Programmer's Guide

VERSION
4.0

Borland® C++

Programmer's Guide

Borland® C++

Version 4.0

Borland International, Inc.
100 Borland Way, Scotts Valley, CA 95066-3249

PRINTED IN THE U.S.A.

1E0R1093
9394959697-9876543
W1

Contents

Tables

Figures

Introduction

For an overview of the Borland C++ documentation set read the Introduction in the *User's Guide.*

This manual contains materials for the advanced programmer. If you already know how to program well (whether in C, C++, or another language), this manual is for you. It is a language reference, and provides you with programming information on C++ streams, container classes, persistent streams, inline assembly, and ANSI implementation details.

Typefaces and icons used in these books are described in the *User's Guide.*

What's in this book

Chapters 1-5: Lexical elements, Language structure, C++ specifics, Exception handling, and **The preprocesor**, describe the Borland C++ language. Any extensions to the ANSI C standard are noted in these chapters. These chapters provide a formal language definition, reference, and syntax for both the C and C++ aspects of Borland C++. Some overall information about Chapters 1 through 5 is included in the next section of this introduction.

Chapter 6: Using C++ iostreams tells you how to program input and output using the C++ stream library.

Chapter 7: Using Borland class libraries tells you how to use the Borland C++ persistent streams and container class libraries.

See the *DOS Reference* for information on DOS programming.

Chapter 8: Windows programming explains the basics of programming under Windows.

Chapter 9: Writing dynamic-link libraries explains dynamic-link libraries and dynamic linking.

Chapter 10: Using inline assembly explains how to embed assembly language instructions within your C/C++ code.

Appendix A: ANSI implementation-specific standards describes those aspects of the ANSI C standard that have been left loosely defined or undefined by ANSI. This appendix tells how Borland C++ operates in respect to each of these aspects.

An introduction to the formal definitions

Chapters 1–5 describe the C and C++ languages as implemented in Borland C++. Together, they provide a formal language definition, reference, and syntax for both the C++ and C aspects of Borland C++. They do not provide a language tutorial. We use a modified Backus-Naur form notation to indicate syntax, supplemented where necessary by brief explanations and program examples. The chapters are organized in this manner:

- **Chapter 1: Lexical elements** shows how the lexical tokens for Borland C++ are categorized. It covers the different categories of word-like units, known as *tokens*, recognized by a language.
- **Chapter 2: Language structure** explains how to use the elements of Borland C++. It details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.
- **Chapter 3: C++ specifics** covers language aspects specific to C++.
- **Chapter 4: Exception handling** describes the exception-handling mechanisms available to C and C++ programs.
- **Chapter 5: The preprocessor** covers the preprocessor, including macros, includes, and pragmas, and many other easy yet useful items.

Borland C++ is a full implementation of AT&T's C++ version 3.0 with exception handling, the object-oriented superset of C developed by Bjarne Stroustrup of AT&T Bell Laboratories. This manual refers to AT&T's previous version as C++ 2.1. In addition to offering many new features and capabilities, C++ often veers from C in varying degrees. These differences are noted. All the Borland C++ language features derived from C++ are discussed in greater detail in Chapter 3.

Borland C++ also fully implements the ANSI C standard, with several extensions as indicated in the text. You can set options in the compiler to warn you if any such extensions are encountered. You can also set the compiler to treat the Borland C++ extension keywords as normal identifiers (see Chapter 3 in the *User's Guide*).

There are also "conforming" extensions provided via the **#pragma** directives offered by ANSI C for handling nonstandard, implementation-dependent features.

Syntax and terminology

Syntactic definitions consist of the name of the nonterminal token or symbol being defined, followed by a colon (:). Alternatives usually follow on separate lines, but a single line of alternatives can be used if prefixed by the phrase "one of." For example,

external-definition:
function-definition
declaration

octal-digit: one of
0 1 2 3 4 5 6 7

Optional elements in a construct are printed within angle brackets:

integer-suffix:
unsigned-suffix <long-suffix>

Throughout this manual, the word "argument" is used to mean the actual value passed in a call to a function. "Parameter" is used to mean the variable defined in the function header to hold the value.

C H A P T E R 1

Lexical elements

This chapter provides a formal definition of the Borland C++ lexical elements. It describes the different categories of word-like units (*tokens*) recognized by a language.

The tokens in Borland C++ are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

A Borland C++ program starts as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Borland C++ editor). The basic program unit in Borland C++ is the file. This usually corresponds to a named file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see the discussion starting on page 185). For example, the directive **#include** <*inc_file*> adds (or *includes*) the contents of the file *inc_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

Whitespace

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and *whitespace*. *Whitespace* is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int   i; float f;
```

and

```
int i ;
    float    f;
```

are lexically equivalent and parse identically to give the six tokens:

- **int**
- **i**
- **;**
- **float**
- **f**
- **;**

The ASCII characters representing whitespace can occur within *literal strings,* in which case they are protected from the normal parsing process (they remain as part of the string). For example,

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International".

Line splicing with \

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \
International"
```

is parsed as "Borland International" (see page 18, "String constants," for more information).

Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. Both are supported by Borland C++, with an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

C comments

A C comment is any sequence of characters placed after the symbol pair **/***. The comment terminates at the first occurrence of the pair ***/** following the initial **/***. The entire sequence, including the four comment-delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

See page 191 for a description of token pasting.

Borland C++ does not support the nonportable *token pasting* strategy using **/**/**. Token pasting in Borland C++ is performed with the ANSI-specified pair ##, as follows:

```
#define VAR(i,j)  (i/**/j)     /* won't work */
#define VAR(i,j)  (i##j)       /* OK in Borland C++ */
#define VAR(i,j)  (i ## j)     /* Also OK */
```

In Borland C++,

```
int /* declaration */ i /* counter */;
```

parses as these three tokens:

```
int   i  ;
```

C++ comments

You can also use // to create comments in C code. This is specific to Borland C++.

C++ allows a single-line comment using two adjacent slashes (//). The comment can start in any position, and extends until the next new line:

```
class X {  // this is a comment
... };
```

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/*  int /* declaration */ i /* counter */; */
```

fails, because the scope of the first **/*** ends at the first ***/**. This gives

```
i ; */
```

which would generate a syntax error.

By default, Borland C++ won't allow nested comments, but you can override this with compiler options. See the *User's Guide*, Chapter 3, for information on enabling nested comments.

Delimiters and whitespace

In rare cases, some whitespace before **/*** and **//**, and after ***/**, although not syntactically mandatory, can avoid portability problems. For example, this C++ code

```
int i = j//* divide by k*/k;
+m;
```

parses as `int i = j +m;` not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

Tokens

Borland C++ recognizes six classes of tokens. Here is the formal definition of a token:

token:
    keyword
    identifier
    constant
    string-literal
    operator
    punctuator (also known as separators)

As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, *external* would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al.*

See page 191 for a description of *token pasting.*

Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names. The following tables list the Borland C++ keywords. You can use options to select ANSI keywords only, UNIX keywords, and so on; see the *User's Guide,* Chapters 1 and 3, for information on these options.

If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version.

Note that the keywords **__try** and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword in the C++ exception-handling mechanism. **try** cannot be substituted by **__try**. The keyword **__try** can only be used to match the **__except** or **__finally** keywords. See the discussion of exception handling in Chapter 4 of this book.

Table 1.1
All Borland C++ keywords

__asm
_asm
asm
auto
break
case
catch
__cdecl
_cdecl
cdecl
char
class
const
continue
__cs
_cs
default
delete
do
double
__ds
_ds
else
enum
__es
_es
__except
__export
_export
extern
far
__far
_far
__fastcall
_fastcall
__finally
float
for
friend
goto
huge
__huge
_huge
if
__import
_import
inline
int
interrupt
__interrupt
_interrupt
__loadds
_loadds
long
near
_near
__near
new
operator
__pascal
_pascal
pascal
private
protected
public
register
return
__rtti
__saveregs
_saveregs
__seg
_seg
short
signed
sizeof
__ss
_ss
static
__stdcall
_stdcall
struct
switch
template
this
__thread
throw
__try
try
typedef
union
unsigned
virtual
void
volatile
while

Table 1.2
Borland C++ register pseudovariables

_AH
_AL
_AX
_BH
_BL
_BP
_BX
_CH
_CL
_CS
_CX
_DH
_DI
_DL
_DS
_DX
_EAX[†]
_EBP[†]
_EBX[†]
_ECX[†]
_EDI[†]
_EDX[†]
_ES
_ESI[†]
_ESP
_FLAGS
_FS
_GS[†]
_SI
_SP
_SS

[†] These pseudovariables are always available to the 32-bit compiler. The 16-bit compiler can use these only when you use the option to generate 80386 instructions.

Table 1.3
Borland C++ keyword extensions

__asm
_asm
__cdecl
_cdecl
cdecl
__cs[1]
_cs[1]
__ds[1]
_ds[1]
__es[1]
_es[1]
__except
__export
_export
__far[1]
_far[1]
far[1]
__fastcall
_fastcall
__finally
__huge[1]
_huge[1]
huge[1]
__import[2]
_import[2]
__interrupt[1]
_interrupt[1]
interrupt[1]
__loadds[1]
_loadds[1]
__near[1]
_near[1]
near[1]
__pascal
_pascal
pascal
__saveregs[1]
_saveregs[1]
__seg[1]
__seg[1]
_seg[1]
__ss[1]
_ss[1]
__rtti
__thread[2]
__try

[1] Available only with the 16-bit compilers.
[2] Available only with the 32-bit compilers.

Table 1.4
Keywords specific to C

__finally
__try

Table 1.5
Keywords specific to C++

asm
catch
class
delete
friend
inline
new
operator
private
protected
public
template
this
throw
try
virtual
__rtti

Identifiers

Here is the formal definition of an identifier:

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. Identifiers can contain the letters *a* to *z* and *A* to *Z*, the underscore character "_", and the digits 0 to 9. There are only two restrictions:

- The first character must be a letter or an underscore.

Identifiers in C++ programs are significant to 32 characters.

- By default, Borland C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu and command-line options, but not increased. See the *User's Guide*, Chapters 1 and 3, for information on these options.

Case sensitivity

Borland C++ identifiers are case sensitive, so that *Sum*, *sum*, and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, Borland C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals *Sum* and *sum* are considered identical, resulting in a possible "`Duplicate symbol`" warning during linking.

See the *User's Guide*, Chapters 1 and 3, for information on linking and case-sensitivity options.

An exception to these rules is that identifiers of type **__pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are legal for *different* name spaces regardless of scope. The scope rules are covered on page 29.

Constants

Constants are tokens representing fixed numeric or character values. Borland C++ supports four classes of constants: integer, floating point, character (including strings), and enumeration. Figure 1.1 shows how these types are represented internally.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in Table 1.6.

Integer constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Table 1.7. Note that the rules vary between decimal and nondecimal constants.

Table 1.6: Constants—formal definitions

constant:
floating-constant
integer-constant
enumeration-constant
character-constant

floating-constant:
fractional-constant <exponent-part> <floating-suffix>
digit-sequence exponent-part <floating-suffix>

fractional-constant:
<digit-sequence> . *digit-sequence*
digit-sequence .

exponent-part:
e *<sign> digit-sequence*
E *<sign> digit-sequence*

sign: one of
\+ −

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
f l F L

integer-constant:
decimal-constant <integer-suffix>
octal-constant <integer-suffix>
hexadecimal-constant <integer-suffix>

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
0
octal-constant octal-digit

hexadecimal-constant:
0 x *hexadecimal-digit*
0 X *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit: one of
1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:
unsigned-suffix <long-suffix>
long-suffix <unsigned-suffix>

unsigned-suffix: one of
u U

long-suffix: one of
l L

enumeration-constant:
identifier

character-constant:
c-char-sequence

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
Any character in the source character set except the single-quote ('), backslash (\\), or newline character
escape-sequence.

escape-sequence: one of

`\"`	`\'`	`\?`	`\\`
`\a`	`\b`	`\f`	`\n`
`\o`	`\oo`	`\ooo`	`\r`
`\t`	`\v`	`\Xh...`	`\xh...`

Decimal

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10;  /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0;   /*decimal 0 = octal 0 */
```

Octal

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces the constant to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul, lu, UL,* and so on.

Table 1.7
Borland C++ integer constants without L or U

Decimal constants	
0 to 32,767	**int**
32,768 to 2,147,483,647	**long**
2,147,483,648 to 4,294,967,295	**unsigned long**
> 4294967295	truncated
Octal constants	
00 to 077777	**int**
010000 to 0177777	**unsigned int**
02000000 to 017777777777	**long**
020000000000 to 037777777777	**unsigned long**
> 037777777777	truncated
Hexadecimal constants	
0x0000 to 0x7FFF	**int**
0x8000 to 0xFFFF	**unsigned int**
0x10000 to 0x7FFFFFFF	**long**
0x80000000 to 0xFFFFFFFF	**unsigned long**
> 0xFFFFFFFF	truncated

The data type of a constant in the absence of any suffix (*U*, *u*, *L*, or *l*) is the first of the following types that can accommodate its value:

Decimal	**int, long int, unsigned long int**
Octal	**int, unsigned int, long int, unsigned long int**
Hexadecimal	**int, unsigned int, long int, unsigned long int**

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int**, **unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int**, **unsigned long int** that can accommodate its value.

If the constant has both *u* and *l* suffixes (*ul*, *lu*, *Ul*, *lU*, *uL*, *Lu*, *LU*, or *UL*), its data type will be **unsigned long int**.

Table 1.7 summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Floating-point constants

A floating constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)
- Type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Here are some examples:

Constant	Value
23.45e6	23.45×10^6
.0	0
0.	0
1.	$1.0 \times 10^0 = 1.0$

Constant	Value
–1.23	–1.23
2e-5	2.0×10^{-5}
3E+10	3.0×10^{10}
.09E34	0.09×10^{34}

In the absence of any suffixes, floating-point constants are of type **double**. However, you can coerce a floating constant to be of type **float** by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The next table shows the ranges available for **float**, **double**, and **long double**.

Table 1.8
Borland C++ floating constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4×10^{-38} to 3.4×10^{38}
double	64	1.7×10^{-308} to 1.7×10^{308}
long double	80	3.4×10^{-4932} to 1.1×10^{4932}

Character constants

A *character constant* is one or more characters enclosed in single quotes, such as `'A'`, `'='`, or `'\n'`. In C, single-character constants have data type **int**. The number of bits used to internally represent a character constant is **sizeof(int)**. In a 16-bit program, the upper byte is zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

To compare sizes of character types, compile this as a C program and then as a C++ program.

```
#include <stdio.h>
#define CH 'x'       /* A CHARACTER CONSTANT */
void main(void) {
   char ch = 'x';    /* A char VARIABLE      */

   printf("\nSizeof int     = %d", sizeof(int) );
   printf("\nSizeof char    = %d", sizeof(char) );
   printf("\nSizeof ch      = %d", sizeof(ch) );
   printf("\nSizeof CH      = %d", sizeof(CH) );
   printf("\nSizeof wchar_t = %d", sizeof(wchar_t) );
   }
```

Table 1.9
Sizes of character types

Sizes are in bytes.

Output when compiled as C program:			Output when compiled as C++ program:		
	16-bit	32-bit		16-bit	32-bit
`Sizeof int`	= 2	4	`Sizeof int`	= 2	4
`Sizeof char`	= 1	1	`Sizeof char`	= 1	1
`Sizeof ch`	= 1	1	`Sizeof ch`	= 1	1
`Sizeof CH`	= 2	4	`Sizeof CH`	= 1	1
`Sizeof wchar_t`	= 2	2	`Sizeof wchar_t`	= 2	2

The three char types

One-character constants, such as `'A'`, `'\t'`, and `'\007'`, are represented as **int** values. In this case, the low-order byte is *sign extended* into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to –1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned**, which forces the high bit to be zero regardless of the value of the low bit. See the *User's Guide*, Chapters 1 and 3, for information on these options.

The three character types, **char**, **signed char**, and **unsigned char**, require an 8-bit (one byte) storage. In C and Borland C++ programs prior to version Borland C++ 4.0, **char** is treated the same as **signed char**. The behavior of C programs is unaffected by the distinction between the three character types.

To retain the old behavior, use the **–K2** command-line option and Borland C++ 3.1 header files.

In a C++ program, a function can be overloaded with arguments of type **char**, **signed char**, or **unsigned char**. For example, the following function prototypes are valid and distinct:

```
void func(char ch);
void func(signed char ch);
void func(unsigned char ch);
```

If only one of the above prototypes exists, it will accept any of the three character types. For example, the following is acceptable:

```
void func(unsigned char ch);
void main(void) {
   signed char ch = 'x';
   func(ch);
   }
```

See the *User's Guide*, Chapters 1 and 3, for a description of code-generation options.

Escape sequences

The backslash character (\) is used to introduce an *escape sequence*, which allows the visual representation of certain nongraphic characters. For example, the constant **\n** is used for the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, `'\03'` for *Ctrl-C* or `'\x3F'` for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for Borland C++). Larger numbers generate the compiler error `Numeric constant too large`. For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Originally, Turbo C allowed only three digits in a hexadecimal escape sequence. The ANSI C rules adopted in Borland C++ might cause problems with old code that assumes only the first three characters are converted. For example, using Turbo C 1.*x* to define a string with a bell (ASCII 7) followed by numeric characters, a programmer might write:

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, Borland C++ compiles it as the hexadecimal number \x0072 and the literal string ".1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The next table shows the available escape sequences.

Table 1.10
Borland C++ escape sequences

The \\ must be used to represent a real ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)

Table 1.10: Borland C++ escape sequences (continued)

\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O = a string of up to three octal digits
\xH		any	H = a string of hex digits
\XH		any	H = a string of hex digits

Wide-character constants

Wide-character types can be used to represent a character that does not fit into the storage space allocated for a **char** type. A wide character is stored in a two-byte space. A character constant preceded immediately by an *L* is a wide-character constant of data type *wchar_t* (defined in stddef.h). For example:

```
wchar_t ch = L'AB';
```

A string preceded immediately by an *L* is a wide-character string. The memory allocation for a string is two bytes per character. For example:

```
wchar_t str = L"ABCD";
```

Multi-character constants

Borland C++ also supports multi-character constants. When using the 32-bit compiler, multi-character constants can consist of as many as four characters. The 16-bit compiler is restricted to two-character constants. For example, 'An', '\n\t', and '\007\007' are acceptable in a 16-bit program. The constant, '\006\007\008\009' is valid only in a 32-bit program. When using the 16-bit compiler, these constants are represented as 16-bit **int** values with the first character in the low-order byte and the second character in the high-order byte. For 32-bit compilers, multi-character constants are always 32-bit **int** values. These constants are not portable to other C compilers.

String constants

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of-**char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written `""`.

The characters inside the double quotes can include escape sequences (see page 15). This code, for example,

```
"\t\t\"Name\"\\\tAddress\n\n"
```

prints out like this:

```
                "Name"\        Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The `\"` provides interior double quotes.

If you compile with the **–A** option for ANSI compatibility, the escape character sequence “\\“, is translated to “\“ by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single `'\0'` character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

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

#pragma argsused

int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdParam,
                    int nCmdShow )
{
   char    *p;

   _InitEasyWin();

   p = "This is an example of how Borland C++"
    " will automatically\ndo the concatenation for"
    " you on very long strings,\nresulting in nicer"
    " looking programs.";
   printf(p);
   return(0);
}
```

The output of the program is:

```
This is an example of how Borland C++ will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character in order to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

Enumeration constants

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed.

See page 74 for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, cubs, and *dodgers* are enumeration constants of type *team* that can be assigned to any variables of type *team* or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

Constants and internal representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For Borland C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80*x*86 microprocessors governs the choices of internal representations for the various data types.

The next table lists the sizes and resulting ranges of the data types for Borland C++; see page 40 for more information on these data types. Figure 1.1 shows how these types are represented internally.

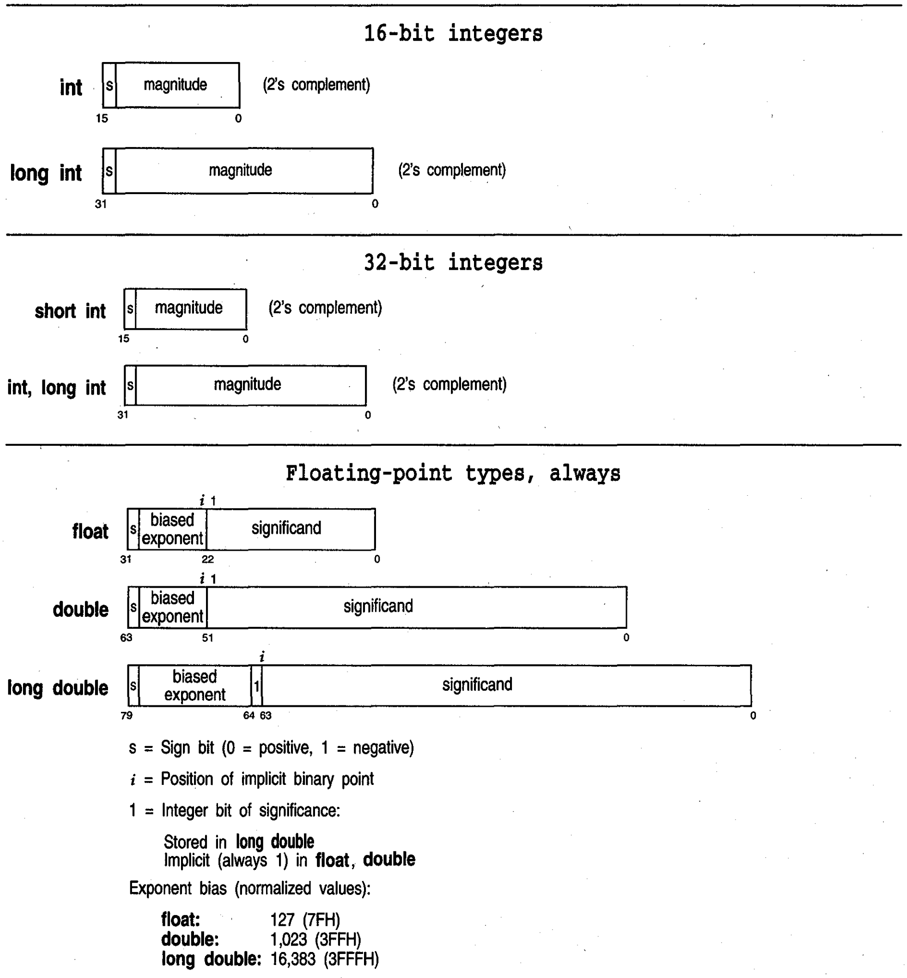
Table 1.11: 16-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	–128 to 127	Very small numbers and ASCII characters
enum	16	–32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	–32,768 to 32,767	Counting, small numbers, loop control
int	16	–32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	–2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4×10^{-38} to 3.4×10^{38}	Scientific (7-digit precision)
double	64	1.7×10^{-308} to 1.7×10^{308}	Scientific (15-digit precision)
long double	80	3.4×10^{-4932} to 1.1×10^{4932}	Financial (19-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

Table 1.12: 32-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	–128 to 127	Very small numbers and ASCII characters
short int	16	–32,768 to 32,767	Counting, small numbers, loop control
unsigned int	32	0 to 4,294,967,295	Larger numbers and loops
int	32	–2,147,483,648 to 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
enum	32	–2,147,483,648 to 2,147,483,647	Ordered sets of values
long	32	–2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	3.4×10^{-38} to 3.4×10^{38}	Scientific (7-digit precision)
double	64	1.7×10^{-308} to 1.7×10^{308}	Scientific (15-digit precision)
long double	80	3.4×10^{-4932} to 1.1×10^{4932}	Financial (19-digit precision)
near pointer	32	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

Figure 1.1 Internal representations of numerical types



Constant expressions

A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is

constant-expression:
    Conditional-expression

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- Assignment
- Comma
- Decrement
- Function call
- Increment

Punctuators

The punctuators (also known as separators) in Borland C++ are defined as follows:

punctuator: one of

```
[ ] ( ) { } , ; : ... * = #
```

Brackets

[ ] (open and close brackets) indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];          /* 3 x 4 matrix */
ch = str[3];            /* 4th element */
   ⋮
```

Parentheses

( ) (open and close parentheses) group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);     /* override normal precedence */

if (d == z) ++x;     /* essential with conditional statement */

func();              /* function call, no args */
int (*fptr)();       /* function pointer declaration */
fptr = func;         /* no () means func pointer */

void func2(int n);   /* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered in the "Expressions" section starting on page 77.

