```
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        interrupt();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following options:

Option	Value

Entry points	`task`
	
	`interrupt_handler`

Temporally exclusive tasks	`task interrupt_handler`

The variable is shared between `task` and `interrupt_handler`. However, because `task` and `interrupt_handler` are temporally exclusive, operations on the variable cannot interrupt each other.

Shared Variables Protected Through Critical Sections

```
#include <limits.h>
int shared_var;

void inc() {
    shared_var+=2;
}

void reset() {
    shared_var = 0;
}

void take_semaphore(void);
void give_semaphore(void);
```

```
void task() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        reset();
        inc();
        inc();
        give_semaphore();
    }
}

void interrupt() {
    shared_var = INT_MAX;
}

void interrupt_handler() {
    volatile int randomValue = 0;
    while(randomValue) {
        take_semaphore();
        interrupt();
        give_semaphore();
    }
}

void main() {
}
```

In this example, `shared_var` is a protected shared variable if you specify the following:

Option	Value	

Entry points	`task`
	
	`interrupt_handler`	

Critical section details	**Starting routine**	**Ending routine**
	`take_semaphore`	`give_semaphore`

The variable is shared between `task` and `interrupt_handler`. However, because operations on the variable are between calls to the starting and ending procedure of the same critical section, they cannot interrupt each other.

Metric Information

Group: Project
Acronym: PSHV

See Also

Introduced in R2018b

Number of Recursions

Number of call graph cycles over one or more functions

Description

The metric provides a quantitative estimate of the number of recursion cycles in your project. The metric is the sum of:

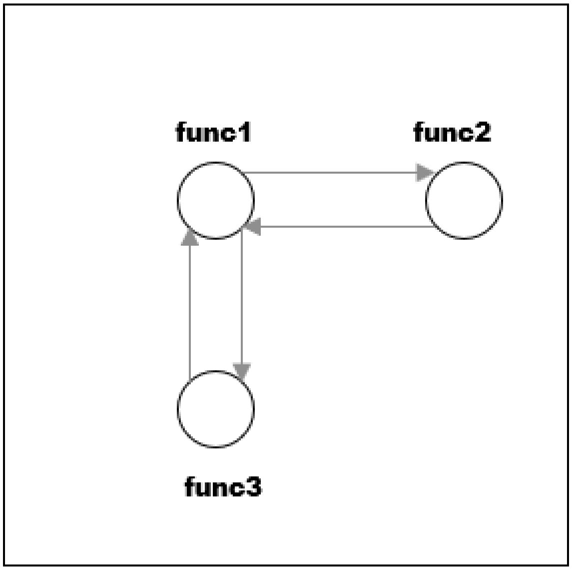
- Number of direct recursions (self recursive functions or functions calling themselves).
- Number of strongly connected components formed by the indirect recursion cycles in your project. If you consider the recursion cycles as a directed graph, the graph is strongly connected if there is a path between all pairs of vertices.

  To compute the number of strongly connected components:

  1. Draw the recursion cycles in your code.

     For instance, the recursion cycles in this example are shown below.

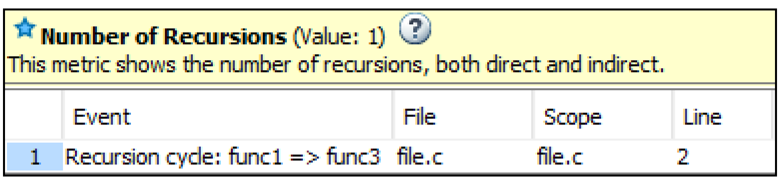
     ```
     volatile int checkStatus;
     void func1() {
        if(checkStatus) {
             func2();
        }
        else {
             func3();
        }
     }

     func2() {
        func1();
     }

     func3() {
        func1();
     }
     ```



2 Identify the number of strongly connected components formed by the recursion cycles.

In the preceding example, there is one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph.

The event list below the metric shows one of the recursion cycles in the strongly connected component.



Number of Recursions (Value: 1)
This metric shows the number of recursions, both direct and indirect.

	Event	File	Scope	Line
1	Recursion cycle: func1 => func3	file.c	file.c	2

Calls through a function pointer are not considered.

The recommended upper limit for this metric is 0. To avoid the possibility of exceeding available stack space, do not use recursions in your code. Recursions can tend to exhaust stack space easily. See examples of stack size growth with recursions described for this CERT-C rule that forbids recursions.

To detect use of recursions, check for violations of one of `MISRA C:2012 Rule 17.2`,MISRA C: 2004 Rule 16.2, `MISRA C++:2008 Rule 7-5-4` or JSF® Rule 119. Note that these rule checkers consider explicit function calls only. For instance, in C++ code, the rule checkers ignore implicit calls to constructors during object creation. However, the metrics computation considers both implicit and explicit calls.

Examples

Direct Recursion