Braces

{ } (open and close braces) indicate the start and end of a compound statement:

```
if (d == z)
{
   ++x;
   func();
}
```

The closing brace serves as a terminator for the compound statement, so a **;** (semicolon) is not required after the **}**, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
   {};                  /*illegal semicolon*/
else
```

Comma

The comma (,) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j);                           /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5));  /* also calls func with two args! */
```

Semicolon

The semicolon (;) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, Borland C++ might ignore it.

```
a + b;    /* maybe evaluate a + b, but discard value */
++a;      /* side effect on a, but discard value of ++a */
;         /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement*:

```
for (i = 0; i < n; i++)
{
   ;
}
```

Colon

Use the colon (:) to indicate a labeled statement:

```
start:    x=0;
   ⋮
goto start;
```

Labels are discussed in the "Labeled statements" section on page 102.

The use of the colon in class initialization is shown in the section beginning on page 144.

Ellipsis

The ellipsis (...) is three successive periods with no whitespace intervening. Ellipses are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that *func* will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.

In C++, you can omit the comma preceding the ellipsis.

Asterisk (pointer declaration)

The * (asterisk) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr;  /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;         /* a pointer to an integer array */
double ***double_ptr;  /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer)

The = (equal sign) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int  x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }  /* Parameter i has default value of zero */
```

The equal sign is also used as the assignment operator in expressions:

```
int a, b, c;
a = b + c;
float *ptr = (float *) malloc(sizeof(float) * 100);
```

Pound sign (preprocessor directive)

The # (pound sign) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See page 185 for more on the preprocessor directives.

and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

CHAPTER 2

Language structure

This chapter provides a formal definition of Borland C++'s language structure. It describes the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

Declarations

This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Objects

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely "points" to the object. The type is used

- To determine the correct memory allocation required initially.
- To interpret the bit patterns found in the object during subsequent accesses.
- In many type-checking situations, to ensure that illegal assignments are trapped.

Borland C++ supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in various memory models.

The Borland C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Borland C++ can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this rule (known as *forward references*) are labels, calls to undeclared functions, and class, struct, or union tags.

lvalues

An *lvalue* is an object locator: an expression that designates an object. An example of an lvalue expression is **P*, where *P* is any expression evaluating to a non-null pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the *l* stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if *a* and *b* are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as *a = 1;* and *b = a + b* are legal.

rvalues

The expression *a + b* is not an lvalue: *a + b = a* is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

Storage classes and types

Associating identifiers with objects requires each identifier to have at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Borland C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type; see page 99 for more on this operator.

Scope

The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

- **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.

- **Function**. The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label_name:* followed by a statement. Label names must be unique within a function.

- **Function prototype**. Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.

- **File**. File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.

- **Class (C++)**. For now, think of a class as a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see pages 124–138.

Name spaces

Name space is the scope within which an identifier must be unique. C uses four distinct classes of identifiers:

- **goto** label names. These must be unique within the function in which they are declared.
- Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all tags defined externally.
- Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
- Variables, **typedef**s, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Structures, classes, and enumerations are in the same name space in C++.

Visibility

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Visibility cannot exceed scope, but scope can exceed visibility.

```
   ⋮
{
   int i; char ch;  // auto by default
   i = 3;           // int i and char ch in scope and visible
   ⋮
   {
      double i;
      i = 3.0e3;    // double i in scope and visible
                    // int i=3 in scope but hidden
      ch = 'A';     // char ch in scope and visible
   }
                    // double i out of scope
   i += 1;          // int i visible and = 4
   ⋮
// char ch still in scope & visible = 'A'
}
   ⋮
// int i and char ch out of scope
```

Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be accessed under certain conditions.

Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedef**s and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

Static

Memory is allocated to objects with *static* duration as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Don't confuse static duration with file or global scope. An object can have static duration and local scope.

Local

Local duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects must always have local or function scope. The storage class specifier **auto** can be used when declaring local duration variables, but is usually redundant, because **auto** is the default for variables declared within a block. An object with local duration also has local scope, because it does not *exist* outside of its enclosing block. The converse is not true: a local scope object can have static duration.

The Borland C++ compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. Borland C++ can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Dynamic

Dynamic duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the heap, using either standard library functions such as *malloc*, or by using the C++ operator **new**. The corresponding deallocations are made using *free* or **delete**.

Translation units

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration
  function-definition
  declaration

For more details, see "External declarations and definitions" on page 37.

The word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the following section, "Linkage.") Any declaration that also reserves storage for an object or function is called a definition (or defining declaration).

Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function within one file only. Identifiers with *no linkage* represent unique entities.

Here are the external and internal linkage rules:

- Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.

  For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external linkage. In C, it will have internal linkage.
- If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
- If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
- If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

The following identifiers have no linkage attribute:

- Any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
- Function parameters
- Block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .OBJ files compiled with a C compiler.

To tell the C++ compiler not to mangle the name of a function, declare the function as `extern "C"`, like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler that references to the function *Cfunc* should not be mangled.

You can also apply the `extern "C"` declaration to a block of names:

```
extern "C" {
   void Cfunc1( int );
   void Cfunc2( int );
   void Cfunc3( int );
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions *Cfunc1*, *Cfunc2*, and *Cfunc3* should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {
   #include "locallib.h"
};
```

Declaration syntax

All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations.*

Declarations can be *defining declarations* (also known as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Tentative definitions

The ANSI C standard introduces a new concept: that of the *tentative definition.* Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;
int x;          /*legal, one copy of x is reserved */

int y;
int y = 4;      /* legal, y is initialized to 4 */

int z = 5;
int z = 6;      /* not legal, both are initialized definitions */
```

Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

Possible declarations

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use **typedefs** to improve legibility.

Table 2.1
Borland C++
declaration syntax

declaration:
<decl-specifiers> <declarator-list>;
asm-declaration
function-declaration
linkage-specification

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
friend (C++ specific)
typedef

decl-specifiers:
<decl-specifiers> decl-specifier

storage-class-specifier:
auto
register
static
extern

function-specifier: (C++ specific)
inline
virtual

type-specifier:
simple-type-name
class-specifier
enum-specifier
elaborated-type-specifier
const
volatile

simple-type-name:
class-name
***typedef**-name*
char
short
int
long
signed
unsigned
float
double
void

elaborated-type-specifier:
class-key identifier
class-key class-name
enum *enum-name*

class-key: (C++ specific)
class
struct
union

enum-specifier:
enum *<identifier>* { *<enum-list>* }

enum-list:
enumerator
enumerator-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

constant-expression:
conditional-expression

linkage-specification: (C++ specific)
extern *string* { *<declaration-list>* }
extern *string declaration*

declaration-list:
declaration
declaration-list ; *declaration*

In Table 2.2, note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail starting on page 48.

Table 2.2: Borland C++ declarator syntax

declarator-list:
init-declarator
declarator-list , *init-declarator*

init-declarator:
declarator <*initializer*>

declarator:
dname
modifier-list
pointer-operator declarator
declarator (*parameter-declaration-list*) <*cv-qualifier-list*>
(The <cv-qualifier-list> is for C++ only.)
declarator [<*constant-expression*>]
(*declarator*)

modifier-list:
modifier
modifier-list modifier

modifier:
__cdecl
__pascal
__interrupt
__near
__far
__huge

pointer-operator:
* <*cv-qualifier-list*>
& <*cv-qualifier-list*> (C++ specific)
class-name **::** * <*cv-qualifier-list*> (C++ specific)

cv-qualifier-list:
cv-qualifier <*cv-qualifier-list*>

cv-qualifier
const
volatile

dname:
name
class-name (C++ specific)
~ *class-name* (C++ specific)
type-defined-name

type-name:
type-specifier <*abstract-declarator*>

abstract-declarator:
pointer-operator <*abstract-declarator*>
<*abstract-declarator*> (*argument-declaration-list*) <*cv-qualifier-list*>
<*abstract-declarator*> [<*constant-expression*>]
(*abstract-declarator*)

argument-declaration-list:
<*arg-declaration-list*>
arg-declaration-list , **...**
<*arg-declaration-list*> **...** (C++ specific)

arg-declaration-list:
argument-declaration
arg-declaration-list , *argument-declaration*

argument-declaration:
decl-specifiers declarator
decl-specifiers declarator = *expression* (C++ specific)
decl-specifiers <*abstract-declarator*>
decl-specifiers <*abstract-declarator*> = *expression* (C++ specific)

function-definition:
<*decl-specifiers*> *declarator* <*ctor-initializer*> *function-body*

function-body:
compound-statement

initializer:
= *expression*
= { *initializer-list* }
(*expression-list*) (C++ specific)

initializer-list:
expression
initializer-list , *expression*
{ *initializer-list* <,> }

External declarations and definitions

The storage class specifiers **auto** and **register** cannot appear in an external declaration (see page 32). For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of **sizeof**), then exactly one external definition of that identifier must be somewhere in the entire program.

Borland C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];                // no size
struct mystruct;        // tag only, no member declarators
   :
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
     int i, j;
};                      // add member declarators
```

Table 2.3 covers class declaration syntax. In the section on classes (beginning on page 124), you can find examples of how to declare a class. The "Referencing" section on page 116 covers C++ reference types (closely related to pointer types) in detail. Finally, see page 160 for a discussion of **template**-type classes.

Table 2.3: Borland C++ class declaration syntax (C++ only)

class-specifier:
class-head { <*member-list*> }

class-head:
class-key <*identifier*> <*base-specifier*>
class-key class-name <*base-specifier*>

member-list:
member-declaration <*member-list*>
access-specifier : <*member-list*>

member-declaration:
<*decl-specifiers*> <*member-declarator-list*> ;
function-definition <;>
qualified-name ;

member-declarator-list:
member-declarator
member-declarator-list, *member-declarator*

member-declarator:
declarator <*pure-specifier*>
<*identifier*> : *constant-expression*

pure-specifier:
= 0

base-specifier:
: *base-list*

base-list:
base-specifier
base-list , *base-specifier*

base-specifier:
class-name
virtual <*access-specifier*> *class-name*
access-specifier <**virtual**> *class-name*

access-specifier:
private
protected
public

conversion-function-name:
operator *conversion-type-name*

conversion-type-name:
type-specifiers <*pointer-operator*>

constructor-initializer:
: *member-initializer-list*

member-initializer-list:
member-initializer
member-initializer , *member-initializer-list*

Table 2.3: Borland C++ class declaration syntax (C++ only) (continued)

member-initializer:	**+**	**–**	*****	**/**	**%**	**^**
class name (<*argument-list*>)	**&**	**|**	**~**	**!**	**=**	**< >**
identifier (<*argument-list*>)	**+=**	**–=**	***=**	**/=**	**%=**	**^=**
operator-function-name:	**&=**	**|=**	**<<**	**>>**	**>>=**	**<<=**
operator *operator-name*	**==**	**!=**	**<=**	**>=**	**&&**	**||**
	++	**– –**	**,**	**–>***	**–>**	**( )**
operator-name: one of	**[ ]**	**.***				
new delete sizeof typeid						

Type specifiers

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i;                  // declare i as a signed integer
unsigned char ch1, ch2; // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++, a missing type specifier can lead to syntactic ambiguity, so C++ practice requires you to explicitly declare all **int** type specifiers.

Type categories

The four basic type categories (and their subcategories) are as follows:

- Aggregate
    - Array
    - **struct**
    - **union**
    - **class (C++ only)**
- Function
- Scalar
    - Arithmetic
    - Enumeration
    - Pointer
    - Reference (C++ only)
- **void** (discussed in the next section)

Types can also be viewed in another way: they can be *fundamental* or *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.

A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes.

Given any nonvoid type ***type*** (with some provisos), you can declare derived types as follows:

Table 2.4
Declaring types

Declaration	**Description**
type *t*;	An object of type ***type***.
type *array*[10];	Ten ***type***s: *array*[0] – *array*[9].
type **ptr*;	*ptr* is a pointer to ***type***.
type &*ref* = *t*;	*ref* is a reference to ***type*** (C++).
type *func*(**void**);	*func* returns value of type ***type***.
void *func1*(***type*** *t*);	*func1* takes a type ***type*** parameter.
struct st {***type*** *t1*; ***type*** *t2*};	structure **st** holds two ***type***s.

type& var, type &var, and type & var are all equivalent.

Type void

void is a special type specifier indicating the absence of any values. It is used in the following situations:

C++ handles *func* in a special manner. See page 62 and code examples on page 64.

- When there is an empty parameter list in a function declaration:

```
int func(void);   // func takes no arguments
```

- When the declared function does not return a value:

```
void func(int n); // return value
```

- As a generic pointer (a pointer to **void** is a generic pointer to anything):

```
void *ptr;        // ptr can later be set to point to any object
```

- In *typecasting* expressions:

```
extern int errfunc();         // returns an error code
 :
(void) errfunc();             // discard return value
```

The fundamental types

The fundamental type specifiers are built from the following keywords:

char
double
float
int
long
short
signed
unsigned

From these keywords you can build the integral and floating-point types, which are together known as the *arithmetic* types. The modifiers **long, short,**

signed, and **unsigned** can be applied to the integral types. The include file limits.h contains definitions of the value ranges for all the fundamental types.

Integral types

char, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. Table 2.5 shows the integral type specifiers, with synonyms listed on the same line.

Table 2.5
Integral types

These synonyms are not valid in C++. See page 16.

char, signed char	Synonyms if default **char** set to **signed**.
unsigned char	
char, unsigned char	Synonyms if default **char** set to **unsigned**.
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Only **signed** or **unsigned** can be used with **char**, **short**, **int**, or **long**. The keywords **signed** and **unsigned**, when used on their own, mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Borland C++ lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares *ch* as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

Only **long** or **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to indicate that **short**, **int**, and **long** form a nondecreasing sequence with "**short** <= **int** <= **long**." All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In a Borland C++ 16-bit program, the types **int** and **short** are equivalent, both being 16 bits. In a Borland C++ 32-bit program, the types **int** and **long** are equivalent, both being 32 bits. The signed varieties are all stored in two's complement format using the most significant bit (MSB) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown on page 21). In the unsigned versions, all bits are used to give a range of $0 - (2^n - 1)$, where n is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. Borland C++ uses the IEEE floating-point formats. Appendix A tells more about implementation-specific items.

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double** *test_case*, for example.

The table on page 21 indicates the storage allocations for the floating-point types.

Standard conversions

When you use an arithmetic expression, such as $a + b$, where a and b are different arithmetic types, Borland C++ performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps Borland C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in the next table. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers), or they are of type **double, float**, or **long double.**
2. If either operand is of type **long double**, the other operand is converted to **long double.**
3. Otherwise, if either operand is of type **double,** the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long,** the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Table 2.6
Methods used in standard arithmetic conversions

Type	Converts to	Method
char	**int**	Zero or sign-extended (depends on default **char** type)
unsigned char	**int**	Zero-filled high byte (always)
signed char	**int**	Sign-extended (always)
short	**int**	Same value; sign extended
unsigned short	**unsigned int**	Same value; zero filled
enum	**int**	Same value

Special char, int, and enum conversions

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

The conversions discussed in this section are specific to Borland C++.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

Initialization

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

If the object has automatic storage duration, its value is indeterminate.

- To zero if it is an arithmetic type
- To null if it is a pointer type

The syntax for initializers is as follows:

initializer
  = expression
  = {initializer-list} <,>}
  (expression list)
initializer-list
  expression
  initializer-list, expression
  {initializer-list} <,>}

The rules governing initializers are

- The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
- The item to be initialized must be an object (for example, an array) of unknown size.
- For C (not required for C++), all expressions must be constants if they appear in one of these places:
  - In an initializer for an object that has static duration.
  - In an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed).
- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list (as described in the following section).
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, which counts how many times each day of the week appears in a month (assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

The following rules initialize character arrays and wide character arrays:

- You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for *name*[0]), 'n' (for *name*[1]), and so on (and including a null terminator).

- You can initialize a wide character array (one that is compatible with *wchar_t*) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
   int i;
   char str[21];
   double d;
} s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces.

Declarations and declarators

A *declaration* is a list of names. The names are sometimes referred to as *declarators* or *identifiers*. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon.

Simple declarations of variable identifiers have the following pattern:

data-type var1 <=init1>, var2 <=init2>, ...;

where *var1, var2,...* are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.

In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions.

See Table 2.1 on page 36 for the declarator syntax. The definition covers both identifier and function declarators.

The format of the declarator indicates how the declared *name* is to be interpreted when used in an expression. If ***type*** is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

storage-class-specifier ***type*** *D1, D2;*

indicates that each occurrence of *D1* or *D2* in an expression will be treated as an object of type ***type*** and storage class *storage class specifier*. The type of the *name* embedded in the declarator will be some phrase containing ***type,*** such as "***type,***" "pointer to ***type,***" "array of ***type,***" "function returning ***type,***" or "pointer to function returning ***type,***" and so on.

For example, in the following table of declarations each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Table 2.7: Declaration syntax examples

Declarator syntax	Implied *type* of *name*	Example
type *name*;	***type***	**int** count;
type *name*[];	(open) array of ***type***	**int** count[];
type *name*[3];	Fixed array of three elements, all of ***type*** (*name*[0], *name*[1], and *name*[2])	**int** count[3];
type **name*;	Pointer to ***type***	**int** *count;
type **name*[];	(open) array of pointers to ***type***	**int** *count[];
type *(name[]);	Same as above	**int** *(count[]);
type (**name*) [];	Pointer to an (open) array of ***type***	**int** (*count) [];
type &name;	Reference to ***type*** (C++ only)	**int** &count;
type *name*();	Function returning ***type***	**int** count();
type **name*();	Function returning pointer to ***type***	**int** *count();
type *(name());	Same as above	**int** *(count());
type (**name*) ();	Pointer to function returning ***type***	**int** (*count) ();

Note the need for parentheses in (**name*)[] and (**name*)(); this is because the precedence of both the array declarator **[ ]** and the function declarator **( )** is higher than the pointer declarator *. The parentheses in *(*name*[]) are optional.

Use of storage class specifiers

A *storage class specifier* (also called a type specifier) must be present in a declaration. The storage class specifiers can be one of the following: **auto**, **extern**, **register**, **static**, or **typedef**.

auto

The storage class specifier **auto** is used only with local scope variable declarations. It conveys local (automatic) duration, but since this is the default for all local scope variable declarations, its use is rare.

extern

The storage class specifier **extern** can be used with function and variable file scope and local scope declarations to indicate external linkage. With file scope variables, the default storage class specifier is **extern**. When used with variables, **extern** indicates that the variable has static duration. (Remember that functions always have static duration.) See page 33 for information on using **extern** to prevent name mangling when combining C and C++ code.

register

The Borland C++ compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

The storage class specifier **register** is allowed only for local variable and function parameter declarations. It is equivalent to **auto**, but it makes a request to the compiler to allocate the variable to a register if possible. The allocation of a register can significantly reduce the size and improve the performance of programs in many situations. However, since Borland C++ does a good job of placing variables in registers, it is rarely necessary to use the **register** keyword.

Borland C++ lets you request register variable options. See the *User's Guide*, Chapter 3, for a discussion of compiling optimizations including register allocation, and passing **this** pointer with __fastThis.