```
int getVal(void);

void main() {
    int count = getVal(), total;
    assert(count > 0 && count <100);
    total = sum(count);
}

int sum(int val) {
    if(val<0)
        return 0;
    else
        return (val + sum(val-1));
}
```

In this example, the number of recursions is 1.

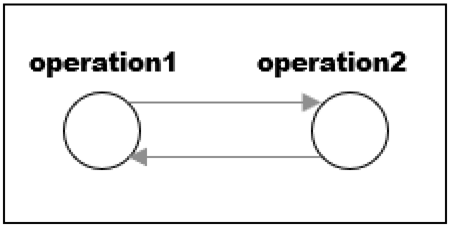
A direct recursion is a recursion where a function calls itself in its own body. For direct recursions, the number of recursions is equal to the number of recursive functions.

Indirect Recursion with One Call Graph Cycle

```
volatile int signal;

void operation1() {
```

```
    int stop = signal%2;
    if(!stop)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is one. The two functions `operation1` and `operation2` are involved in the call graph cycle `operation1` → `operation2` → `operation1`.



An indirect function is a recursion where a function calls itself through other functions. For indirect recursions, the number of recursions can be different from the number of recursive functions.

Multiple Call Graph Cycles Forming One Strongly Connected Component

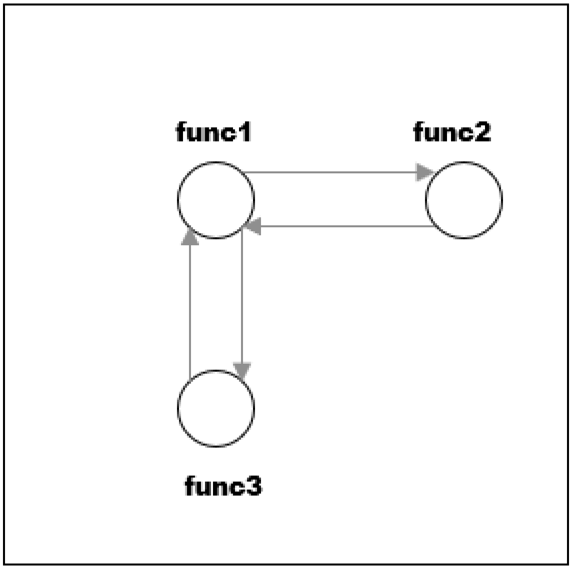
```
volatile int checkStatus;
void func1() {
   if(checkStatus) {
        func2();
   }
   else {
```

```
        func3();
    }
}

func2() {
    func1();
}

func3() {
    func1();
}
```

In this example, there are two call graph cycles:

- `func1` → `func2` → `func1`
- `func1` → `func3` → `func1`

However, the cycles form one strongly connected component. You can move from any vertex to another vertex by following the paths in the graph. Hence, the number of recursions is one.



Indirect Recursion with Two Call Graph Cycles

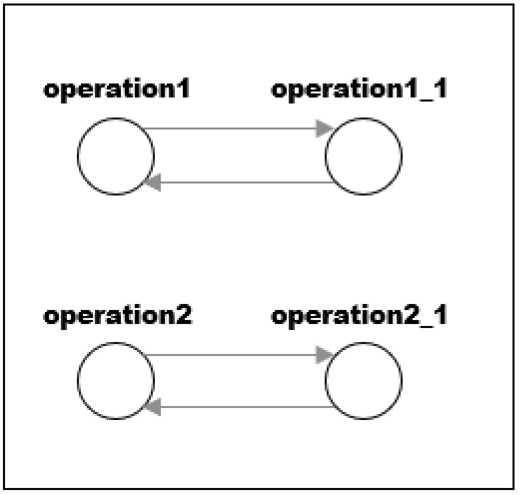
```
volatile int signal;

void operation1() {
    int stop = signal%2;
    if(!stop)
        operation1_1();
}


void operation1_1() {
    operation1();
}

void operation2() {
    int stop = signal%2;
    if(!stop)
        operation2_1();
}

void operation2_1() {
    operation2();
}

void main(){
    operation1();
    operation2();
}
```

In this example, the number of recursions is two.