See "The **__fastcall** modifier" section on page 55 for a discussion of passing function parameters in registers.

static

The storage class specifier **static** can be used with function and variable file scope and local scope declarations to indicate internal linkage. **static** also indicates that the variable has static duration. In the absence of constructors or explicit initializers, static variables are initialized with 0 or null.

In C++, a static data member of a class has the same value for all instances of a class. A static member function of a class can be invoked independently of any class instance.

typedef

The keyword **typedef** indicates that you are defining a new data type specifier rather than declaring an object. **typedef** is included as a storage class specifier because of syntactical rather than functional similarities.

```
static  long int biggy;
typedef long int BIGGY;
```

The first declaration creates a 32-bit, **long int**, static-duration object called *biggy*. The second declaration establishes the identifier *BIGGY* as a new type specifier, but does not create any run-time object. *BIGGY* can be used in any subsequent declaration where a type specifier would be legal. Here's an example:

```
extern BIGGY salary;
```

has the same effect as

```
extern long int salary;
```

Although this simple example can be achieved by `#define BIGGY long int`, more complex **typedef** applications achieve more than is possible with textual substitutions.

Important! **typedef** does not create new data types; it merely creates useful mnemonic synonyms or aliases for existing types. It is especially valuable in simplifying complex declarations:

```
typedef double (*PFD)();
PFD array_pfd[10];
/* array_pfd is an array of 10 pointers to functions returning double */
```

You can't use **typedef** identifiers with other data-type specifiers:

```
unsigned BIGGY pay;      /* ILLEGAL */
```

Variable modifiers

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier/object mapping. The modifiers available with Borland C++ are summarized in Table 2.8 and discussed in the following sections.

Table 2.8: Borland C++ modifiers

Modifier	Use with	Description
const†	Variables	Prevents changes to object.
volatile†	Variables	Prevents register allocation and some optimization. Warns compiler that object might be subject to outside change during evaluation.

Table 2.8: Borland C++ modifiers (continued)

Borland C++ extensions		
__cdecl††	Functions	Forces C argument-passing convention. Affects Linker and link-time names.
__cdecl††	Variables	Forces global identifier case-sensitivity and leading underscores.
__interrupt	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt handlers.
__pascal	Functions	Forces Pascal argument-passing convention. Affects Linker and link-time names.
__pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
__near, **__far**, **__huge**	Pointer types	Overrides the default pointer type specified by the current memory model.
__cs, **__ds**, **__es**, **__seg**, **__ss**	Pointer types	Segment pointers.
__near, **__far**, **__huge**	Functions	Overrides the default function type specified by the current memory model.
__near, **__far**	Variables	Directs the placement of the object in memory.
__export	Functions/classes	Tells the compiler which functions or classes to export.
__import	Functions/classes	Tells the compiler which functions or classes to import. (In 16-bit programs, this keyword can be used only for class declarations.)
__loadds	Functions	Sets DS to point to the current data segment.
__saveregs	Functions	Preserves all register values (except for return values) during execution of the function.
__fastcall	Functions	Forces register parameter passing convention. Affects the linker and link-time names.
__stdcall	Functions	Forces the standard WIN32 argument-passing convention.

† C++ extends **const** and **volatile** to include classes and member functions.

†† This is the default.

const

The modifier **const** used by itself is equivalent to **const int.**

The **const** modifier prevents any assignments to the object or any other side effects, such as increment or decrement. A **const** pointer cannot be modified, though the object to which it points can be. Consider the following examples:

```
const  float  pi      = 3.1415926;
const         maxint  = 32767;
char   *const str     = "Hello, world"; // A constant pointer
char   const  *str2   = "Hello, world"; /* A pointer to a constant char */
```

Given these, the following statements are illegal:

```
pi  = 3.0;            /* Assigns a value to a const */
i   = maxint++;       /* Increments a const */
str = "Hi, there!";   /* Points str to something else */
```

Note, however, that the function call `strcpy(str,"Hi, there!")` is legal, because it does a character-by-character copy from the string literal "Hi, there!" into the memory locations pointed to by *str*.

In C++, **const** also hides the **const** object and prevents external linkage. You need to use **extern const**. A pointer to a **const** can't be assigned to a pointer to a non-**const** (otherwise, the **const** value could be assigned to using the non-**const** pointer). Here's an example:

```
char *str3 = str2   /* disallowed */
```

Only **const** member functions can be called for a **const** object.

volatile

In C++, **volatile** has a special meaning for class member functions. If you've declared a **volatile** object, you can use only its **volatile** member functions.

The **volatile** modifier indicates that the object can be modified; not only by you, but also by something outside of your program, such as an interrupt routine or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions containing it, because the value could change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile  int  ticks;
void _ _interrupt timer() {
   ticks++;
}

void wait(int interval) {
   ticks = 0;
   while (ticks < interval);    // Do nothing
}
```

These routines (assuming *timer* has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop, since the loop does not change the value of *ticks*.

Mixed-language calling conventions

Borland C++ allows your programs to easily call routines written in other languages, and vice versa. When you mix languages like this, you have to deal with two important issues: identifiers and parameter passing.

The section beginning on page 32 tells how to use **extern**, which allows C names to be referenced from a C++ program.

By default, Borland C++ saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you have can select the **–u–** command-line option, or uncheck the compiler option setting in the IDE.

The following table summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the *caller*) or the called function (the *callee*) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.

Table 2.9
Calling conventions

Modifier	Push parameters	Pop parameters	Name change
_ _cdecl†	Right first	Caller	'_' prepended
_ _fastcall	Left first	Callee	'@' prepended
_ _pascal	Left first	Callee	Uppercase
_ _stdcall	Right first	Callee	No change

† This is the default.

_ _cdecl

You might want to ensure that certain identifiers have their case preserved and keep the underscore on the front, especially if they're C identifiers in a separate file. You can do so by declaring those identifiers to be **_ _cdecl**. (This also has an effect on parameter passing for functions).

main( ) must be declared as **_ _cdecl**; this is because the C start-up code always tries to call *main( )* with the C calling convention.

Like **_ _pascal**, the **_ _cdecl** modifier is specific to Borland C++. It is used with functions and pointers to functions. It overrides the compiler directives and IDE options and allows a function to be called as a regular C function. For example, if you were to compile the previous program with the Pascal calling option set but wanted to use *printf*, you might do something like this:

```
extern _ _cdecl printf(const char *format, ...); // NOT REQUIRED IF YOU INCLUDE
                                                    stdio.h
void putnums(int i, int j, int k);

void _ _cdecl main()
{
   putnums(1,4,9);
}

void putnums(int i, int j, int k)
{
   printf("And the answers are:  %d, %d, and %d\n",i,j,k);
}
```

If you compile a program with Pascal calling conventions, all functions (except those with variable parameters) used from the run-time library will need to use the **_ _ cdecl** modifier. Any function that uses variable parameters must be declared with the **_ _cdecl** modifier. Every function in the Borland C++ run-time libraries is properly defined in anticipation of this.

_ _pascal

In Pascal, global identifiers are not saved in their original case, nor are underscores prepended to them. Borland C++ lets you declare any identifier to be of type **_ _pascal**; the identifier is converted to uppercase, and no underscore is prepended. (If the identifier is a function, this also affects the parameter-passing sequence used; see the section on page 54 for more details.)

The **_ _pascal** modifier is specific to Borland C++; it is intended for functions (and pointers to functions) that use the Pascal parameter-passing sequence. Also, functions declared to be of type **_ _pascal** can still be called from C routines, as long as the C routine sees that the function is of type **_ _pascal**.

```
_ _pascal putnums(int i, int j, int k)
{
   printf("And the answers are:  %d, %d, and %d\n",i,j,k);
}
```

Functions of type **_ _pascal** cannot take a variable number of arguments, unlike functions such as *printf*. For this reason, you cannot use an ellipsis (...) in a **_ _pascal** function definition.

Most of the 16-bit Windows API functions are **_ _pascal** functions. Most Win32 API functions are **_ _stdcall** functions.

Multithread variables

32

The keyword _ _**thread** is used in multithread programs to preserve a unique copy of global and static class-variables. Each program thread maintains a private copy of a _ _**thread** variable for each threaded process.

The syntax is ***Type***_ _**thread** *var_name*. For example, `int _ _thread x;` declares an integer type that will be global but private to each thread in the program in which the statement occurs.

The _ _**thread** modifier can be used with global (file-scope) and static variables. The modifier cannot be used with pointers or functions. (However, you can have pointers to _ _**thread** objects.) A program element that requires run-time initialization or run-time finalization cannot be declared to be a _ _**thread** type. The following declarations require run-time initialization and are therefore illegal:

```
int f();
int _ _thread x = f();  // Illegal
```

Instantiation of a class with a user-defined constructor or destructor requires run-time initialization and is therefore illegal.

```
class X { X(); ~X(); }
X _ _thread myclass;        // Illegal
```

Pointer modifiers

16

Borland C++ has modifiers that affect the pointer declarator (*); that is, they modify pointers to data. These are _ _**near**, _ _**far**, _ _**huge**, _ _**cs**, _ _**ds**, _ _**es**, _ _**seg**, and _ _**ss**.

You can compile a program using one of several memory models. The model you use determines (among other things) the internal format of pointers. For example, if you use a small data model (small or medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact or large), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes when you're using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the pointer modifiers.

See the discussion in Chapter 8 for an in-depth explanation of _ _**near**, _ _**far**, and _ _**huge** pointers, and a description of normalized pointers. The chapter also presents additional discussions of _ _**cs**, _ _**ds**, _ _**es**, _ _**seg**, and _ _**ss**.

Function modifiers

Tiny and huge memory models are not supported.

This section presents descriptions of the Borland C++ function modifiers.

In addition to their use as pointer modifiers, the **__near**, **__far**, and **__huge** modifiers can also be used as function type modifiers; that is, they can modify functions and function pointers as well as data pointers. In addition, you can use the **__loadds**, **__export**, **__import**, and **__saveregs** modifiers to modify functions.

See also Section "Class memory model specifications" beginning page 125.

In a 16-bit program, the **__import** can be used only as a modifier for class declarations. In 32-bit programs the keyword can be applied to class, function, and variable declarations.

The **__near**, **__far**, and **__huge** function modifiers can be combined with **__cdecl** or **__pascal**, but not with **__interrupt**.

Functions of type **__huge** are useful when interfacing with code in assembly language that doesn't use the same memory allocation as Borland C++.

A function that is not an **__interrupt** type can be declared to be **__near**, **__far**, or **__huge** in order to override the default settings for the current memory model.

A **__near** function uses **__near** calls; a **__far** or **__huge** function uses **__far** call instructions.

In the small and compact memory models, an unqualified function defaults to type **__near**. In the medium and large models, an unqualified function defaults to type **__far**.

A **__huge** function is the same as a **__far** function, except that the DS register is set to the data segment address of the source module when a **__huge** function is entered, but left unset for a **__far** function.

The **__export** modifier makes the function exportable from Windows. The **__import** modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL; see page 248 of Chapter 8 for details.

The **__loadds** modifier indicates that a function should set the DS register, just as a **__huge** function does, but does not imply **__near** or **__far** calls. Thus, **__loadds __far** is equivalent to **__huge**.

The **__saveregs** modifier causes the function to preserve all register values and restore them before returning (except for explicit return values passed in registers such as AX or DX).

The **__loadds** and **__saveregs** modifiers are useful for writing low-level interface routines, such as mouse support routines.

Functions declared with the **__fastcall** modifier have different names than their non-**__fastcall** counterparts. The compiler prefixes the **__fastcall** function name with an `@`. This prefix applies to both unmangled C function names and to mangled C++ function names.

__interrupt functions

The **__interrupt** modifier is specific to Borland C++. **__interrupt** functions are designed to be used with the 8086/8088 interrupt vectors. Borland C++ will compile an **__interrupt** function with extra function entry and exit code so that registers AX, BX, CX, DX, SI, DI, ES, and DS are preserved. The other registers (BP, SP, SS, CS, and IP) are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function will use an **iret** instruction to return, so that the function can be used to service hardware or software interrupts. Here is an example of a typical **__interrupt** declaration:

```
void __interrupt myhandler();
```

You should declare interrupt functions to be of type **void**. **__interrupt** functions can be declared in any memory model. For all memory models, DS is set to the program data segment.

The __fastcall modifier

The 16-bit compiler does not support **__fastcall** with virtual functions.

You can request the Borland C++ compiler to use registers for parameter passing. Such a request is made by using the **__fastcall** function modifier, or by selecting compiler optimization __fastThis. See the *User's Guide*, Chapter 3, for a discussion of __fastThis.

The compiler treats this calling convention as a language specifier, along the lines of **__cdecl** and **__pascal**. Functions declared with either of these two languages modifiers cannot also have the **__fastcall** modifier since they use the stack to pass parameters. Likewise, the **__fastcall** modifier cannot be used together with **__loadds**. The compiler generates a warning if you try to mix functions of these types.

Pointers

See pages 86 and 98 for discussions of referencing and dereferencing.

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain Borland C++ operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Pointers to objects

A pointer of type "pointer to object of ***type***" holds the address of (that is, points to) an object of ***type***. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

The size of pointers to objects is dependent on the memory model and the size and disposition of your data segments, possibly influenced by the optional pointer modifiers (discussed starting on page 53).

Pointers to functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called "pointer to function returning ***type***," where ***type*** is the function's return type. For example,

```
void (*func)();
```

In C++, this is a pointer to a function taking no arguments, and returning **void**. In C, it's a pointer to a function taking an unspecified number of arguments and returning **void**. In this example,

```
void (*func)(int);
```

func* is a pointer to a function taking an **int argument and returning **void**.

For C++, such a pointer can be used to access static member functions. Pointers to class members must use pointer-to-member operators. See page 98.

Pointer declarations

See page 40 for details on **void**.

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. Borland C++ lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to **void**. And in C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer.

If ***type*** is any predefined or user-defined type, including **void**, the declaration

Warning! You need to initialize pointers before using them.

```
type *ptr;    /* Uninitialized pointer */
```

declares *ptr* to be of type "pointer to ***type***." All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

The pointer type "pointer to **void**" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any "pointer to ***type***" value, including null, without complaint. Assignments without proper casting between a "pointer to ***type1***" and a "pointer to ***type2***," where ***type1*** and ***type2*** are different types, can invoke a compiler warning or error. If ***type1*** is a function and ***type2*** isn't (or vice versa), pointer assignments are illegal. If ***type1*** is a pointer to **void**, no cast is needed. Under C, if ***type2*** is a pointer to **void**, no cast is needed.

Assignment restrictions also apply to pointers of different sizes (**_ _near**, **_ _far**, and **_ _huge**). You can assign a smaller pointer to a larger one without error, but you can't assign a larger pointer to a smaller one unless you are using an explicit cast. For example,

```
char _ _near *ncp;
char _ _far  *fcp;
char _ _huge *hcp;
fcp = ncp;                    // legal
hcp = fcp;                    // legal
```

```
fcp = hcp;                   // not legal
ncp = fcp;                   // not legal
ncp = (char _ _near*)fcp;    // now legal
```

Pointer constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```
int i;                       // i is an int

int * pi;                    // pi is a pointer to int (uninitialized)

int * const cp = &i;         // cp is a constant pointer to int

const int ci = 7;            // ci is a constant int

const int * pci;             // pci is a pointer to constant int

const int * const cpc = &ci; // cpc is a constant pointer to a
                             // constant int
```

The following assignments are legal:

```
i = ci;                     // Assign const-int to int

*cp = ci;                   // Assign const-int to
                            // object-pointed-at-by-a-const-pointer

++pci;                      // Increment a pointer-to-const

pci = cpc;                  // Assign a const-pointer-to-a-const to a
                            // pointer-to-const
```

The following assignments are illegal:

```
ci = 0;                     // NO--cannot assign to a const-int

ci--;                       // NO--cannot change a const-int

*pci = 3;                   // NO--cannot assign to an object
                            // pointed at by pointer-to-const

cp = &ci;                   // NO--cannot assign to a const-pointer,
                            // even if value would be unchanged

cpc++;                      // NO--cannot change const-pointer

pi = pci;                   // NO--if this assignment were allowed,
                            // you would be able to assign to *pci
                            // (a const value) by assigning to *pi.
```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

Pointer arithmetic

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

The difference between two pointers has meaning only if both pointers point into the same array.

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to ***type***" automatically take into account the size of ***type***; that is, the number of bytes needed to store a ***type*** object.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to ***type***, adding an integral value to the pointer advances the pointer by that number of objects of ***type***. If ***type*** has size 10 bytes, then adding an integer 5 to a pointer to ***type*** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of `ptr2 - ptr1` would be 7.

When an integral value is added to or subtracted from a "pointer to ***type***," the result is also of type "pointer to ***type***."

There is no such element as "one past the last element," of course, but a pointer is allowed to assume such a value. If P points to the last array element, $P + 1$ is legal, but $P + 2$ is undefined. If P points to one past the last array element, $P - 1$ is legal, giving a pointer to the last element. However, applying the indirection operator * to a "pointer to one past the last element" leads to undefined behavior.

Informally, you can think of $P + n$ as advancing the pointer by (n * **sizeof(*type*)**) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff_t* defined in stddef.h (**signed long** for **_ _huge** and **_ _far** pointers; **signed int** for all others). This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff_t*. In the expression $P1 - P2$, where $P1$ and $P2$ are of type pointer to ***type*** (or pointer to qualified ***type***), $P1$ and $P2$ must point to existing elements or to one past the last element. If $P1$ points to the i-th element, and $P2$ points to the j-th element, $P1 - P2$ has the value $(i - j)$.

Pointer conversions

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (***type****) will convert a pointer to type "pointer to ***type***." See page 109 for a discussion of C++ typecast mechanisms.

C++ reference declarations

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See page 116 for complete details.

Arrays

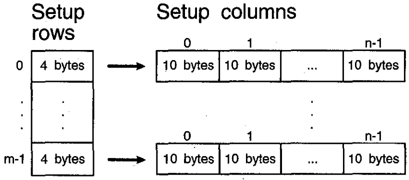
The declaration

type *declarator* [<*constant-expression*>]

declares an array composed of elements of ***type***. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implemention is for three rows and five columns but it can be very easily modified to accept run-time user input.



See the *Library Reference*, Chapter 3, for a description of *calloc*, *free*, and *printf*.

```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>

typedef long double TYPE;
typedef TYPE **OBJECT;

unsigned int rows = 3, columns = 5;

void de_allocate(OBJECT);
```

```
int main(void) {
   OBJECT matrix;
   unsigned int i, j;

   /* STEP 1: SET UP THE ROWS. */
   matrix = (OBJECT) calloc( rows, sizeof(TYPE *));

   /* STEP 2: SET UP THE COLUMNS. */
   for (i = 0; i < rows; ++i)
       matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));

     for (i = 0; i < rows; i++)
         for (j = 0; j < columns; j++)
             matrix[i][j] = i + j;          /* INITIALIZE */

   for (i = 0; i < rows; ++i) {
       printf("\n\n");
       for (j = 0; j < columns; ++j)
       printf("%5.2Lf", matrix[i][j]);
       }
   de_allocate(matrix);
   return 0;
   }

void de_allocate(OBJECT x) {
   int i;

   for (i = 0; i < rows; i++)      /* STEP 1: DELETE THE COLUMNS. */
       free(x[i]);

   free(x);                        /* STEP 2: DELETE THE ROWS. */
   }
```

This code produces the following output:

```
0.00 1.00 2.00 3.00 4.00

1.00 2.00 3.00 4.00 5.00

2.00 3.00 4.00 5.00 6.00
```

In certain contexts, the first array declarator of a series might have no expression inside the brackets. Such an array is of indeterminate size. This is legitimate in contexts where the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array; neither does an array function parameter. As a special extension to ANSI C, Borland C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

Functions

Functions are central to C and C++ programming. Languages such as Pascal distinguish between procedure and function. For C and C++, functions play both roles.

Declarations and definitions

Each program must have a single external function named *main* marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see page 32).

Functions are defined in your source files or made available by linking precompiled libraries.

In C++ you must always use function prototypes. We recommend that you also use them in C.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide Borland C++ with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.)

Declarations and prototypes

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows:

In C++, this declaration means <***type***> *func(**void**)*

<***type***> *func*()

where ***type*** is the optional return type defaulting to **int**. A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

You can enable a warning within the IDE or with the command-line compiler: "`Function called without a prototype.`"

This problem was eased by the introduction of function prototypes with the following declaration syntax:

<***type***> *func*(*parameter-declarator-list*);

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */

foo()
{
   int limit = 32;
   char ch = 'A';

   long mval;

   mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for *lmax*, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to *lmax*. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to *lmax* would not match in size or content what *lmax* was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function *strcpy* takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```

In C++, *func()* also declares a function taking no arguments.

stdarg.h and varargs.h contain macros that you can use in user-defined functions with variable numbers of parameters.

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as *printf*), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Here are some more examples of function declarators and prototypes:

```
int  f();                      /* In C, a function returning an int with no
                                  information about parameters. This is the K&R
                                  "classic style." */

int f();                       /* In C++, a function taking no arguments */

int  f(void);                  /* A function returning an int that takes no
                                  parameters. */

int  p(int,long);              /* A function returning an int that accepts two
                                  parameters: the first, an int; the second, a
                                  long. */

int  _ _pascal q(void);        /* A pascal function returning an int that takes
                                  no parameters at all. */

char _ _far *s(char *source, int kind); /* A function returning a far pointer to
                                           a char and accepting two parameters:
                                           the first, a pointer to a char; the
                                           second, an int. */

int  printf(char *format,...); /* A function returning an int and accepting a
                                  pointer to a char fixed parameter and any
                                  number of additional parameters of unknown
                                  type. */

int  (*fp)(int);               /* A pointer to a function returning an int and
                                  accepting a single int parameter. */
```

Definitions

Table 2.10 gives the general syntax for external function definitions.

Table 2.10 External function definitions

file
  external-definition
  file external-definition

external-definition:
  function-definition
  declaration
  asm-statement

Table 2.10: External function definitions (continued)

function-definition:
<declaration-specifiers> declarator <declaration-list>
compound-statement

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

You can mix elements from 1 and 2.

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **__pascal**, **__cdecl**, **__export**, **__interrupt**, **__near**, **__far**, **__huge**, **__loadds**, **__saveregs**. The defaults depend on the memory model and compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is `func(void)`. The old style of *func* is legal in C but antiquated and possibly unsafe.
6. A function body representing the code to be executed when the function is called.

Formal parameter declarations

The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) {                   // no args

int func(T1 t1, T2 t2, T3 t3=1) {  // three simple parameters, one
                                   // with default argument

int func(T1* ptr1, T2& tref) {     // A pointer and a reference arg

int func(register int i) {         // Request register for arg

int func(char *str,...) {          /* One string arg with a variable number of
         other args, or with a fixed number of args with varying types */
```

In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, or enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all have automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal parameter declarators.

Function calls and argument conversions

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how Borland C++ deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, Borland C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in the section "Standard conversions," starting on page 42. When a function prototype is in scope, Borland C++ converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Borland C++ converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Important! If your function prototype does not match the actual function definition, Borland C++ will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

Structures

Structure initialization is discussed on page 43.

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The Borland C++ structure type lets you handle complex data structures almost as easily as single variables.

In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example,

```
struct mystruct { ... }; // mystruct is the structure tag
   ⋮
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

Untagged structures and typedefs

Untagged structure and union members are ignored during initialization.

If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere:

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];       // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, arry[20];
```

Usually, you don't need both a tag and a **typedef**: either can be used in structure declarations.

Structure member declarations

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in Table 2.2 on page 37.

A structure member can be of any type, with two exceptions:

- The member type cannot be the same as the **struct** type being currently declared:

  You can omit the **struct** keyword in C++.

  ```
  struct mystruct { mystruct s } s1, s2; // illegal
  ```

  However, a member can be a pointer to the structure being declared, as in the following example:

  ```
  struct mystruct { mystruct *ps } s1, s2; // OK
  ```

  Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.
- Except in C++, a member cannot have the type "function returning...," but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

Structures and functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void);  // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s);        // directly
void func2(mystruct *sptr);    // via a pointer
void func3(mystruct &sref);    // as a reference (C++ only)
```

Structure member access

Structure and union members are accessed using the following two selection operators:

- **.** (period)
- **->** (right arrow)

Suppose that the object *s* is of struct type *S*, and *sptr* is a pointer to *S*. Then if *m* is a member identifier of type *M* declared in *S*, the expressions *s.m* and *sptr->m* are of type *M*, and both represent the member object *m* in *S*. The expression *sptr->m* is a convenient synonym for `(*sptr).m`.

The operator **.** is called the direct member selector and the operator **->** is called the indirect (or pointer) member selector. For example:

```
struct mystruct
{
   int i;
   char str[21];
   double d;
```

```
} s, *sptr = &s;
    ⋮
s.i = 3;            // assign to the i member of mystruct s
sptr -> d = 1.23;   // assign to the d member of mystruct s
```

The expression *s.m* is an lvalue, provided that *s* is an lvalue and *m* is not an array type. The expression *sptr->m* is an lvalue unless *m* is an array type.

If structure *B* contains a field whose type is structure *A*, the members of *A* can be accessed by two applications of the member selectors:

```
struct A {
   int j;
   double x;
};

struct B {
   int i;
   struct A a;
   double d;
} s, *sptr;
    ⋮
s.i = 3;            // assign to the i member of B
s.a.j = 2;          // assign to the j member of A
sptr->d = 1.23;     // assign to the d member of B
(sptr->a).x = 3.14  // assign to x member of A
```

Each structure declaration introduces a unique structure type, so that in

```
struct A {
   int i,j;
   double d;
} a, a1;

struct B {
   int i,j;
   double d;
} b;
```

the objects *a* and *a1* are both of type struct *A*, but the objects *a* and *b* are of different structure types. Structures can be assigned only if the source and destination have the same type:

```
a = a1;   // OK: same type, so member by member assignment
a = b;    // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign member-by-member */
```

Structure word alignment

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```
struct mystruct {
   int i;
   char str[21];
   double d;
} s;
```

Word alignment is off by default.

the object *s* occupies sufficient memory to hold a 2-byte integer for a 16-bit program, or a 4-byte integer for a 32-bit program, a 21-byte string, and an 8-byte **double**. The format of this object in memory is determined by selecting the word alignment option. Without word alignment, *s* will be allocated 31 contiguous bytes (by the 16-bit compiler) or 33 contiguous bytes (by the 32-bit compiler).

If you turn on word alignment, Borland C++ pads the structure with bytes to ensure the structure is aligned as follows:

16

1. The structure will start on a word boundary (even address).
2. Any non-**char** member will have an even byte offset from the start of the structure.
3. A final byte is added (if necessary) at the end to ensure that the whole structure contains an even number of bytes.

32

1. The structure boundaries are defined by 4-byte multiples.
2. For any non-**char** member, the offset will be a multiple of the member size. A **short** will be at an offset that is some multiple of 2 **int**s from the start of the structure.
3. One to three bytes can be added (if necessary) at the end to ensure that the whole structure contains a 4-byte multiple.

For the 16-bit compiler, with word alignment on, the structure would therefore have a byte added before the **double**, making a 32-byte object.

For the 32-bit compiler, with word alignment on, three bytes would be added before the **double**, making a 36-byte object.

Structure name spaces

Structure tag names share the same name space with union tags and enumeration tags (but **enum**s within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example,

```
goto s;
   ⋮
s:          // Label
struct s {  // OK: tag and label name spaces different
   int s;   // OK: label, tag and member name spaces different
   float s; // ILLEGAL: member name duplicated
} s;        // OK: var name space different. In C++, this can only
            // be done if s does not have a constructor.

union s {   // ILLEGAL: tag space duplicate
   int s;   // OK: new member space
   float f;
} f;        // OK: var name space

struct t {
   int s;   // OK: different member space
   ⋮
} s;        // ILLEGAL: var name duplicate
```

Incomplete declarations

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before *A* has been declared:

```
struct A;                 // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of *A* is called *incomplete* because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of *B* doesn't need the size of *A*.

Bit fields

A structure can contain any mixture of bit-field and non-bit-field types.

You can declare **signed** or **unsigned** integer members as bit fields from 1 to 16 bits wide. You specify the bit-field width and optional identifier as follows:

type-specifier <*bitfield-id*> : *width*;

where *type-specifier* is **char**, **unsigned char**, **int**, or **unsigned int**. Bit fields are allocated from low-order to high-order bits within a word. The expression *width* must be present and must evaluate to a constant integer in the range 1 to 16.

If the bit field identifier is omitted, the number of bits specified in *width* is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused. For example,

```
struct mystruct {
   int      i : 2;
   unsigned j : 5;
   int        : 4;
   int      k : 1;
   unsigned m : 4;
} a, b, c;
```

produces the following layout:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td colspan="4">←→</td><td>↔</td><td colspan="4">←→</td><td colspan="5">←→</td><td colspan="2">←→</td></tr>
<tr><td colspan="4">m</td><td>k</td><td colspan="4">(unused)</td><td colspan="5">j</td><td colspan="2">i</td></tr>
</table>

Integer fields are stored in two's-complement form, with the leftmost bit being the MSB (most significant bit). With **int** (for example, **signed**) bit fields, the MSB is interpreted as a sign bit. A bit field of width 2 holding binary 11, therefore, would be interpreted as 3 if **unsigned**, but as –1 if **int**. In the previous example, the legal assignment `a.i = 6` would leave binary 10 = –2 in *a.i* with no warning. The signed **int** field *k* of width 1 can hold only the values –1 and 0, because the bit pattern 1 is interpreted as –1.

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (**.** and **->**) used for non-bit-field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent.

The expression &*mystruct.x* is illegal if *x* is a bit field identifier, because there is no guarantee that *mystruct.x* lies at a byte address.

Unions

Unions correspond to the variant record types of Pascal and Modula-2.

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
   int i;
   double d;
   char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union** *myunion*, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time.

sizeof(union *myunion*) and **sizeof**(*mu*) both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (**.** and **->**), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d);     // OK: displays mu.d = 4.016
printf("mu.i = %d\n",mu.i);     // peculiar result
mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch);  // OK: displays mu.ch = A
printf("mu.d = %f\n",mu.d);     // peculiar result
muptr->i = 3;
printf("mu.i = %d\n",mu.i);     // OK: displays mu.i = 3
```

The second *printf* is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

Anonymous unions (C++ only)

A union that doesn't have a tag and is not used to declare a named object (or other type) is called an *anonymous union*. It has the following form:

```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the `x.y` or `p->y` syntax.

Anonymous unions can't have member functions and at file level must be declared static. In other words, an anonymous union cannot have external linkage.

Union declarations

The general declaration syntax for unions is similar to that for structures. The differences are

- Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (*Note that, because bit fields are machine dependent, they can pose problems when writing portable code.*)
- Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
- Unions can be initialized only through their first declared member:

```
union local87 {
   int i;
   double d;
   } a = { 20 };
```

- A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.

Enumerations

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable *anyday* of this type, and a set of enumerators (*sun, mon,...*) with constant integer values.

Borland C++ is free to store enumerators in a single byte when `Treat enums as ints` is unchecked (O | C | Code Generation) or the **–b** flag is used. The default is on (meaning **enum**s are always **int**s) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, that is the type of each enumerator.

In C, a variable of an enumerated type can be assigned any value of type **int**—no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;         // OK
anyday = 1;           // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```

In C++, you can omit the **enum** keyword if **days** is not the name of anything else in the same scope.

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

See page 20 for more on enumeration constants.

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on).

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* Initializer expression can include previously declared enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
             quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;         // OK
*daysptr = anyday;    // OK
mon = tues;           // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int mon = 11;
{
   enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
   /* enumerator mon hides outer declaration of int mon */
   struct days { int i, j;};   // ILLEGAL: days duplicate tag
   double sat;                 // ILLEGAL: redefinition of sat
}
mon = 12;                      // back in int mon scope
```

In C++, enumerators declared within a class are in the scope of that class.

In C++ it is possible to overload most operators for an enumeration. However, because the **=**, **[ ]**, **( )**, and **–>** operators must be overloaded as member functions, it is not possible to overload them for an **enum**. The following example shows how to overload the postfix and prefix increment operators.

```
// OVERLOAD THE POSTFIX AND PREFIX INCREMENT OPERATORS FOR enum
#include <iostream.h>

enum _SEASON { spring, summer, fall, winter };

_SEASON operator++(_SEASON &s) {      // PREFIX INCREMENT
   _SEASON tmp = s;                   // SAVE THE ORIGINAL VALUE

   // DO MODULAR ARITHMETIC AND CAST THE RESULT TO _SEASON TYPE
   s = _SEASON( (s + 1) % 4 );        // INCREMENT THE ORIGINAL
   return tmp;                        // RETURN THE OLD VALUE
   }

  // UNNAMED int ARGUMENT IS NOT USED
_SEASON operator++(_SEASON &s, int) { // POSTFIX INCREMENT
   switch (s) {
      case spring: s = summer; break;
      case summer: s = fall; break;
      case fall:   s = winter; break;
      case winter: s = spring; break;
      }
   return (s);
   }
```

```
int main(void) {
   _SEASON season = fall;

   cout << "\nThe season is " << season;
   cout << "\nSeason is unchanged: " << ++season;
   cout << "\nFinally:" << season++;
   return 0;
   }
```

Expressions

Table 2.12 (on page 78) shows how identifiers and operators are combined to form grammatically legal "phrases."

The standard conversions are detailed in Table 2.6 on page 43.

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Table 2.12, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by Borland C++ (see "Evaluation order" on page 80).

Expressions can produce an lvalue, an rvalue, or no value. Expressions might cause side effects whether they produce a value or not.

The precedence and associativity of the operators are summarized in Table 2.11. The grammar in Table 2.12 on page 78 completely defines the precedence and associativity of the operators.

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator category in the following table is indicated by its order in the table. The first category (the first line) has the highest precedence.

Table 2.11
Associativity and precedence of Borland C++ operators

Operators	**Associativity**
( ) [ ] -> :: .	Left to right
! ~ + − ++ −− & * (*typecast*)	Right to left
sizeof new delete typeid	Right to left
.* ->*	Left to right
* / %	Left to right
+ −	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?: (conditional expression)	Right to left
= *= /= %= += −= &= ^= |= <<= >>=	Right to left
,	Left to right

Table 2.12: Borland C++ expressions

primary-expression:
  literal
  this (C++ specific)
  :: *identifier* (C++ specific)
  :: *operator-function-name* (C++ specific)
  ::*qualified-name* (C++ specific)
  (*expression*)
  name

literal:
  integer-constant
  character-constant
  floating-constant
  string-literal

name:
  identifier
  operator-function-name (C++ specific)
  conversion-function-name (C++ specific)
  ~ *class-name* (C++ specific)
  qualified-name (C++ specific)

qualified-name: (C++ specific)
  qualified-class-name :: *name*

postfix-expression:
  primary-expression
  postfix-expression [*expression*]
  postfix-expression (<*expression-list*>)
  simple-type-name (<*expression-list*>) (C++ specific)
  postfix-expression . *name*
  postfix-expression -> *name*
  postfix-expression ++
  postfix-expression −−

Table 2.12: Borland C++ expressions (continued)

const_cast < *type-id* > **(** *expression* **)** (C++ specific)
dynamic_cast < *type-id* > **(** *expression* **)** (C++ specific)
reinterpret_cast < *type-id* > **(** *expression* **)** (C++ specific)
static_cast < *type-id* > **(** *expression* **)** (C++ specific)
typeid (*expression* **)** (C++ specific)
typeid (*type-name* **)** (C++ specific)

expression-list:
assignment-expression
expression-list **,** *assignment-expression*

unary-expression:
postfix-expression
++ *unary-expression*
– – *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name* **)**
allocation-expression (C++ specific)
deallocation-expression (C++ specific)

unary-operator: one of
& ***** **+** **–** **~** **!**

allocation-expression: (C++ specific)
<::> **new** <*placement*> *new-type-name* <*initializer*>
<::> **new** <*placement*> (*type-name*) <*initializer*>

placement: (C++ specific)
(*expression-list*)

new-type-name: (C++ specific)
type-specifiers <*new-declarator*>

new-declarator: (C++ specific)
ptr-operator <*new-declarator*>
new-declarator [<*expression*>]

deallocation-expression: (C++ specific)
<::> **delete** *cast-expression*
<::> **delete** [ ] *cast-expression*

cast-expression:
unary-expression
(*type-name*) *cast-expression*

pm-expression:
cast-expression
pm-expression **.*** *cast-expression* (C++ specific)
pm-expression **–>*** *cast-expression* (C++ specific)

multiplicative-expression:
pm-expression
multiplicative-expression ***** *pm-expression*
multiplicative-expression **/** *pm-expression*
multiplicative-expression **%** *pm-expression*

additive-expression:
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **–** *multiplicative-expression*

shift-expression:
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

relational-expression:
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

equality-expression:
relational-expression
equality expression **==** *relational-expression*
equality expression **!=** *relational-expression*

AND-expression:
equality-expression
AND-expression **&** *equality-expression*

exclusive-OR-expression:
AND-expression
exclusive-OR-expression **^** *AND-expression*

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

logical-OR-expression:
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

conditional-expression:
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of

= ***=** **/=** **%=** **+=** **–=**
<<= **>>=** **&=** **^=** **|=**

expression:
assignment-expression
expression **,** *assignment-expression*

constant-expression:
conditional-expression

Expressions and C++

C++ allows the overloading of certain standard C operators, as explained starting on page 152. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator **==** might be defined in class *complex* to test the equality of two complex numbers without changing its normal usage with non-class data types.

An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the C language rules for operators and conversions might not apply to expressions in C++.

Evaluation order

The order in which Borland C++ evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression

```
i = v[i++];  // i is undefined
```

The value of *i* depends on whether *i* is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for *sum* and *total*. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value.

Borland C++ regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression.

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables a, b, c, and f, then the expression $f = a + (b + c)$ forces $(b + c)$ to be evaluated before adding the result to a.

Errors and overflows

See *_matherr* and *signal* in the *Library Reference.*

Table 2.11 (on page 78) summarizes the precedence and associativity of the operators. During the evaluation of an expression, Borland C++ can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2^n arithmetic on n-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.

Operator semantics

The Borland C++ operators described here are the standard ANSI C operators.

Unless the operators are overloaded, the following information is true in both C and C++. In C++ you can overload all of these operators with the exception of **.** (member access operator), **?:** (conditional operator), **::** and **.*** (scope access operators).

If an operator is overloaded, the discussion might not be true for it anymore. Table 2.12 on page 78 gives the syntax for all operators and operator expressions.

Operator descriptions

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Borland C++ is especially rich in operators, offering not only the common arithmetical and logical operators, but also many for bit-level manipulations, structure and union component access, and pointer operations (referencing and dereferencing).

Overloading is discussed starting on page 149.

C++ extensions offer additional operators for accessing class members and their objects, together with a mechanism for overloading operators. *Overloading* lets you redefine the action of any standard operators when applied to the objects of a given class. In this section, the discussion is confined to the standard operator definitions.

After defining the standard operators, data types and declarations are discussed and an explanation is provided about how these affect the actions of each operator. Then the syntax for building expressions from operators, punctuators, and object is provided.

The operators # and ## are used only by the preprocessor (see page 185).

The operators in Borland C++ are defined as follows:

operator: one of

```
[ ]      ( )     .      ->     ++     --
&        *       +      -      ~      !
sizeof   /       %      <<     >>     <
>        <=      >=     ==     !=     ^
|        &&      ||     ?:     =      *=
/=       %=      +=     -=     <<=    >>=
&=       ^=      |=     ,      #      ##
```

The following operators are specific to C++:

:: .* ->*

Except for **[ ]**, **( )**, and **?:**, which bracket expressions, the multicharacter operators are considered as single tokens. The same operator token can have more than one interpretation, depending on the context. For example,

`A * B`	Multiplication
`*ptr`	Dereference (indirection)
`A & B`	Bitwise AND
`&A`	Address operation
`int &`	Reference modifier (C++)
`label:`	Statement label
`a ? x : y`	Conditional statement
`void func(int n);`	Function declaration
`a = (b+c)*d;`	Parenthesized expression
`a, b, c;`	Comma expression
`func(a, b, c);`	Function call
`a = ~b;`	Bitwise negation (one's complement)
`~func() {delete a;}`	Destructor (C++)

Primary expression operators

For ANSI C, the primary expressions are *literal* (also sometimes referred to as *constant*), *identifier*, and **(** *expression* **)**. The C++ language extends this list of primary expressions to include the keyword **this**, scope resolution operator **::**, *name*, and the class destructor **~** (tilde).

The Borland C++ primary expressions are summarized in the following list. The complete list of expressions and operators is shown in Table 2.12 on page 78.

primary-expression:
literal
this (C++ specific)
:: *identifier* (C++ specific)
:: *operator-function-name* (C++ specific)
:: *qualified-name* (C++ specific)
(*expression*)
name

literal:
integer-constant
character-constant
floating-constant
string-literal

name:
identifier
operator-function-name (C++ specific)
conversion-function-name (C++ specific)
~ *class-name* (C++ specific)
qualified-name (C++ specific)

qualified-name: (C++ specific)
qualified-class-name **::** *name*

For a description of *literals*, see page 58. A complete list of formal definitions of literals is shown in Table 1.6 on page 58.

For a discussion of the primary expression **this**, see the section beginning on page 127. The keyword **this** cannot be used outside a class member function body.

The discussion of the scope resolution operator **::** begins on page 118. The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The discussion of **::** *identifier* and **::** *qualified-function-name* begins on page 133. You can find a summary on the use of operator **::** on page 159.

The parenthesis surrounding an *expression* do not change the unadorned *expression* itself.

The primary expression *name* is restricted to the category of primary expressions that sometimes appear after the member access operators **.** (dot) and **–>** . Therefore, *name* must be either an lvalue or a function (see page 28). See also the discussion of member access operators beginning on page 85.

An *identifier* is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown on page 10.

The discussion on how to use the destructor operator ~ (tilde), begins on page 138 and continues on page 146.

Postfix expression operators

The six postfix expression operators **[ ] ( ) . -> ++** and **- -** are used to build postfix expressions as shown in the expressions syntax table (Table 2.12 on page 78). Postfix expression operators group from left to right.

The following postfix expressions let you make safe, explicit typecasts in a C++ program.

See the "Typecasting" section beginning on page 109 for a description of these operators.

const_cast < *T* > (*expression* **)**
dynamic_cast < *T* > (*expression* **)**
reinterpret_cast < *T* > (*expression* **)**
static_cast < *T* > (*expression* **)**

To obtain run-time type information (RTTI), use the **typeid()** operator. The syntax is as follows:

typeid(*expression* **)**
typeid(*type-name* **)**

Array subscript operator [ ]

In the expression

postfix-expression [*expression*]

either *postfix-expression* or *expression* must be a pointer and the other an integral type.

In C, but not necessarily in C++, the expression *exp1[exp2]* is defined as

```
* ((exp1) + (exp2))
```

where either *exp1* is a pointer and *exp2* is an integer, or *exp1* is an integer and *exp2* is a pointer. The punctuators [ ], *, and + can be individually overloaded in C++.

Function call operators ( )

The expression

postfix-expression(<arg-expression-list>)

is a call to the function given by the postfix expression. The *arg-expression-list* is a comma-delimited list of expressions of any type representing the actual (or real) function arguments. The value of the function call expression, if any, is determined by the return statement in the function definition. See page 66 for more information on function calls.

Member access operators . (dot)

In the expression

postfix-expression **.** *name*

lvalues are defined on page 28.

the postfix expression must be of type structure or union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue. Detailed examples of the use of **.** (dot) and **->** for structures are given starting on page 68.

Member access operator ->

In the expression

postfix-expression **->** *name*

the postfix expression must be of type pointer to structure or pointer to union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if the selected member is an lvalue.

Increment operator ++

In the expression

postfix-expression **++**

the postfix expression is the operand; it must be of scalar type (arithmetic or pointer types) and must be a lvalue (see page 28 for more on modifiable lvalues). The postfix **++** is also known as the *postincrement* operator. The value of the whole expression is the value of the postfix expression *before* the increment is applied. After the postfix expression is evaluated, the operand is incremented by 1. The increment value is appropriate to the type of the operand. Pointer types are subject to the rules for pointer arithmetic.

Decrement operator --

The postfix decrement, also known as the *postdecrement,* operator follows the same rules as the postfix increment, except that 1 is subtracted from the operand *after* the evaluation.

Unary operators

The unary operators are described in the following table. Each operator is described in more detail in the sections following the table.

Table 2.13
Unary operators

Unary operator	Description
&	Address operator
*****	Indirection operator
+	Unary plus
–	Unary minus
~	Bitwise complement (one's complement)
!	Logical negation
++	Prefix: preincrement; Postfix: postincrement
– –	Prefix: predecrement; Postfix: postdecrement

The syntax is

unary-operator cast-expression

cast-expression:
  unary-expression
  (type-name) cast-expression

In C++, an explicit type cast can also be accomplished with cast operators. See page 109.

Address operator &

The symbol **&** is also used in C++ to specify reference types; see page 116.

The **&** operator and * operator (the * operator is described in the next section) work together as the *referencing* and *dereferencing* operators. In the expression

& *cast-expression*

the *cast-expression* operand must be either a function designator or an lvalue designating an object that is not a bit field and is not declared with the **register** storage class specifier. If the operand is of type ***T***, the result is of type pointer to ***T***.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into "pointer to *X*" types when appearing in certain contexts. The **&** operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following extract:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;       // initialized pointer
*ptr = t2;          // same effect as t1 = t2
```

`T *ptr = &t1` is treated as

```
T *ptr;
ptr = &ti;
```

so it is *ptr*, not **ptr*, that gets assigned. Once *ptr* has been initialized with the address &*t1*, it can be safely dereferenced to give the lvalue **ptr*.