There are two call graph cycles:

- `operation1` → `operation1_1` → `operation1`
- `operation2` → `operation2_1` → `operation2`

The call graph cycles form two strongly connected components.



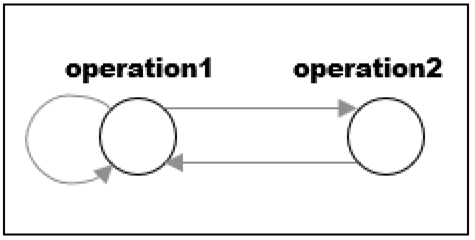
Same Function Called in Direct and Indirect Recursion

```
volatile int signal;

void operation1() {
    int stop = signal%3;
    if(stop==1)
        operation1();
    else if(stop==2)
        operation2();
}

void operation2() {
    operation1();
}

void main() {
    operation1();
}
```

In this example, the number of recursions is two:

- The strongly connected component formed by the cycle `operation1` → `operation2` → `operation1`.

- The self-recursive function `operation1`.



Metric Information

Group: Project
Acronym: `AP_CG_CYCLE`

See Also

Number of Return Statements

Number of `return` statements in a function

Description

This metric measures the number of `return` statements in a function.

The recommended upper limit for this metric is 1. If one return statement is present, when reading the code, you can easily identify what the function returns.

Examples

Function with Return Points

```
int getSign (int arg) {
    if(arg <0)
        return -1;
    else if(arg > 0)
        return 1;
    return 0;
}
```

In this example, `getSign` has 3 `return` statements.

Metric Information

Group: Function
Acronym: RETURN

See Also

Program Maximum Stack Usage

Maximum stack usage in the analyzed program

Description

This metric shows the maximum stack usage from your program.

The metric shows the maximum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of maximum stack usage for a function, see the metric `Maximum Stack Usage`.

Metric Information

Group: Project
Acronym: `PROG_MAX_STACK`

See Also

`Higher Estimate of Local Variable Size` | `Maximum Stack Usage` | `Program Minimum Stack Usage`

Topics

"Determination of Program Stack Usage"

Introduced in R2017b

Program Minimum Stack Usage

Maximum stack usage in the analyzed program taking nested scopes into account

Description

This metric shows the maximum stack usage from your program, taking nested scopes into account.

The metric shows the minimum stack usage for the function with the highest stack usage. If you provide a complete application, the function with the highest stack usage is typically the `main` function because the `main` function is at the top of the call hierarchy. For a description of minimum stack usage for a function, see the metric `Minimum Stack Usage`.

Considering nested scopes is useful for compilers that reuse stack space for variables defined in nested scopes. For instance, in this code, the space for `var_1` is reused for `var_2`.

```
type func (type param_1, ...) {

  {
    /* Scope 1 */
    type var_1, ...;
  }
  {
    /* Scope 2 */
    type var_2, ...;
  }
}
```

Metric Information

Group: Project
Acronym: `PROG_MIN_STACK`

See Also

`Lower Estimate of Local Variable Size` | `Minimum Stack Usage` | `Program Maximum Stack Usage`

Topics

"Determination of Program Stack Usage"

Introduced in R2017b

Polyspace Code Prover Access Functions

cop-docker-agent

(DOS/UNIX) Launch cluster operator to manage Polyspace Access services

Syntax

```
cop-docker-agent [OPTIONS]
```

Description

`cop-docker-agent [OPTIONS]` launches the cluster operator (COP). If you do not specify additional `OPTIONS`, the cluster operator starts on port `8080` and uses the `HTTP` protocol.

Input Arguments

`OPTIONS` — Options to manage the COP or create a node
string

Options to specify and manage the connection settings of the cluster operator (COP) or to create a node.

General options

Option	Description
`--port` *portNumber*	Specify the server port number that you use to access the cluster operator web interface. The default port value is 8080 for HTTP protocol and 8443 for HTTPS.

Option	Description
`--data-dir` *`dirPath`*	Specify the full path to the folder of the `settings.json` file. If the file does not exist, the COP creates it in the specified folder. If the file already exists, the COP reuses its contents to configure the settings. The default folder is the current folder.
`--reset-password`	Reset the password that you use to log into the COP web interface.
`--help`	Display the help menu.

HTTPS configuration options

On Windows® systems, all paths must point to local drives.

Option	Description
`--hostname` *`hostName`*	Specify the host name of the machine where you are running the COP. The host name you specify must match the `Common Name (CN)` you specify to obtain a signed certificate.
`--https-certificate-file` *`absolutePath`*	Specify the absolute path to the HTTPS certificate `PEM` file.
`--https-private-key-file` *`absolutePath`*	Specify the absolute path to the HTTPS private key `PEM` file that you used to generate the certificate.

Option	Description
`--https-trusted-certificates-file` *`absolutePath`*	Specify the full path to the certificate store where you store trusted certificate authorities. For instance, on a Linux® Debian® distribution, `/etc/ssl/certs/ca-certificates.crt` . If you use self-signed certificates, use the same file that you specify for `--https-certificate-file`

New node configuration options

If you choose to install Polyspace Access on multiple machines, use these options to create nodes on the different machines. In the COP settings, for each service, you select the node of the machine on which you want to run the service.

Before you create a node, you must have an instance of the COP already running on at lease one machine. This other machine hosts the `master` node.

Option	Description
`--operator-host` *`hostName:port`*	Specify the host name and port number of the machine hosting the `master` node.
`--node-id` *`nodeName`*	Name of the node that you create. After you start the COP, you see this node listed in the COP web interface on the **Nodes** tab, and in the **Node:** drop-down lists in the **Settings** tab.

Examples

Configure HTTPS protocol with self-signed certificate

The cluster operator (COP) uses HTTP protocol by default. You can encrypt the data between the COP server and client machines by configuring the COP with HTTPS protocol.

Create a self-signed certificate and private key file by using the `openssl` toolkit.

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 365 \
-keyout private_key.pem -out certificate.pem
```

After you enter the command, follow the prompt on the screen. You can leave most fields blank, but you must provide a `Common Name`. The common name must match the host name of the machine running the COP. The command outputs a certificate file `certificate.pem` and a private key file `private_key.pem`.

Start an instance of the COP by using the certificate and private key files that you generated. In the command, specify the full path to the files.

```
./cop-docker-agent --hostname hostName\
--https-certificate-file fullPathTo/certificate.pem \
--https-private-key-file fullPathTo\private_key.pem \
--https-trusted-certificates-file fullPathTo/certificate.pem
```

You can now access the COP web interface from your browser by using `https://`*`hostName`*`:8443`, where *`hostName`* is the host name of the machine running the COP.

Create a node and run Polyspace Access on multiple machines

If you choose to install and run Polyspace Access on multiple machines, you must create nodes on each machine. You associate a Polyspace Access service with a node from the machine on which you run that service.

For instance, suppose you have two machines with host names `host1` and `host2`. You want to run the **Web Server**, **Database**, and **ETL** services on `host2`, and all the other services on `host1`.

From `host1`, start COP on port `8083`.

```
./cop-docker-agent --hostname host1 --port 8083
```

In the COP web interface at `http://host1:8083`, on the **Settings** tab, all the services have their **Node** parameter set to `master`.

Copy the COP binary and associated TAR files from `host1` to `host2`, for instance by using the `scp` command.

```
scp -r jdoe@host1:cop/install/dir ./
```

From host2, navigate to the folder that you copied from host1 and create a node new_node.

```
./cop-docker-agent --hostname host2 --operator-host host1:8083 --node-id new_node &
```

After you run the command, from the COP web interface at http://host1:8083, you see new_node listed in the **Nodes** tab. Click new_node to see a list of data volumes available for this node. If there are no volumes listed, create one.

Go to the **Settings** tab and set the **Node** attribute for the **Web Server**, **Database**, and **ETL** to new_node, then set **Data volume** for the **Database** to one of the new_node volumes.

Save the settings, go to the **Services** tab, click **PROVISION** and then **START ALL**. The **Web Server**, **ETL**, and **Database** services are running on host2, while all the other services run on host1.

See Also

Topics

"Configure and Start the Cluster Operator"
"Configure Polyspace Access Services"

Introduced in R2018b