Indirection operator *

In the expression

* *cast-expression*

the *cast-expression* operand must have type "pointer to ***T***," where ***T*** is any type. The result of the indirection is of type ***T***. If the operand is of type "pointer to function," the result is a function designator; if the operand is a pointer to an object, the result is an lvalue designating that object. In the following situations, the result of indirection is undefined:

- The *cast-expression* is a null pointer.
- The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Plus operator +

In the expression

\+ *cast-expression*

the *cast-expression* operand must be of arithmetic type. The result is the value of the operand after any required integral promotions.

Minus operator –

In the expression

– *cast-expression*

the *cast-expression* operand must be of arithmetic type. The result is the negative of the value of the operand after any required integral promotions.

Bitwise complement operator ~

In the expression

~ *cast-expression*

the *cast-expression* operand must be of integral type. The result is the bitwise complement of the operand after any required integral promotions. Each 0 bit in the operand is set to 1, and each 1 bit in the operand is set to 0.

Logical negation operator !

In the expression

! *cast-expression*

the *cast-expression* operand must be of scalar type. The result is of type **int** and is the logical negation of the operand: 0 if the operand is nonzero; 1 if the operand is zero. The expression *!E* is equivalent to `(0 == E)`.

Increment operator ++

In the expressions

++ *unary-expression*
unary-expression **++**

the unary expression is the operand; it must be of scalar type and must be a modifiable lvalue. The first expression shows the syntax for the prefix increment operator, also known as the *preincrement* operator. The operand is incremented by 1 *before* the expression is evaluated; the value of the whole expression is the incremented value of the operand. The 1 used to increment is the appropriate value for the type of the operand. Pointer types follow the rules of pointer arithmetic.

The second expression shows the syntax for the postfix increment operator (also known as the *postincrement* operator). The operand is incremented by 1 *after* the expression is evaluated.

Decrement operator --

The following expressions show the syntax for prefix and postfix decrementation. The prefix decrement is also known as the *predecrement*; the postfix decrement is also known as the *postdecrement*.

– – *unary-expression*
unary-expression **– –**

The operator follows the same rules as the increment operator, except that the operand is decremented by 1.

Binary operators

This section presents the binary operators, which are operators that require two operands.

Table 2.14
Binary operators

Type of operator	Binary operator	Description
Additive	+	Binary plus (addition)
	–	Binary minus (subtraction)
Multiplicative	*	Multiply
	/	Divide
	%	Remainder
Shift	<<	Shift left
	>>	Shift right
Bitwise	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	|	Bitwise inclusive OR

Table 2.14: Binary operators (continued)

Logical	&&	Logical AND
	||	Logical OR
Assignment	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	–=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
Relational	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
Equality	==	Equal to
	!=	Not equal to
Component selection	.	Direct component selector
	–>	Indirect component selector
C++ operators	::	Scope access/resolution
	.*	Dereference pointer to class member
	–>*	Dereference pointer to class member
	:	Class initializer
Conditional	*a* **?** *x* **:** *y*	"if *a* then *x*; else *y*"
Comma	,	Evaluate; for example, `a, b, c;` from left to right

The operator functions, as well as their syntax, precedences, and associativities, are covered starting on page 77.

Additive operators

There are two additive operators: **+** and **–**. The syntax is

additive-expression:
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **–** *multiplicative-expression*

Addition +

The legal operand types for *op1* **+** *op2* are

- Both *op1* and *op2* are of arithmetic type.
- *op1* is of integral type, and *op2* is of pointer to object type.
- *op2* is of integral type, and *op1* is of pointer to object type.

In the first case, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In the second and third cases, the rules of pointer arithmetic apply. (Pointer arithmetic is covered on page 59.)

Subtraction –

The legal operand types for *op1* – *op2* are

- Both *op1* and *op2* are of arithmetic type.
- Both *op1* and *op2* are pointers to compatible object types. The unqualified type ***type*** is considered to be compatible with the qualified types **const *type***, **volatile *type***, and **const volatile *type***.
- *op1* is of pointer to object type, and *op2* is integral type.

In the first case, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In the second and third cases, the rules of pointer arithmetic apply.

Multiplicative operators

There are three multiplicative operators: * **/** and **%**. The syntax is

multiplicative-expression:
    cast-expression
    multiplicative-expression * *cast-expression*
    multiplicative-expression **/** *cast-expression*
    multiplicative-expression **%** *cast-expression*

The operands for * (multiplication) and **/** (division) must be of arithmetical type. The operands for **%** (modulus, or remainder) must be of integral type. The usual arithmetic conversions are made on the operands (see page 42).

The result of (*op1* * *op2*) is the product of the two operands. The results of (*op1* **/** *op2*) and (*op1* **%** *op2*) are the quotient and remainder, respectively, when *op1* is divided by *op2*, provided that *op2* is nonzero. Use of **/** or **%** with a zero second operand results in an error.

When *op1* and *op2* are integers and the quotient is not an integer, the results are as follows:

Rounding is always toward zero.

- If *op1* and *op2* have the same sign, *op1* **/** *op2* is the largest integer less than the true quotient, and *op1* **%** *op2* has the sign of *op1*.
- If *op1* and *op2* have opposite signs, *op1* **/** *op2* is the smallest integer greater than the true quotient, and *op1* **%** *op2* has the sign of *op1*.

Bitwise logic operators

There are three bitwise logical operators: **&**, **^** and **|**.

AND &

The syntax is

AND-expression:
  equality-expression
  AND-expression **&** *equality-expression*

In the expression *E1* **&** *E2*, both operands must be of integral type. The usual arithmetical conversions are performed on *E1* and *E2*, and the result is the bitwise AND of *E1* and *E2*. Each bit in the result is determined as shown in Table 2.15.

Table 2.15 Bitwise operators truth table

Bit value in *E1*	Bit value in *E2*	*E1* & *E2*	*E1* ^ *E2*	*E1* | *E2*
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Exclusive OR ^

The syntax is

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression **^** *AND-expression*

In the expression *E1* **^** *E2*, both operands must be of integral type. The usual arithmetic conversions are performed on *E1* and *E2*, and the result is the bitwise exclusive OR of *E1* and *E2*. Each bit in the result is determined as shown in Table 2.15.

Inclusive OR |

The syntax is

inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression | *exclusive-OR-expression*

In the expression *E1* | *E2*, both operands must be of integral type. The usual arithmetic conversions are performed on *E1* and *E2*, and the result is the bitwise inclusive OR of *E1* and *E2*. Each bit in the result is determined as shown in Table 2.15.

Bitwise shift operators

There are two bitwise shift operators: **<<** and **>>**. The syntax is

shift-expression:
  additive-expression
  shift-expression **<<** *additive-expression*
  shift-expression **>>** *additive-expression*

Shift (<< and >>)

In the expressions *E1* **<<** *E2* and *E1* **>>** *E2*, the operands *E1* and *E2* must be of integral type. The normal integral promotions are performed on *E1* and *E2*, and the type of the result is the type of the promoted *E1*. If *E2* is negative or is greater than or equal to the width in bits of *E1*, the operation is undefined.

The constants ULONG_MAX and UINT_MAX are defined in limits.h.

The result of *E1* **<<** *E2* is the value of *E1* left-shifted by *E2* bit positions, zero-filled from the right if necessary. Left shifts of an **unsigned long** *E1* are equivalent to multiplying *E1* by 2^{E2}, reduced modulo ULONG_MAX + 1; left shifts of **unsigned int**s are equivalent to multiplying by 2^{E2} reduced modulo UINT_MAX + 1. If *E1* is a signed integer, the result must be interpreted with care, because the sign bit might change.

The result of *E1* **>>** *E2* is the value of *E1* right-shifted by *E2* bit positions. If *E1* is of **unsigned** type, zero-fill occurs from the left if necessary. If *E1* is of **signed** type, the fill from the left uses the sign bit (0 for positive, 1 for negative *E1*). This sign-bit extension ensures that the sign of *E1* **>>** *E2* is the same as the sign of *E1*. Except for signed types, the value of *E1* **>>** *E2* is the integral part of the quotient $E1/2^{E2}$.

Relational operators

There are four relational operators: **<** **>** **<=** and **>=**. The syntax for these operators is

relational-expression:
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

Less-than <

In the expression *E1* **<** *E2*, the operands must conform to one of the following sets of conditions:

Qualified names are defined on page 132.

- Both *E1* and *E2* are of arithmetic type.
- Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible object types.
- Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible incomplete types.

In the first case, the usual arithmetic conversions are performed. The result of *E1* **<** *E2* is of type **int**. If the value of *E1* is less than the value of *E2*, the result is 1 (true); otherwise, the result is zero (false).

In the second and third cases, in which *E1* and *E2* are pointers to compatible types, the result of *E1* **<** *E2* depends on the relative locations (addresses) of the two objects being pointed at. When comparing structure members within the same structure, the "higher" pointer indicates a later declaration. Within arrays, the "higher" pointer indicates a larger subscript value. All pointers to members of the same union object compare as equal.

Normally, the comparison of pointers to different structure, array, or union objects, or the comparison of pointers outside the range of an array object give undefined results; however, an exception is made for the "pointer beyond the last element" situation as discussed in the "Pointer arithmetic" section on page 59. If P points to an element of an array object, and Q points to the last element, the expression $P < Q + 1$ is allowed, evaluating to 1 (true), even though $Q + 1$ does not point to an element of the array object.

Greater-than >

The expression *E1* **>** *E2* gives 1 (true) if the value of *E1* is greater than the value of *E2*; otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons as are defined for the less-than operator. The same operand rules and restrictions also apply.

Less-than or equal-to <=

Similarly, the expression *E1* **<=** *E2* gives 1 (true) if the value of *E1* is less than or equal to the value of *E2*. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons as are defined for the less-than operator. The same operand rules and restrictions also apply.

Greater-than or equal-to >=

Finally, the expression *E1* **>=** *E2* gives 1 (true) if the value of *E1* is greater than or equal to the value of *E2*. Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons as are defined for the less-than operator. The same operand rules and restrictions also apply.

Equality operators

There are two equality operators: **==** and **!=**. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

Notice that **==** and **!=** have a lower precedence than the relational operators **<** and **>**, **<=**, and **>=**. Also, **==** and **!=** can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

equality-expression:
  relational-expression
  equality-expression **==** *relational-expression*
  equality-expression **!=** *relational-expression*

Equal-to ==

In the expression *E1* == *E2*, the operands must conform to one of the following sets of conditions:

- Both *E1* and *E2* are of arithmetic type.
- Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible types.
- One of *E1* and *E2* is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**.
- One of *E1* or *E2* is a pointer and the other is a null pointer constant.

If *E1* and *E2* have types that are valid operand types for a relational operator, the same comparison rules just detailed for *E1* **<** *E2*, *E1* **<=** *E2*, and so on, apply.

In the first case, for example, the usual arithmetic conversions are performed, and the result of *E1* **==** *E2* is of type **int**. If the value of *E1* is equal to the value of *E2*, the result is 1 (true); otherwise, the result is zero (false).

In the second case, *E1* **==** *E2* gives 1 (true) if *E1* and *E2* point to the same object, or both point "one past the last element" of the same array object, or both are null pointers.

If *E1* and *E2* are pointers to function types, *E1* **==** *E2* gives 1 (true) if they are both null or if they both point to the same function. Conversely, if *E1* **==** *E2* gives 1 (true), then either *E1* and *E2* point to the same function, or they are both null.

In the fourth case, the pointer to an object or incomplete type is converted to the type of the other operand (pointer to a qualified or unqualified version of **void**).

Inequality !=

The expression *E1* **!=** *E2* follows the same rules as those for *E1* **==** *E2*, except that the result is 1 (true) if the operands are unequal, and 0 (false) if the operands are equal.

Logical operators

There are two logical operators: **&&** and **||**.

AND &&

The syntax is

> *logical-AND-expression:*
> *inclusive-OR-expression*
> *logical-AND-expression* **&&** *inclusive-OR-expression*

In the expression *E1* **&&** *E2*, both operands must be of scalar type. The result is of type **int**, and the result is 1 (true) if the values of *E1* and *E2* are both nonzero; otherwise, the result is 0 (false).

Unlike the bitwise **&** operator, **&&** guarantees left-to-right evaluation. *E1* is evaluated first; if *E1* is zero, *E1* **&&** *E2* gives 0 (false), and *E2* is not evaluated.

OR ||

The syntax is

logical-OR-expression:
  logical-AND-expression
  logical-OR-expression **||** *logical-AND-expression*

In the expression *E1* **||** *E2*, both operands must be of scalar type. The result is of type **int**, and the result is 1 (true) if either of the values of *E1* and *E2* are nonzero. Otherwise, the result is 0 (false).

Unlike the bitwise **|** operator, **||** guarantees left-to-right evaluation. *E1* is evaluated first; if *E1* is nonzero, *E1* **||** *E2* gives 1 (true), and *E2* is not evaluated.

Conditional ? :

The syntax is

conditional-expression
  logical-OR-expression
  logical-OR-expression **?** *expression* **:** *conditional-expression*

In C++, the result is an lvalue.

In the expression *E1* **?** *E2* **:** *E3*, the operand *E1* must be of scalar type. The operands *E2* and *E3* must obey one of the following rules:

- Rule 1: Both are of arithmetic type.
- Rule 2: Both are of compatible structure or union types.
- Rule 3: Both are of **void** type.
- Rule 4: Both are of type pointer to qualified or unqualified versions of compatible types.
- Rule 5: One operand is of pointer type, the other is a null pointer constant.
- Rule 6: One operand is of type pointer to an object or incomplete type, the other is of type pointer to a qualified or unqualified version of **void**.

First, *E1* is evaluated; if its value is nonzero (true), then *E2* is evaluated and *E3* is ignored. If *E1* evaluates to zero (false), then *E3* is evaluated and *E2* is ignored. The result of *E1* **?** *E2* **:** *E3* will be the value of whichever of *E2* and *E3* is evaluated.

In rule 1, both *E2* and *E3* are subject to the usual arithmetic conversions, and the type of the result is the common type resulting from these conversions. In rule 2, the type of the result is the structure or union type of *E2* and *E3*. In rule 3, the result is of type **void**.

In rules 4 and 5, the type of the result is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands. In rule 6, the type of the result is that of the nonpointer-to-void operand.

Assignment operators

There are 11 assignment operators. The **=** operator is the simple assignment operator; the other 10 are known as compound assignment operators.

The syntax is

assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator: one of

= *= /= %= += –=
<<= >>= &= ^= |=

Simple assignment =

In the expression *E1* = *E2*, *E1* must be a modifiable lvalue. The value of *E2*, after conversion to the type of *E1*, is stored in the object designated by *E1* (replacing *E1*'s previous value). The value of the assignment expression is the value of *E1* after the assignment. The assignment expression is not itself an lvalue.

In C++, the result is an lvalue.

The operands *E1* and *E2* must obey one of the following rules:

- Rule 1: *E1* is of qualified or unqualified arithmetic type and *E2* is of arithmetic type.
- Rule 2: *E1* has a qualified or unqualified version of a structure or union type compatible with the type of *E2*.
- Rule 3: *E1* and *E2* are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- Rule 4: One of *E1* or *E2* is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
- Rule 5: *E1* is a pointer and *E2* is a null pointer constant.

Compound assignment

The compound assignments *op*=, where *op* can be any one of the 10 operator symbols *** / % + – << >> & ^ |**, are interpreted as follows:

E1 op= *E2*

has the same effect as

E1 = *E1 op E2*

except that the lvalue *E1* is evaluated only once. (For example, *E1* **+=** *E2* is the same as *E1* = *E1* + *E2*.)

The rules for compound assignment are therefore covered in the previous section (on the simple assignment operator **=**).

Comma operator

The syntax is

expression:
  assignment-expression
  expression **,** *assignment-expression*

In C++, the result is an lvalue.

In the comma expression

E1, *E2*

the left operand *E1* is evaluated as a **void** expression, then *E2* is evaluated to give the result and type of the comma expression. By recursion, the expression

E1, *E2*, ..., *En*

results in the left-to-right evaluation of each *Ei*, with the value and type of *En* giving the result of the whole expression. To avoid potential ambiguity (which might arise from the commas being used in both function arguments and in initializer lists), parentheses must be used. For example,

```
func(i, (j = 1, j + 4), k);
```

calls *func* with three arguments, not four. The arguments are *i*, 5, and *k*.

C++ operators

The operators specific to C++ are as follows:

- **::** (scope resolution)
- **.*** (dereference pointer)
- **–>*** (dereference pointer)
- **:** (class initializer)

See page 118 for information on the scope access operator ::. See also page 144 for a discussion of : class initializer.

The syntax for the **.*** and **–>*** operators is as follows:

pm-expression
  cast-expression
  pm expression **.*** *cast-expression*
  pm expression **–>*** *cast-expression*

The **.*** operator dereferences pointers to class members. It binds the *cast-expression*, which must be of type "pointer to member of class *type*", to the *pm-expression*, which must be of class *type* or of a class publicly derived from class *type*. The result is an object or function of the type specified by the *cast-expression*.

The **–>*** operator dereferences pointers to pointers to class members (this isn't a typographical error; it does indeed dereference pointers to pointers). It binds the *cast-expression*, which must be of type "pointer to member of *type*," to the *pm-expression*, which must be of type pointer to *type* or of type "pointer to class publicly derived from *type*." The result is an object or function of the type specified by the *cast-expression*.

If the result of either of these operators is a function, you can only use that result as the operand for the function call operator **( )**. For example,

```
#include <iostream.h>

class B {
public:
   void g(int i = 0) { cout << "\nInput = " << i; };
};

int main(void) {
   B Binst;              // Instantiate class B

   // pf is a pointer to a B member function that takes an integer and returns void
   void (B::*pf) (int);
   pf = B::g;            // Initialize pf to the B::g() member function.
   (Binst.*pf) (21);     // Call g() and give it the argument 21.
   return 0;
   }
```

The sizeof operator

The **sizeof** operator has two distinct uses:

sizeof *unary-expression*
sizeof (*type-name*)

The amount of space that is reserved for each type depends on the machine.

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is *not* converted to a pointer type). The number of elements in an array equals **sizeof** *array* / **sizeof** *array*[0].

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

sizeof cannot be used with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is *size_t*, defined as **unsigned int** in stddef.h.

You can use **sizeof** in preprocessor directives; this is specific to Borland C++.

In C++, **sizeof**(*classtype*), where *classtype* is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Source

```
/*  USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>

struct st {
     char *name;     /* 2 BYTES IN SMALL-DATA MODELS; 4 BYTES IN LARGE-DATA MODEL
                        */
     int age;        /* 2 BYTES IN SMALL-DATA MODELS; 4 BYTES IN LARGE-DATA MODEL
                        */
     double height;  /* EIGHT BYTES */
     };
struct st St_Array[]= {  /* AN ARRAY OF structs */
     { "Jr.",     4,  34.20 },  /* ST_Array[0] */
     { "Suzie",  23,  69.75 },  /* ST_Array[1] */
     };

int main() {
     long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };

     printf("\nNumber of elements in LD_Array = %d",
               sizeof(LD_Array) / sizeof(LD_Array[0]));

     /****  THE NUMBER OF ELEMENTS IN THE ST_Array. ****/
     printf("\nSt_Array has %d elements",
                sizeof(St_Array)/sizeof(St_Array[0]));

     /****  THE NUMBER OF BYTES IN EACH ST_Array ELEMENT.  ****/
     printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));

     /****  THE TOTAL NUMBER OF BYTES IN ST_Array.  ****/
     printf("\nSt_Array= %d", sizeof(St_Array));
   return 0;
   }
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 12
St_Array= 24
```

Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. The following table shows the syntax for statements.

Table 2.16: Borland C++ statements

statement:
  labeled-statement
  compound-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  asm-statement
  declaration (C++ specific)

labeled-statement:
  identifier : statement
  case *constant-expression : statement*
  default : statement

compound-statement:
  { <declaration-list> <statement-list> }

declaration-list:
  declaration
  declaration-list declaration

statement-list:
  statement
  statement-list statement

expression-statement:
  <expression> ;

asm-statement:
  asm *tokens newline*
  asm *tokens;*
  asm { *tokens; <tokens;>=*
    <tokens;>
    }

selection-statement:
  if *(expression) statement*
  if *(expression) statement* **else** *statement*
  switch *(expression) statement*

iteration-statement:
  while *(expression) statement*
  do *statement* **while** *(expression) ;*
  for *(for-init-statement <expression> ; <expression>) statement*

for-init-statement
  expression-statement
  declaration (C++ specific)

jump-statement:
  goto *identifier ;*
  continue ;
  break ;
  return *<expression> ;*

Blocks

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

Labeled statements

A statement can be labeled in two ways:

- *label-identifier* **:** *statement*

  The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.
- **case** *constant-expression* **:** *statement*
  default : *statement*

  Case and default labeled statements are used only in conjunction with switch statements.

Expression statements

Any expression followed by a semicolon forms an *expression statement*:

<expression>;

Borland C++ executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls.

The *null statement* is a special case, consisting of a single semicolon (**;**). The null statement does nothing, and is therefore useful in situations where the Borland C++ syntax expects a statement but your program does not need one.

Selection statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch.**

if statements

The basic **if** statement has the following pattern:

if (*cond-expression*) *t-st* <**else** *f-st*>

The parentheses around *cond-expression* are essential.

The *cond-expression* must be of scalar type. The expression is evaluated. If the value is zero (or null for pointer types), *cond-expression* is false; otherwise, it is true.

If there is no **else** clause and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored.

If the optional **else** *f-st* is present and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored and *f-st* is executed.

➡ Unlike Pascal, for example, Borland C++ does not have a specific Boolean data type. Any expression of integer or pointer type can serve a Boolean role in conditional tests. The relational expression (*a* > *b*) (if legal) evaluates to **int** 1 (true) if (*a* > *b*), and to **int** 0 (false) if (*a* <= *b*). Pointer conversions are such that a pointer can always be correctly compared to a constant expression evaluating to 0. That is, the test for null pointers can be written `if (!ptr)...` or `if (ptr == 0)....`

The *f-st* and *t-st* statements can themselves be **if** statements, allowing for a series of conditional tests nested to any depth. Care is needed with nested **if...else** constructs to ensure that the correct statements are selected. There is no **endif** statement: any "else" ambiguity is resolved by matching an **else** with the last encountered **if**-without-an-**else** at the same block level. For example,

```
if (x == 1)
   if (y == 1) puts("x=1 and y=1");
else puts("x != 1");
```

draws the wrong conclusion. The **else** matches with the second **if**, despite the indentation. The correct conclusion is that $x = 1$ and y != 1. Note the effect of braces:

```
if (x == 1) {
   if (y == 1) puts("x = 1 and y = 1");
}
else puts("x != 1"); // correct conclusion
```

switch statements

The **switch** statement uses the following basic format:

switch (*sw-expression*) *case-st*

A **switch** statement lets you transfer control to one of several case-labeled statements, depending on the value of *sw-expression*. The latter must be of integral type (in C++, it can be of class type, provided that there is an unambiguous conversion to integral type available). Any statement in *case-st* (including empty statements) can be labeled with one or more case labels:

It is illegal to have duplicate case constants in the same **switch** statement.

case *const-exp-i* **:** *case-st-i*

where each case constant, *const-exp-i*, is a constant expression with a unique integer value (converted to the type of the controlling expression) within its enclosing **switch** statement.

There can also be at most one **default** label:

default : *default-st*

After evaluating *sw-expression*, a match is sought with one of the *const-exp-i*. If a match is found, control passes to the statement *case-st-i* with the matching case label.

If no match is found and there is a **default** label, control passes to *default-st*. If no match is found and there is no **default** label, none of the statements in *case-st* is executed. Program execution is not affected when **case** and **default** labels are encountered. Control simply passes through the labels to the following statement or switch. To stop execution at the end of a group of statements for a particular case, use **break**.

```
/* THIS ILLUSTRATES THE USE OF KEYWORDS switch, case, AND default. */
#include <stdio.h>

int main(void) {
   int ch;

   printf("\tPRESS a, b, OR c. ANY OTHER CHOICE WILL "
         "TERMINATE THIS PROGRAM.");
   for ( /* FOREVER */; ((ch = getch(stdin)) != EOF); )
      switch (ch) {
         case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
            printf("\nOption a was selected.\n");
            break;
         case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
         case 'c' :
            printf("\nOption b or c was selected.\n");
            break;
         default :
            printf("\nNOT A VALID CHOICE!  Bye ...");
            return(-1);
         }
   return(0);
   }
```

Iteration statements

Iteration statements let you loop a set of statements. There are three forms of iteration in Borland C++: **while**, **do while**, and **for** loops.

while statements

The general format for this statement is

while (*cond-exp*) *t-st*

The parentheses are essential.

The loop statement, *t-st*, is executed repeatedly until the conditional expression, *cond-exp*, compares equal to zero (false).

The *cond-exp* is evaluated and tested first (as described on page 102). If this value is nonzero (true), *t-st* is executed; if no jump statements that exit from the loop are encountered, *cond-exp* is evaluated again. This cycle repeats until *cond-exp* is zero.

As with **if** statements, pointer type expressions can be compared with the null pointer, so that `while (ptr)...` is equivalent to `while (ptr != NULL)....`

The **while** loop offers a concise method for scanning strings and other null-terminated data structures:

```
char str[10]="Borland";
char *ptr=&str[0];
int count=0;
   ⋮
while (*ptr++)  // loop until end of string
   count++;
```

In the absence of jump statements, *t-st* must affect the value of *cond-exp* in some way, or *cond-exp* itself must change during evaluation in order to prevent unwanted endless loops.

do while statements

The general format is

do *do-st* **while** (*cond-exp*);

The *do-st* statement is executed repeatedly until *cond-exp* compares equal to zero (false). The key difference from the **while** statement is that *cond-exp* is tested *after*, rather than before, each execution of the loop statement. At least one execution of *do-st* is assured. The same restrictions apply to the type of *cond-exp* (scalar).

for statement

For C++, <*init-exp*> can be an expression or a declaration.

The **for** statement format in C is

for (<*init-exp*>; <*test-exp*>; <*increment-exp*>) *statement*

The sequence of events is as follows:

1. The initializing expression *init-exp*, if any, is executed. As the name implies, this usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity (including declarations in C++)—hence the claim that any C program can be written as a single **for** loop.
2. The expression *test-exp* is evaluated following the rules of the **while** loop. If *test-exp* is nonzero (true), the loop statement is executed. An

empty expression here is taken as `while (1)`; that is, always true. If the value of *test-exp* is zero (false), the **for** loop terminates.

3. *increment-exp* advances one or more counters.
4. The expression *statement* (possibly empty) is evaluated and control returns to step 2.

If any of the optional elements are empty, appropriate semicolons are required:

```
for (;;) {      // same as for (; 1;)
   // loop forever
}
```

The C rules for **for** statements apply in C++. However, the *init-exp* in C++ can also be a declaration. The scope of a declared identifier extends through the enclosing loop. For example,

```
for (int i = 1; i < 3; ++i) {
   if (i ...)                 // ok to refer to i here
      ⋮
   for (int x = 0;;;) ;       // do nothing
}
if (i...)                     // legal
if (x...)                     // illegal; x is now out of scope
```

Jump statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**.

break statements

The syntax is

break;

A **break** statement can be used only inside an iteration (**while**, **do**, and **for** loops) or a **switch** statement. It terminates the iteration or **switch** statement. Because iteration and **switch** statements can be intermixed and nested to any depth, you must ensure that your **break** exits from the correct loop or switch. The rule is that a **break** terminates the *nearest* enclosing iteration or **switch** statement.

continue statements

The syntax is

continue;

A **continue** statement can be used only inside an iteration statement; it transfers control to the test condition for **while** and **do** loops, and to the increment expression in a **for** loop.

With nested iteration loops, a **continue** statement is taken as belonging to the *nearest* enclosing iteration.

goto statements

The syntax is

goto *label;*

The **goto** statement transfers control to the statement labeled *label* (see page 102), which must be in the same function.

In C++, it is illegal to bypass a declaration having an explicit or implicit initializer unless that declaration is within an inner block that is also bypassed.

return statements

Unless the function return type is **void**, a function body must contain at least one **return** statement with the following format:

return *return-expression;*

where *return-expression* must be of type ***type*** or of a type that is convertible to ***type*** by assignment. The value of the *return-expression* is the value returned by the function. An expression that calls the function, such as `func(actual-arg-list)`, is an rvalue of type ***type***, not an lvalue:

```
t = func(arg);       // OK
func(arg) = t;       /* illegal in C; legal in C++ if return type of func is a
                        reference */
(func(arg))++;       /* illegal in C; legal in C++ if return type of func is a
                        reference */
```

The execution of a function call terminates if a **return** statement is encountered; if no **return** is met, execution continues, ending at the final closing brace of the function body.

If the return type is **void**, the **return** statement can be written as

```
{  :
   return;
}
```

with no return expression; alternatively, the **return** statement can be omitted.

CHAPTER 3

C++ specifics

See Chapter 4 for details on compiling C and C++ programs with exception handling.

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function *func* declared twice in C with different argument types invokes a duplicated name error. Under C++, however, *func* will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This chapter reviews the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

New-style typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions available in the C language.

Types cannot be defined in a cast.

const_cast typecast operator

Use the **const_cast** operator to add or remove the **const** or **volatile** modifier from a type.

In the statement, `const_cast< T > (arg)`, *T* and *arg* must be of the same type except for **const** and **volatile** modifiers. The cast is resolved at compile time. The result is of type *T*. Any number of **const** or **volatile** modifiers can be added or removed with a single **const_cast** expression.

A pointer to **const** can be converted to a pointer to non-**const** that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A **const** object or a reference to **const** cast results in a non-**const** object or reference that is otherwise an identical type.

The **const_cast** operator performs similar typecasts on the **volatile** modifier. A pointer to **volatile** object can be cast to a pointer to non-**volatile** object without otherwise changing the object's type. The result is a pointer to the original object. A **volatile**-type object or a reference to **volatile**-type can be converted into an identical non-**volatile** type.

dynamic_cast typecast operator

Run-time type identification (RTTI) is required for **dynamic_cast**. See the description of class *Type_info* in the *Library Reference*, Chapter 10. See also the discussion of RTTI on page 113.

In the expression `dynamic_cast< T > (ptr)`, *T* must be a pointer or a reference to a defined class type or **void***. The argument *ptr* must be an expression that resolves to a pointer or reference.

If *T* is **void*** then *ptr* must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if *T* is a pointer and *ptr* is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If *T* is a reference and *ptr* is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type. See page 155 for a discussion of polymorphic types.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at run time.

If successful, `dynamic_cast< T > (ptr)` converts *ptr* to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the *Bad_cast* exception is thrown.

This program must be compiled with the **–RT** (Generate RTTI) option.

```
// HOW TO MAKE DYNAMIC CASTS
#include <iostream.h>
#include <typeinfo.h>

class Base1
{
   // For the RTTI mechanism to function correctly,
   // a base class must be polymorphic.
   virtual void f(void) { /* A virtual function makes the class polymorphic */ }
};

class Base2 { };
class Derived : public Base1, public Base2 { };
```

```
int main(void) {
   try {
      Derived d, *pd;
      Base1 *b1 = &d;

      // Perform a downcast from a Base1 to a Derived.
      if ((pd = dynamic_cast<Derived *>(b1)) != 0) {
           cout << "The resulting pointer is of type "
                << typeid(pd).name() << endl;
         }
      else throw Bad_cast();

      // Attempt cast across the hierarchy.  That is, cast from
      // the first base to the most derived class and then back
      // to another accessible base.
      Base2 *b2;
      if ((b2 = dynamic_cast<Base2 *>(b1)) != 0) {
          cout << "The resulting pointer is of type "
               << typeid(b2).name() << endl;
         }
      else throw Bad_cast();
      }
   catch (Bad_cast) {
      cout << "dynamic_cast failed" << endl;
      return 1;
      }
   catch (...) {
      cout << "Exception handling error." << endl;
      return 1;
      }

   return 0;
}
```

reinterpret_cast typecast operator

In the statement `reinterpret_cast< T > (arg)`, *T* must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral *arg* can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

```
// Use reinterpret_cast<Type>(expr) to replace (Type)expr casts
// for conversions that are unsafe or implementation dependent.

void func(void *v) {
        // Cast from pointer type to integral type.
        int i = reinterpret_cast<int>(v);
   :
}

void main() {
   // Cast from an integral type to pointer type.
   func(reinterpret_cast<void *>(5));

   // Cast from a pointer to function of one type to
   // pointer to function of another type.
   typedef void (* PFV)();

   PFV pfunc = reinterpret_cast<PFV>(func);

   pfunc();
   }
```

static_cast typecast operator

In the statement `static_cast< T > (arg)`, *T* must be a pointer, reference, arithmetic type, or **enum** type. The *arg*-type must match the *T*-type. Both *T* and *arg* must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using **static_cast** achieves exactly the same thing.

Integral types can be converted to **enum** types. A request to convert *arg* to a value that is not an element of **enum** is undefined.

The null pointer is converted to itself.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class *X* to a pointer to some class *Y* if *X* is a base class for *Y*. A static conversion can be made only under the following conditions:

- If an unambiguous conversion exists from *Y* to *X*
- If *X* is not a virtual base class

See page 137 for a discussion of virtual base classes.

An object can be explicitly converted to a reference type *X&* if a pointer to that object can be explicitly converted to an *X**. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When *T* is a reference the result of `static_cast< T > (arg)` is an lvalue. The result of a pointer or reference cast refers to the original expression.

Run-time type identification

The recent addition of run-time type identification (RTTI) into the ANSI/ISO C++ working paper makes it possible to write portable code that can determine the actual type of a data object at run time even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. See page 110 for a description of the **dynamic_cast** operator, which uses run-time type information.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with **typeid** operator, which determines the actual type of its argument and returns a reference to an object of type **const** *Type_info*, which describes that type. You can also use a type name as the argument to **typeid**, and **typeid** will return a reference to a **const** *Type_info* object for that type. The class *Type_info* provides an **operator==** and an **operator!=** that you can use to determine whether two objects are of the same type. Class *Type_info* also provides a member function *name* that returns a pointer to a **char** array that holds the name of the type. See the *Library Reference*, Chapter 10, for a description of class *Type_info*.

The typeid operator

You can use **typeid** to get run-time information about types or expressions. A call to **typeid** returns a reference to an object of type **const Type_info**. The returned object represents the type of the **typeid** operand.

To use the **typeid** operator you must include the typeinfo.h header file.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

Example

```
// HOW TO USE typeid, Type_info::before(), and Type_info::name().
#include <iostream.h>
#include <string.h>
#include <typeinfo.h>

class A { };
class B : A { };
char *true  = "true";
char *false = "false";

void main() {
   char C;
   float X;

   if (typeid( C ) == typeid( X ))
      cout << "C and X are the same type." << endl;
   else cout << "C and X are NOT the same type." << endl;

    cout << typeid(int).name();
    cout << " before " << typeid(double).name() << ": " <<
         (typeid(int).before(typeid(double)) ? true : false) << endl;

    cout << typeid(double).name();
    cout << " before " << typeid(int).name() << ": " <<
         (typeid(double).before(typeid(int)) ? true : false) << endl;

    cout << typeid(A).name();
    cout << " before " << typeid(B).name() << ": " <<
         (typeid(A).before(typeid(B)) ? true : false) << endl;
   }
```

Program output

```
C and X are NOT the same type.
int before double: false
double before int: true
A before B: true
```

If the **typeid** operand is a dereferenced NULL pointer, the *Bad_typeid* exception is thrown. See the *Library Reference*, Chapter 10, for a description of *Bad_typeid*.

The _ _ rtti keyword and the –RT option

RTTI is enabled by default in Borland C++. You can use the **–RT** command-line option to disable it (**–RT-**) or to enable it (**–RT**). If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class (see page 155 for a discussion of polymorphic classes), **typeid** returns a reference to a **const** *Type_info* object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic run-time type identification (where appropriate) by using the Borland C++ keyword **_ _rtti** in the class definition.

When you use the **–RT-** compiler option, if any base class is declared **_ _rtti**, then all polymorphic base classes must also be declared **_ _rtti**.

```
struct __rtti S1 { virtual s1func(); };  // Polymorphic

struct __rtti S2 { virtual s2func(); };  // Polymorphic

struct X : S1, S2 { };
```

If you turn off the RTTI mechanism (by using the **–RT-** compiler option), RTTI might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations with **–RT-**, you should declare *X* with the **_ _rtti** modifier. Otherwise, switching the order of the base classes for the class *X* results in the compile-time error `Can't inherit non-RTTI class from RTTI base 'S1'.`

Note that the class *X* is explicitly declared with **_ _rtti**. This makes it safe to mix the order and type of classes.

```
struct _ _rtti S1 { virtual func(); };    // Polymorphic class

struct S2 { };                            // Non-polymorphic class

struct _ _rtti X : S1, S2 { };
```

In this example, class *X* inherits only non-polymorphic classes. Class *X* does not need to be declared **_ _rtti**.

```
struct _ _rtti S1 {  };    // Non-polymorphic class

struct S2 { };

struct X : S2, S1 { };    // The order is not essential
```

Applying either **_ _rtti** or using the **–RT** compiler option will *not* make a static class into a polymorphic class. See page 155 for a discussion of polymorphic classes.

Example

```
// HOW TO GET RUN-TIME TYPE INFORMATION FOR POLYMORPHIC CLASSES.
#include <iostream.h>
#include <typeinfo.h>
```

```
class _ _rtti Alpha {        //   Provide RTTI for this class and
                            //   all classes derived from it
   virtual void func() {};  // A virtual function makes Alpha a polymorphic class.
};

class B : public Alpha {};

int main(void) {
   B Binst;           // Instantiate class B
   B *Bptr;           // Declare a B-type pointer
   Bptr = &Binst;     // Initialize the pointer

   // THESE TESTS ARE DONE AT RUN TIME
   try {
      if (typeid( *Bptr ) == typeid( B ) )
          // Ask "WHAT IS THE TYPE FOR *Bptr?"
          cout << "Name is " << typeid( *Bptr).name();
      if (typeid( *Bptr ) != typeid( Alpha ) )
          cout << "\nPointer is not an Alpha-type.";
      return 0;
      }
   catch (Bad_typeid) {
      cout << "typeid() has failed.";
      return 1;
      }
   }
```

Program output

```
Name is B
Pointer is not an Alpha-type.
```

Referencing

C++ specific pointer referencing and dereferencing is discussed on page 98.

While in C, you pass arguments only by value; in C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects and let you pass arguments to functions by reference.

Simple references

The reference declarator can be used to declare references outside functions:

Note that type& var, type &var, and type & var are all equivalent.

```
int  i  = 0;
int &ir = i;    // ir is an alias for i
ir = 2;         // same effect as i = 2
```

This creates the lvalue *ir* as an alias for *i*, provided the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, `ir = 2` assigns 2 to *i*, and `&ir` returns the address of *i*.

Reference arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);    // ir is type "reference to int"
      ⋮
int sum=3;
func1(sum);              // sum passed by value
func2(&sum);             // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by *func2*. On the other hand, *func1* gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by *func1*.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can return a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The C method for changing *x* uses the actual argument `&x`, the address of *x*, rather than *x* itself. Although `&x` is passed by value, the function can access *x* through the copy of `&x` it receives. Even if the function does not need to change *x*, it is still useful (though subject to potentially dangerous side effects) to pass `&x`, especially if *x* is a large data structure. Passing *x* directly by value involves wasteful copying of the data structure.

Compare the three implementations of the function *treble*:

Implementation 1

```
int treble_1(int n)
{
   return 3 * n;
}
   ⋮
int x, i = 4;
x = treble_1(i);         // x now = 12, i = 4
   ⋮
```

Implementation 2

```
void treble_2(int* np)
{
   *np = (*np) * 3;
}
   ⋮
treble_2(int& i);        // i now = 12
```

Implementation 3

```
void treble_3(int& n)      // n is a reference type
{
   n = 3 * n;
}
   ⋮
treble_3(i);               // i now = 36
```

The formal argument declaration ***type*** `& t` (or equivalently, ***type*** `& t`) establishes *t* as type "reference to ***type***." So, when *treble_3* is called with the real argument *i, i* is used to initialize the formal reference argument *n. n* therefore acts as an alias for *i*, so `n = 3*n` also assigns *3 * i* to *i*.

If the initializer is a constant or an object of a different type than the reference type, Borland C++ creates a temporary object for which the reference acts as an alias:

```
int& ir = 6;    /* temporary int object created, aliased by ir, gets value 6 */
float f;
int& ir2 = f;  /* creates temporary int object aliased by ir2; f converted
                  before assignment */
ir2 = 2.0      // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

Scope resolution operator ::

The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) name even if it is hidden by a local redeclaration of that name (see page 29 for more on scope):

This code also works if the global i is a file-level static.

```
int i;                        // global i
   ⋮
void func(void) {
       int i=0;               // local i hides global i
       i = 3;                 // this i is the local i
       ::i = 4;               // this i is the global i
       printf ("%d",i);       // prints out 3
       }
```

The **::** operator has other uses with class types, as discussed throughout this chapter.

The new and delete operators

The **new** and **delete** operators offer dynamic storage allocation and deallocation, similar but superior to the standard library functions *malloc* and *free*. See the *Library Reference* for information on *malloc* and *free*.

Syntax for a **new**-expression is one of the following:

<::> **new** <new-args> type-name <**(**initializer**)**>
<::> **new** <new-args> **(**type-name**)** <**(**initializer**)**>

Syntax for a **delete**-expression is one of the following:

<::> **delete** cast-expression
<::> **delete [ ]** cast-expression

The **new** operator must always be supplied with a data type in place of *type-name*. Items surrounded by angle brackets are optional. The optional arguments can be as follows:

- The :: operator invokes the global version of **new**.
- *new-args* can be used to supply additional arguments to **new**. You can use this syntax only if you have an overloaded version of **new** that matches the optional arguments.
- *initializer*, if present, is used to initialize the allocation.

A request for non-array allocation uses the appropriate **operator new()** function. Any request for array allocation calls the appropriate **operator new[]()** function. The selection of an operator with which to allocate class ***Type*** is done as follows:

Allocation of arrays of ***Type***:

Arrays of classes require the default constructor.

1. Attempts to use a class-specific array allocator:
   Type::operator new[]()
2. If the class-specific array allocator is not defined, the global version is used:
   ::operator new[]()

Allocation of non-arrays:

1. Memory for a non-array object of ***Type*** is allocated using ***Type*::operator new()**
2. If the above is not defined, the global **::operator new()** is used

new tries to create an object of type ***Type*** by allocating (if possible) **sizeof(*Type*)** bytes in free store (also called the heap). **new** calculates the size of ***Type*** without the need for an explicit **sizeof** operator. Further, the pointer returned is of the correct type, "pointer to ***Type***," without the need for explicit casting. The storage duration of the new object is from the point of creation until the operator **delete** destroys it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the new object. By default, an allocation failure (such as insufficient or fragmented heap memory) results in the predefined exception *xalloc* being thrown. Your program should always be prepared to catch the *xalloc* exception before trying to access the new object (unless you use a new-handler; see the following section for details).

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

Handling errors

You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use *set_new_handler* and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use *set_new_handler*(0). See the *Library Reference*, Chapter 10, for discussions of *set_new_handler*, *_new_handler*, and the predefined exception *xalloc*.

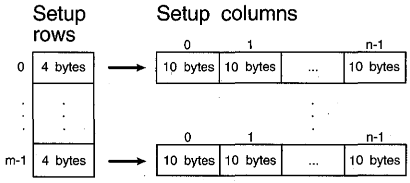
The operator new with arrays

If *Type* is an array, the pointer returned by **operator new[]()** points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];     // OK
mat_ptr = new int[n][10][12];     // OK
mat_ptr = new int[3][][12];       // illegal
mat_ptr = new int[][10][12];      // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

The following example shows you one way to allocate and delete memory for a two-dimensional array. The order of operations taken to allocate the space must be reversed when you delete the space. The illustration shows the amout of space allocated for 32-bit programs.



See the *Library Reference*, Chapter 10, for a description of *xalloc*.

```
/* ALLOCATE A TWO-DIMENSIONAL SPACE, INITIALIZE, AND DELETE IT. */
#include <except.h>
#include <iostream.h>

void display(long double **);
void de_allocate(long double **);

int m = 3;                               // THE NUMBER OF ROWS.
int n = 5;                               // THE NUMBER OF COLUMNS.
int main(void) {
   long double **data;

   try {                                 // TEST FOR EXCEPTIONS.
      data = new long double*[m];        // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < m; j++)
          data[j] = new long double[n];  // STEP 2: SET UP THE COLUMNS
      }
   catch (xalloc) {  // ENTER THIS BLOCK ONLY IF xalloc IS THROWN.
      // YOU COULD REQUEST OTHER ACTIONS BEFORE TERMINATING
      cout << "Could not allocate. Bye ...";
      exit(-1);
      }

   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
          data[i][j] = i + j;            // ARBITRARY INITIALIZATION

   display(data);
   de_allocate(data);
   return 0;
   }

void display(long double **data) {
   for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
             cout << data[i][j] << " ";
       cout << "\n" << endl;
       }
   }

void de_allocate(long double **data) {
   for (int i = 0; i < m;  i++)
       delete[] data[i];                 // STEP 1: DELETE THE COLUMNS

   delete[] data;                        // STEP 2: DELETE THE ROWS
   }
```

produces this output:

```
0 1 2 3 4

1 2 3 4 5

2 3 4 5 6
```

The operator delete with arrays

Arrays are deleted by **operator delete[]()**. You must use the syntax `delete [] expr` when deleting an array. After C++ 2.1, the array dimension should not be specified within the brackets:

```
char * p;

void func()
{
   p = new char[10];    //  allocate 10 chars
   delete[] p;          // delete 10 chars
}
```

C++ 2.0 code required the array size. To allow 2.0 code to compile, Borland C++ issues a warning and ignores any size that is specified. For example, if the preceding example reads `delete[10] p` and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

The ::operator new

By default, if there is no overloaded version of **new**, a request for dynamic memory allocation always uses the global version of **new**, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type *name*, a specific operator called *name*::**operator new()** or *name*::**operator new[]()** can be defined. **new** applied to class *name* objects invokes the appropriate *name*::**operator new** if it is present; otherwise, the global **::operator new** is used.

Initializers with the new operator

Only the **operator new()** function accepts an optional initializer. The array allocator version, **operator new[]()**, does not accept initializers. In the absence of explicit initializers, the object created by **new** contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor (see page 140). The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

Overloading new and delete

The global **::operator new()** and **::operator new[]()** can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific **new** operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a *size_t* as its first argument. To overload the **new** operators, use the following prototypes:

The type *size_t* is defined in stdlib.h

- `void * operator new(size_t Type_size);    // For non-array`
- `void * operator new[](size_t Type_size);  // For arrays`

The Borland C++ compiler provides ***Type_size*** to the **new** operatorAny data type can be substituted for ***Type*** except function names (although a pointer to function is permitted), class declarations, enumeration declarations, **const**, and **volatile**.

The global operators **::operator delete()** and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the global **delete** function can exist in the program.

The user-defined operator **delete** must have a **void** return type and **void*** as its first argument; a second argument of type *size_t* is optional. A class *T* can define at most one version of each of *T*::**operator delete[]()** and *T*::**operator delete()**. To overload the **delete** operators, use the following prototypes:

- `void operator delete(void *Type_ptr, [size_t Type_size]);     // For non-array`
- `void operator delete[](size_t Type_ptr, [size_t Type_size]);  // For arrays`

For example,

```
#include <stdlib.h>

class X {
      :
public:
   void* operator new(size_t size) { return newalloc(size);}
   void operator delete(void* p) { newfree(p); }
   X() { /* initialize here */ }
   X(char ch) { /* and here */ }

   ~X() { /* clean up here */ }
      :
};
```

Destructors are called only if you use the **–xd** compiler option and an exception is thrown.

The *size* argument gives the size of the object being created, and *newalloc* and *newfree* are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of **class** *X* (or objects of classes derived from X that do not have their own overloaded operators **new** and **delete**) invoke the matching user-defined *X*::**operator new()** and *X*::**operator delete()**, respectively.

The *X*::**operator new()**, *X*::**operator new[]()**, *X*::**operator delete()** and *X*::**operator delete[]()** operator functions are static members of *X* whether explicitly declared as **static** or not, so they cannot be virtual functions.

The standard, predefined (global) **new()**, **new[]()**, **delete()**, and **delete[]()** operators can still be used within the scope of *X*, either explicitly with the global scope operator (**::operator new()**, **::operator new[]()**, **::operator delete()**, and **::operator delete[]()**), or implicitly when creating and destroying non-*X* or non-*X*-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
   void* ptr = new char[s]; // standard new called
         :
   return ptr;
}

void X::operator delete(void* ptr)
{
         :
   delete (void*) ptr;     // standard delete called
}
```

The reason for the *size* argument is that classes derived from *X* inherit the *X*::**operator new()** and *X*::**operator new[]()**. The size of a derived class object might differ from that of the base class.

Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, "first-look" syntax for class declarations is

class-key {*<distance-attrib> <distance-attrib>*} *<type-info> class-name*
<: base-list> { *<member-list>* };

class-key is one of **class**, **struct**, or **union**.

The optional *type-info* indicates a request for run-time type information about the class. You can compile with the **–RT** compiler option, or you can use the _ _**rtti** keyword. See the discussion of class *Type_info* in the *Library Reference*, Chapter 10.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class (see page 134). The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes (see page 133).

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that can affect which functions can access which members.

Class memory model specifications

For 16-bit applications only, distance modifiers can be applied to a class declaration. The modifier(s) applied to a class declaration determine the addressing of the class's **this** pointer and the class's table of virtual functions (vtable). The distance modifiers allowed for class declarations, and their effect on the addressing of **this** and the vtable are as follows:

Table 3.1
Class memory model specifications

Modifier	***this**	vtable
_ _**near**	near	near
_ _**far**	far	near
_ _**huge**	far	far
_ _**huge** _ _**near**	near	far
_ _**export**	far	far
_ _**import**	far	far

If you're importing classes that are declared with the modifier _ _**huge**, you must change the modifier to the keyword _ _**import**. The _ _**huge** modifier merely causes far addressing of the virtual tables (the same effect as the **–Vf** compiler option). The _ _**import** modifier makes all function and static addresses default to far.

See Chapter 8 for a discussion of declaration of classes used in DLLs.

Class names

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted. See page 67 for discussion of untagged structures.

Class types

The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```
class X { ... };
X x, &xr, *xptr, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X*/

struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// C would have
// struct Y y, *yptr, yarray[10];

union Z { ... };
Z z, &zr, *zptr, zarray[10];
// C would have
// union Z z, *zptr, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++, they are needed only when the class names, *Y* and *Z*, are hidden (see the following section).

Class name scope

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union**, must be used with the class name. For example,

```
struct S { ... };

int S(struct S *Sptr);

void func(void) {
   S t;        // ILLEGAL declaration: no class key and function S in scope
   struct S s; // OK: elaborated with class key
   S(&s);      // OK: this is a function call
}
```

C++ also allows an incomplete class declaration:

```
class X;  // no members, yet!
```

Incomplete declarations permit certain references to class name X (usually references to pointers to class objects) before the class has been fully defined. See the discussion of structure member declarations beginning page 67. Of course, you must make a complete class declaration with members before you can define and use class objects.

Class objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class. Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

Class member list

The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes), function declarations, and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions

A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

The keyword this

Nonstatic member functions operate on the class type object they are called with. For example, if *x* is an object of class X and *f()* is a member function of X, the function call `x.f()` operates on *x*. Similarly, if *xptr* is a pointer to an X object, the function call `xptr->f()` operates on *xptr. But how does *f* know which instance of X it is operating on? C++ provides *f* with a pointer to *x* called **this**. **this** is passed as a hidden argument in all calls to nonstatic member functions.

this is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If *x.f(y)* is called, for example, where *y* is a member of *X*, **this** is set to *&x* and *y* is set to **this->***y*, which is equivalent to *x.y*.

Inline functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline function*.

The Borland C++ compiler can ignore requests for inline expansion.

Borland C++ can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an inline expansion.

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of *func*:

```
int i;                          // global int

class X {
public:
   char* func(void) { return i; }  // inline by default
   char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit **inline** specifier. The *i* returned by *func* is the **char*** *i* of class *X* (see page 131).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by Borland C++. For example,

```
inline void f1() throw(int)
   {
   // Warning: Functions with exception specifications are not expanded inline
   }
```

Destructors are called by default. See the *User's Guide*, Chapter 3, for information about exception-handling switches.

The remaining restrictions (those listed below) apply only when destructor cleanup is enabled.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
   foo();
   ~foo();
   };

inline void f2(foo& x) {
   // no warning, f2() can be expanded inline
   }

inline void f3(foo x) {
   // Warning: Functions taking class-by-value argument(s) are
   //          not expanded inline in function f3(foo)
   }
```

An inline function that returns a class with a destructor by value will not be expanded inline whenever there are variables or temporaries that need to be destructed within the return expression:

```
struct foo {
   foo();
   ~foo();
   };

inline foo f4() {
   return foo();
   // no warning, f4() can be expanded inline
   }

inline foo f5() {
   foo X;
   return foo(); // Object X needs to be destructed
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f5()
   }

inline foo f6() {
   return ( foo(), foo() );  // temporary in return value
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f6()
   }
```

Static members

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy "exists" for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If *x* is a static member of class *X*, it can be referenced as *X*::*x* (even if objects of class *X* haven't been created yet). It is still possible to access *x* using the normal member access operators. For example, *y.x* and *yptr->x*, where *y* is an object of class *X* and *yptr* is a pointer to an object of class *X*, although the expressions *y* and *yptr* are *not* evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};

void g(void);
{
   X obj;
   func(1, &obj);       // error unless there is a global func()
                        // defined elsewhere

   X::func(1, &obj);    // calls the static func() in X
                        // OK for static functions only
   obj.func(1, &obj);   // so does this (OK for static and
                        // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don't have a **this** pointer, and therefore cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, *func* might be defined as follows:

```
void X::func(int i, X* ptr)
{
   member_int = i;       // which object does member_int
                         // refer to? Error
   ptr->member_int = i;  // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```
class X {
   static int x;
   class inner {
      static float f;
      void func(void);      // nested declaration
      };
};

int X::x = 1;
float X::inner::f = 3.14;  // initialization of nested static
X::inner::func(void) {     /*  define the nested function */  }
```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

Member scope

The expression `X::func()` in the example in the "Inline functions" section on page 128 uses the class name X with the scope access modifier to signify that *func*, although defined "outside" the class, is indeed a member function of X and exists within the scope of X. The influence of X:: extends into the body of the definition. This explains why the *i* returned by *func* refers to *X::i*, the **char*** *i* of X, rather than the global **int** *i*. Without the *X::* modifier, the function *func* would represent an ordinary non-class function, returning the global **int** *i*.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class X can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see page 127). For example,

```
class X {

public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right);   // define elsewhere
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr->i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class *X*, the expression *X*::*m* is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. It is important to note that, even if the class name *X* is hidden by a non-type name, the qualified name *X*::*m* will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class *X* cannot contain objects of class *X*, but can contain pointers or references to objects of class *X* (note the similarity with C's structure and union types).

Nested types

Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the ***xxx::yyy*** notation, except when in the scope of the appropriate class.

A class declared within another class is called a *nested class*. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
   typedef int t;  // 'outer::t' is a typedef name
   struct inner    // 'outer::inner' is a class
   {
      static int x;
   };

   static int x;
      int f();
   class deep;     // nested declaration
};

int outer::x;     // define static data member

int outer::f() {
   t x;           // 't' visible directly here
   return x;
   }

int outer::inner::x;    // define static data member
outer::t x;             //  have to use 'outer::t' here
class outer::deep { };  // define the nested class here
```

With C++ 2.0, any tags or **typedef** names declared inside a class actually belong to the global (file) scope. For example,

```
struct foo
{
   enum bar { x };   // 2.0 rules: 'bar' belongs to file scope
                     // 2.1 rules: 'bar' belongs to 'foo' scope
};

bar x;
```

The preceding fragment compiles without errors. But because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Member access control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

Friend function declarations are not affected by access specifiers (see page 137).

public	The member can be used by any function.
private	The member can be used only by member functions and friends of the class it's declared in.

protected Same as for **private**. Additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in the next section.)

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

```
class X {
   int i;    // X::i is private by default
   char ch;  // so is X::ch
public:
   int j;    // next two are public
   int k;
protected:
   int l;    // X::l is protected
};

struct Y {
   int i;    // Y::i is public by default
private:
   int j;    // Y::j is private
public:
   int k;    // Y::k is public
};

union Z {
   int i;    // public by default; no other choice
   double d;
};
```

Base and derived class access

When you declare a derived class *D*, you list the base classes *B1*, *B2*, ... in a comma-delimited *base-list*:

class-key D : *base-list* { *<member-list>* }

Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basest of the base classes, those that do not inherit.

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) *D* can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by *D*? *D* might want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

When declaring *D*, you can use the access specifier **public, protected**, or **private** in front of the classes in the *base-list*:

Unions cannot have base classes, and unions cannot be used as base classes.

```
class D : public B1, private B2, ... {
   :
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they *can* alter the access attributes of base members as viewed by the derived class.

The default is **private** if *D* is a **class** declaration, and **public** if *D* is a **struct** declaration.

The derived class inherits access attributes from a base class as follows:

- **public** base class: **public** members of the base class are **public** members of the derived class. **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **protected** base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **private** members of the base class remain **private** to the base class.

Note that **private** members of a base class are always inaccessible to member functions of the derived class *unless* **friend** declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {              // default for class is private A
   :
}
```

```
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */
class Y : B, public C {     // override default for C
   ⋮
}

/* struct S is derived from D */
struct S : D {              // default for struct is public D
   ⋮
}

/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E {   // override default for D
                            // E is public by default
   ⋮
}
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations of the derived class. For example,

```
class B {
   int a;                // private by default
public:
   int b, c;
   int Bfunc(void);
};

class X : private B {    // a, b, c, Bfunc are now private in X
   int d;                // private by default, NOTE: a is not
                         // accessible in X
public:
   B::c;                 // c was private, now is public
   int e;
   int Xfunc(void);
};

int Efunc(X& x);         // external to B and X
```

The function *Efunc()* can use only the public names *c*, *e*, and *Xfunc()*.

The function *Xfunc()* is in *X*, which is derived from **private** *B*, so it has access to

- The "adjusted-to-public" *c*
- The "private-to-*X*" members from *B*: *b* and *Bfunc()*
- *X*'s own private and public members: *d*, *e*, and *Xfunc()*

However, *Xfunc()* cannot access the "private-to-*B*" member, *a*.

Virtual base classes

With multiple inheritance, a base class can't be specified more than once in a derived class:

```
class B { ...};
class D : B, B { ... };  // Illegal
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }   // OK
```

In this case, each object of class *Z* will have two sub-objects of class *B*. If this causes problems, the keyword **virtual** can be added to a base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class *Z* has only one sub-object of class *B*.

Friends of classes

A **friend** *F* of a class *X* is a function or class, although not a member function of *X*, with full access rights to the private and protected members of *X*. In all other respects, *F* is a normal function with respect to scope, declarations, and definitions.

Since *F* is not a member of *X*, it is not in the scope of *X*, and it cannot be called with the *x.F* and *xptr->F* selector operators (where *x* is an *X* object and *xptr* is a pointer to an *X* object).

If the specifier **friend** is used with a function declaration or definition within the class *X*, it becomes a friend of *X*.

friend functions defined within a class obey the same inline rules as member functions (see page 128). **Friend** functions are not affected by their position within the class or by any access specifiers. For example,

```
class X {
   int i;                          // private to X
   friend void friend_func(X*, int);
/* friend_func is not private, even though it's declared in the private section
   */
public:
     void member_func(int);
};
```

```
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }


X xobj;

/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class *Y* into friends of class *X* with a single declaration:

```
class Y;                    // incomplete declaration
class X {
   friend Y;
   int i;
   void member_funcX();
};

class Y; {                  // complete the declaration
   void friend_X1(X&);
   void friend_X2(X*);
      ⋮
};
```

The functions declared in *Y* are friends of *X*, although they have no **friend** specifiers. They can access the private members of *X*, such as *i* and *member_funcX*.

It is also possible for an individual member function of class *X* to be a friend of class *Y*:

```
class X {
      ⋮
   void member_funcX();
}

class Y {
   int i;
   friend void X::member_funcX();
      ⋮
};
```

Class friendship is not transitive: *X* friend of *Y* and *Y* friend of *Z* does not imply *X* friend of *Z*. Friendship is not inherited.

Constructors and destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even **void**).
- They cannot be inherited, though a derived class can call the base class's constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be **virtual**, but constructors cannot. (See page 148.)
- You can't take their addresses.

```
int main(void)
{
        ⋮
     void *ptr = base::base;     // illegal
        ⋮
}
```

- Constructors and destructors can be generated by Borland C++ if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
   ⋮
  X *p;
   ⋮
  p->X::~X();                  // legal call of destructor
  X::X();                      // illegal call of constructor
   ⋮
}
```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.

- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class *X* to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class *X*. Note that this rule applies only to any constructor with *one* parameter and no initializers that use the "**=**" syntax.

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                        // illegal: Y(X(1)) not tried
```

If **class** *X* has one or more constructors, one of them is invoked each time you define an object *x* of **class** *X*. The constructor creates *x* and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the *main* function is called. When the **#pragma startup** directive is used to install a function prior to the *main* function, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
   X();   // class X constructor
};
```

A **class** *X* constructor cannot take *X* as an argument:

```
class X {
public:
   X(X);                         // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the *copy constructor*. A constructor that accepts no parameters is called the *default constructor*. The default constructor and the copy constructor are discussed in the following sections.

Constructor defaults

The default constructor for **class** X is one that takes no arguments; it usually has the form `X::X()`. If no user-defined constructors exist for a class, Borland C++ generates a default constructor. On a declaration such as `X x`, the default constructor creates the object *x*.

➡ Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes *no* arguments and must not be confused with, say, `X::X(int = 0)`, which can be called with no arguments as a default constructor, or can take an argument.

You should avoid ambiguity in calling constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};

int main()
{
   X one(10);  // OK; uses X::X(int)
   X two;      // illegal; ambiguous whether to call X::X() or
               // X::X(int = 0)
   return 0;
}
```

The copy constructor

A copy constructor for **class** X is one that can be called with a single argument of type `X`, as follows:

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

Borland C++ generates a copy constructor for **class** *X* if one is needed and no other constructor has been defined in **class** *X*. The copy constructor that is generated by the Borland C++ compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor only if your program creates aggregate, complex types such as **class**, **struct**, and arrays.

See also the discussion of member-by-member class assignment beginning on page 154. You should define the copy constructor if you overload the assignment operator.

Overloading constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X {
   int    integer_part;
   double double_part;
public:
   X(int i)    { integer_part = i; }
   X(double d) { double_part = d; }
};

int main() {
   X one(10);   // invokes X::X(int) and sets integer_part to 10
   X one(3.14); // invokes X::X(double) setting double_part to 3.14
   return 0;
}
```

Order of calling constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y();   // base class constructor
X();   // derived class constructor
```

For the case of multiple base classes,

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y();  // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // nonvirtual base class
X();   // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy
             // base is constructed only once
base2();     // nonvirtual base of virtual base level2
             // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

Class initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
   int i;
public:
   X();          // function bodies omitted for clarity
   X(int x);
   X(const X&);
};

void main()
{
   X one;        // default constructor invoked
   X two(1);     // constructor X::X(int) is used
   X three = 1;  // calls X::X(int)
   X four = one; // invokes X::X(const X&) for copy
   X five(two);  // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways:

- It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

- An initializer list can be used prior to the function body:

```
class X
{
   int a, b, &c;  // Note the reference variable.
public:
   X(int i, int j) : a(i), b(j), c(a) {}
};
```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to *x::a* and 2 to *x::b*. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Base class constructors must be declared as either **public** or **protected** to be called from a derived class.

```
class base1
{
   int x;
public:
   base1(int i) { x = i; }
};

class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};

class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};
```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of *base1* with the value 5 and *base2* with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
  int a, b;
public:
  X(int i, j) :  a(i), b(a+j) {}
};
```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to *x::a* and 2 to *x::b*.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
};

class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized a
};
```

With this class setup, a call of `derived d(1)` will *not* result in a value of 10 for the base class member *x*. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
   ⋮
}
```

Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
   ~X(){};  // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

Invoking destructors

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see page 142).

atexit, #pragma exit, and destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the *main* function, are destroyed as they go out of scope. The order of execution at the end of a Borland C++ program is as follows:

- *atexit()* functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

exit and destructors

When you call *exit* from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

abort and destructors

If you call *abort* anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**.

```
#include <stdlib.h>
class X {
public:
   ⋮
   ~X(){};
   ⋮
};

void* operator new(size_t size, void *ptr)
{
   return ptr;
}

char buffer[sizeof(X)];

void main() {
   X* pointer = new X;
   X* exact_pointer;

   exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
   ⋮

   delete pointer;                 // delete used to destroy pointer
   exact_pointer->X::~X();         // direct call used to deallocate
}
```

virtual destructors

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
class color
{
public:
   virtual ~color();    // virtual destructor for color
};

class red : public color
{
public:
   ~red();             // destructor for red is also virtual
};

class brightred: public red
{
public:
   ~brightred();       // brightred's destructor also virtual
};
```

The previously listed classes and these declarations:

```
color *palette[3];

palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;
```

produce these results:

```
delete palette[0];
//   The destructor for red is called, followed by the
//   destructor for color.

delete palette[1];
// The destructor for brightred is called, followed by ~red
// and ~color.

delete palette[2];
// The destructor for color is invoked.
```

However, if no destructors are declared as virtual, **delete** *palette*[0], **delete** *palette*[1], and **delete** *palette*[2] would all call only the destructor for class *color*. This would incorrectly destruct the first two elements, which were actually of type *red* and *brightred*.

Operator overloading

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

The keyword **operator** followed by the operator symbol is called the *operator function name;* it is used like a normal function name when defining the new (overloaded) action of the operator.

All the operators listed on page 81 can be overloaded except for:

. .* :: ?:

The preprocessing symbols # and ## also cannot be overloaded.

The **=**, **[ ]**, **( )**, and **->** operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for **enum** types. Any attempt to overload a global version of these operators is a compile-time error.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function can't alter the number of arguments or the precedence and associativity rules (see Table 2.11 on page 78) applying to normal operator use.

The following example extends the class *complex* to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in a natural syntax.

Some of the issues illustrated by the example are

- The default constructor is defined. This is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of *cvector* types, you specify exactly the actions to be taken.
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The **const** version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for *cvector* types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a **friend**. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.
- The stream insertion operator is overloaded to naturally display a *cvector*. Large objects that don't display well on a limited size screen might require a different display strategy.

Source

See the *Library Reference*, Chapter 8, for a description of **class** *complex*.

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
#pragma warn -inl      // IGNORE not expanded inline WARNINGS.
#include <complex.h>  // THIS ALREADY INCLUDES iostream.h

// COMPLEX VECTORS
class cvector {
   int size;
   complex *data;
```

```
public:
   cvector() { size = 0; data = NULL; };
   cvector(int i = 5) : size(i) {    // DEFAULT VECTOR SIZE.
      data = new complex[size];
      for (int j = 0; j < size; j++)
           data[j] = j + (0.1 * j);  // ARBITRARY INITIALIZATION.
      };

   /* THIS VERSION IS CALLED IN main() */
   complex& operator [](int i) { return data[i]; };
   /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR */
   const complex& operator [](int i) const { return data[i]; };

   cvector operator +(cvector& A) {  // ADDITION OPERATOR
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < size; i++)
          result[i] = data[i] + A.data[i];
      return result;
      };

   /* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER OF
      THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENSURE
      PROPER MULTIPLICATION. */
   friend cvector operator *(int scalar, cvector& A) {
      cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
      for (int i = 0; i < A.size; i++)
        result.data[i] = scalar * A.data[i];
      return result;
      }

   /* THE STREAM INSERTION OPERATOR. */
   friend ostream& operator <<(ostream& out_data, cvector& C) {
      for (int i = 0; i < C.size; i++)
           out_data << "[" << i << "]=" << C.data[i] << "    ";
      cout << endl;
      return out_data;
      };

   cvector( const cvector &C ) {  // COPY CONSTRUCTOR
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
           data[i] = C[i];
     }

   cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
      if (this == &C) return *this;

      delete[] data;
      size = C.size;
      data = new complex[size];
      for (int i = 0; i < size; i++)
```

```
            data[i] = C[i];
         return *this;
      };

      virtual ~cvector() { delete[] data; }; // DESTRUCTOR
      };

   int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
      cvector cvector1(4), cvector2(4), result(4);

      // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
      cvector1[3] = complex(3.3, 102.8);
      cout << "Here is cvector1:" << endl;
      cout << cvector1;

      cvector2[3] = complex(33.3, 81);
      cout << "Here is cvector2:" << endl;
      cout << cvector2;

      result = cvector1 + cvector2;
      cout << "The result of vector addition:" << endl;
      cout << result;

      result = 10 * cvector2;
      cout << "The result of 10 * cvector2:" << endl;
      cout << result;
      return 0;
      }
```

Output

```
Here is cvector1:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(3.3, 102.8)
Here is cvector2:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(33.3, 81)
The result of vector addition:
[0]=(0, 0)   [1]=(2.2, 0)   [2]=(4.4, 0)   [3]=(36.6, 183.8)
The result of 10 * cvector2:
[0]=(0, 0)   [1]=(11, 0)   [2]=(22, 0)   [3]=(333, 810)
```

Overloading operator functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);   // same as c3 = c1 + c2
```

Apart from **new** and **delete**, which have their own rules (see page 122), an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions **=**, **( )**, **[ ]** and **->** must be nonstatic member functions.

Overloaded operators and inheritance

With the exception of the assignment function **operator =( )** (see the section beginning on page 154), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could be further overloaded for Y.

Unary operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If **@** represents a unary operator, *x*@ and *x*@ can both be interpreted as either *x*.**operator@()** or **operator@**(*x*), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Beginning with C++ 2.1, when an **operator++** or **operator- –** is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix **operator++** or **operator- –**. You can only overload a postfix **operator++** or **operator- –** by defining it as a member function taking an **int** parameter or as a nonmember function taking one class and one **int** parameter. The **int** parameter is used by the compiler only to distinguish operator prototypes—it is not used in the operator definition. See page 76 for an example of postfix and prefix increment operator overloading.

When only the prefix version of an **operator++** or **operator- –** is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning and calls the prefix operator. If a function *func* calls the postfix operator the compiler issues the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in function
func()
```

```
Warning: Overloaded prefix 'operator --' used as a postfix operator in function
func()
```

Binary operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a nonmember function (usually **friend**) taking two arguments. If **@** represents a binary operator, *x*@*y* can be interpreted as either *x*.**operator@**(*y*) or **operator@**(*x*,*y*) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Assignment operator=

The assignment **operator=( )** can be overloaded by declaring a nonstatic member function. For example,

```
class String {
         ⋮
   String& operator = (String& str);
         ⋮
   String (String&);
   ~String();
}
```

This code, with suitable definitions of *String*::**operator =()**, allows string assignments *str1* **=** *str2* just like other languages. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class *X*, there is no user-defined operator **=**, the operator **=** is defined by default as a member-by-member assignment of the members of class *X*:

```
X& X::operator = (const X& source)
{
   // memberwise assignment
}
```

Function call operator( )

The function call

primary-expression (<*expression-list*>)

is considered a binary operator with operands *primary-expression* and *expression-list* (possibly empty). The corresponding operator function is **operator()**. This function can be user-defined for a class *X* (and any derived classes) only by means of a nonstatic member function. A call *X*(*arg1*, *arg2*), where *X* is an object of class *X*, is interpreted as *X*.**operator()**(*arg1*,*arg2*).

Subscript operator[ ]

Similarly, the subscripting operation

primary-expression [*expression*]

is considered a binary operator with operands *primary-expression* and *expression*. The corresponding operator function is **operator[]**; this can be user-defined for a class *X* (and any derived classes) only by means of a nonstatic member function. The expression *X*[*y*], where *X* is an object of class *X*, is interpreted as *x*.**operator[]**(*y*).

Class member access operator->

Class member access using

primary-expression –> expression

is considered a unary operator. The function **operator->** must be a nonstatic member function. The expression *x*->*m*, where *x* is a **class** X object, is interpreted as (*x*.**operator->()**)->*m*, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

Polymorphic classes

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.

virtual functions

See the following section for a discussion of pure virtual functions.

virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a *pure* function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

```
virtual int funct1(void);       // A virtual function declaration.
virtual int funct2(void) = 0;   // A pure function declaration.

virtual void funct3(void) = 0 { // This is a valid declaration.
   // Some code in here.
   };
```

When you declare **virtual** functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a **friend** of another class.
- They cannot be a static member.

A **virtual** function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a **virtual** function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined **virtual** functions differing only in return type is discussed below.) A redefined function is said to *override* the base class function.

You can also declare the functions `int Base::Fun(int)` and `int Derived::Fun(int)` even when they are not **virtual**. In such a case, `int Derived::Fun(int)` is said to *hide* any other versions of `Fun(int)` that exist in any base classes. In addition, if class *Derived* defines other versions of *Fun()*, (that is, versions of *Fun()* with different signatures) such versions are said to be *overloaded* versions of *Fun()*.

virtual function return types

Generally, when redefining a **virtual** function, you cannot change just the function return type. To redefine a **virtual** function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the **virtual** function mechanism is ignored.

However, for certain **virtual** functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when *both* of the following conditions are met:

- The overridden **virtual** function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class *B* and class *D* (derived publicly from *B*) each contain a **virtual** function *vf*, then if *vf* is called for an object *d* of *D*, the call made is `D::vf()`, even when the access is via a pointer or reference to *B*. For example,

```
struct X {};           // Base class.
struct Y : X {};       // Derived class.

struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();    // Return type is a pointer to base. This can
                       //  be overridden.
   };

class D : public B {
```

```
public:
   virtual void vf1(); // Virtual specifier is legal but redundant.
   void vf2(int);      // Not virtual, since it's using a different
                       //  arg list. This hides B::vf2().
// char vf3();         // Illegal: return-type-only change!
   void f();
   Y*   pf();          // Overriding function differs only
                       //  in return type. Returns a pointer to
                       //  the derived class.
   };

void extf() {
   D d;        // Instantiate D
   B* bp = &d; // Standard conversion from D* to B*
               // Initialize bp with the table of functions
               // provided for object d. If there is no entry for a
               // function in the d-table, use the function
               //  in the B-table.
   bp->vf1(); // Calls D::vf1
   bp->vf2(); // Calls B::vf2 since D's vf2 has different args
   bp->f();   // Calls B::f (not virtual)

   X* xptr = bp->pf();   // Calls D::pf() and converts the result
                         //  to a pointer to X.

   D* dptr = &d;
   Y* yptr = dptr->pf();  // Calls D::pf() and initializes yptr.
                          //  No further conversion is done.
   }
```

The overriding function *vf1* in *D* is automatically **virtual**. The **virtual** specifier *can* be used with an overriding function declaration in the derived class. If other classes will be derived from *D*, the **virtual** keyword is required. If no further classes will be derived from *D*, the use of **virtual** is redundant.

The interpretation of a **virtual** function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

Abstract classes

An *abstract class* is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {       // abstract class
   point center;
      :
public:
   where() { return center; }
   move(point p) { center = p; draw(); }
   virtual void rotate(int) = 0; // pure virtual function
   virtual void draw() = 0;      // pure virtual function
   virtual void hilite() = 0;    // pure virtual function
      :
}

shape x;           // ERROR: attempt to create an object of an abstract class
   shape* sptr;    // pointer to abstract class is OK
   shape f();      // ERROR: abstract class cannot be a return type
int g(shape s);    // ERROR: abstract class cannot be a function argument type
shape& h(shape&); // reference to abstract class as return
                   // value or function argument is OK
```

Suppose that *D* is a derived class with the abstract class *B* as its immediate base class. Then for each pure virtual function *pvf* in *B*, if *D* doesn't provide a definition for *pvf*, *pvf* becomes a pure member function of *D*, and *D* will also be an abstract class.

For example, using the class *shape* previously outlined,

```
class circle : public shape { // circle derived from abstract class
   int radius;                // private
public:
   void rotate(int) { }       // virtual function defined: no action
                              //  to rotate a circle
   void draw();               // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

C++ scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as "enclosing scope" and "point of declaration."

Class scope

The name *M* of a member of a class *X* has class scope "local to *X*"; it can be used only in the following situations:

- In member functions of *X*
- In expressions such as `x.M`, where *x* is an object of *X*
- In expressions such as *xptr->M*, where *xptr* is a pointer to an object of *X*
- In expressions such as `X::M` or `D::M`, where *D* is a derived class of *X*
- In forward references within the class of which it is a member

Names of functions declared as friends of *X* are not members of *X*; their names simply have enclosing scope.

Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: `X::M`. A hidden file scope (global) name can be referenced with the unary operator **::** (for example, **::***g*). A class name *X* can be hidden by the name of an object, function, or enumerator declared within the scope of *X*, regardless of the order in which the names are declared. However, the hidden class name *X* can still be accessed by prefixing *X* with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name *x* is immediately after its complete declaration but before its initializer, if one exists.

C++ scoping rules summary

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator **::**, *and* if the name is not qualified by the binary **::** operator or the member selection operators **.** and **–>**, then the name must be a global object, function, or enumerator.
- If the name *n* appears in any of the forms *X***::***n*, *x.n* (where *x* is an object of *X* or a reference to *X*), or *ptr***->***n* (where *ptr* is a pointer to *X*), then *n* is the

name of a member of *X* or the member of a class from which *X* is derived.

- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class *X* must either be declared in the block it occurs in or in an enclosing block, be a member of class *X* or a base class of *X*, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer (see *ctor-initializer* in the class declarator syntax in Table 2.3 on page 38) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

Templates

Templates, also called *generics* or *parameterized types*, let you construct a family of related functions or classes. This section introduces the basic concept of templates, then provides some specific points. The template syntax is shown below:

Template-declaration:
  template < *template-argument-list* > *declaration*

template-argument-list:
  template-argument
  template-argument-list, template argument

template-argument:
  type-argument
  argument-declaration

type-argument:
  class *identifier*

template-class-name:
  template-name < template-arg-list >

template-arg-list:
  template-arg
  template-arg-list , template-arg

template-arg:
  expression
  type-name

Function templates

Consider a function *max(x, y)* that returns the larger of its two arguments. *x* and *y* can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters *x* and *y* to be declared at compile time. Without using templates, many overloaded versions of *max* are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger. For example, the following code could be followed by yet other versions of *max*:

```
int max(int x, int y) {
   return (x > y) ? x : y;
   }

long max(long x, long y) {
   return (x > y) ? x : y;
   }
   ⋮
```

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the **#define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of *max(x, y)* is to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be:

```
class Compare
{
 public:
  int max(int, int);    // Results in syntax error;
                        //  this gets expanded!!!
      ⋮
};
```

By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

Function template definition

```
template <class T> T max(T x, T y)
{
   return (x > y) ? x : y;
};
```

The data type is represented by the template argument **<class T>**. When used in an application, the compiler generates the appropriate function according to the data type actually used in the call:

```
int i;
Myclass a, b;

int j = max(i,0);        // arguments are integers
Myclass m = max(a,b);    // arguments are type Myclass
```

Any data type (not just a class) can be used for ***<class T>***. The compiler takes care of calling the appropriate **operator>()**, so you can use *max* with arguments of any type for which **operator>()** is defined.

Overriding a template function

The previous example is called a *function template* (or *generic function*). A specific instantiation of a function template is called a *template function*. Template function instantiation occurs when you take the function address, or when you call the function with defined (nongeneric) data types. You can override the generation of a template function for a specific type with a nontemplate function:

```
#include <string.h>

char *max(char *x, char *y)
{
   return(strcmp(x,y)>0) ?x:y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

Template function argument matching

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b)
{
        return  (a > b) ? a : b;
}

void    f(int i, char c)
{
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // no match for max(int,char)
        max(c, i);              // no match for max(char,int)
}
```

This code results in the following error messages:

Could not find a match for 'max(int,char)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)

If the user explicitly declares a template function, however, this function, participates fully in overload resolution. For example,

Explicit template function

```
template<class T> T max(T a, T b)
{
        return  (a > b) ? a : b;
}
int     max(int,int);           // declare max(int,int) explicitly

void    f(int i, char c)
{
        max(i, i);              // calls max(int ,int )
        max(c, c);              // calls max(char,char)
        max(i, c);              // calls max(int,int)
        max(c, i);              // calls max(int,int)
}
```

When searching for an exact match for template function parameters trivial conversions are considered to be exact matches. For example:

```
template<class T> void func(const T a)
{
   ⋮
}
   ⋮
func(0); // This is illegal under ANSI C++: unresolved func(int).
         // However, Borland C++ now allows func(const int) to be called.
```

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. For example:

```
template<class T> class B
{
      ⋮
};

template<class T> class D : public B<T>
{
      ⋮
};

template<class T> void func(B<T> *b)
{
      ⋮
}

func(new D<int>); // This is illegal under ANSI C++:
                  //   unresolved func(D<int> *).
                  // However, Borland C++ calls func(B<int> *).
```

The conversion from derived class to base class is allowed only for template parameters, non-template parameters still require exact matches. For example:

```
class B
{
      ⋮
};

class D : public B
{
      ⋮
};

template<class T> void bar(T ignored, B *b)
{
      ⋮
};
```

```
bar(0, new D);  // Illegal under CFRONT 3.0, ANSI C++ and Borland C++:
                // unresolved external bar(int, D *), D * -> B *
                // is not considered an exact match.
```

Class templates

A class template (also called a *generic class* or *class generator*) lets you define a pattern for class definitions. Generic container classes are good examples. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a ***T*** parameter to the class, the system will generate type-safe class definitions on the fly:

Class template definition

```
#include <iostream.h>

template <class T> class Vector
{
   T *data;
   int size;

 public:
   Vector(int);
   ~Vector() {delete[] data;}
   T& operator[](int i) {return data[i];}
};

        // Note the syntax for out-of-line definitions:
template <class T> Vector<T>::Vector(int n)
{
   data = new T[n];
   size = n;
};

int main()
{
   Vector<int> x(5);// Generate a vector of ints

   for (int i = 0; i < 5; ++i)
      x[i] = i;
   for (i = 0; i < 5; ++i)
      cout << x[i] << ' ';
   cout << '\n';
   return 0;
}

// Output will be: 0 1 2 3 4
```

As with function templates, an explicit *template class* definition can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol *Vector* must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

For a more complete implementation of a vector class, see the file vectimp.h in the container class library source code, found in the BC4\INCLUDE\CLASSLIB subdirectory. Also see Chapter 7.

Arguments

Although these examples use only one template argument, multiple arguments are allowed. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Nontype template arguments such as *size* can have default values. The value supplied for a nontype template argument must be a constant expression:

```
const int N = 128;
int i = 256;

Buffer<int, 2*N> b1;// OK
Buffer<float, i> b2;// Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

Angle brackets

Be careful when using the right angle-bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between *x* and 100 would prematurely close the template argument list.

Nested templates also require careful use of angle brackets. It is a common error to omit a space between multiple '>' closing delimiters of a nested template class name.

Note the use of delimiters in the following example:

```
template <class T> struct foo{};
foo<foo<int>> x;
```

The Borland C++ compiler allows such a construct with the following warning:

This is a compile-time error if you compile with –A option.

```
Warning myfile.cpp: Use '> >' for nested templates instead of '>>'
```

Type-safe generic lists

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition (a generic list class) are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get:

```
class GList
{
 public:
   void insert( void * );
   void *peek();
       ⋮
};
```

You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList
{
 public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo *)GList::peek(); }
          ⋮
};
```

This is type-safe. *insert* will only take arguments of type pointer-to-*Foo* or object-derived-from-*Foo*, so the underlying container will hold only pointers that in fact point to something of type *Foo*. This means that the cast in *FooList::peek()* is always safe, and you've created a true *FooList*. To do the same for a *BarList*, a *BazList*, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and be type-safe, you can once again use templates:

Type-safe generic list class definition

```
template <class T> class List : public GList
{
public:
   void insert( T *t ) { GList::insert( t ); }
   T *peek() { return (T *)GList::peek(); }
       ⋮
};

  List<Foo> fList; // create a FooList class and an instance
                      named fList.
  List<Bar> bList; // create a BarList class and an instance
                      named bList.
  List<Baz> zList; // create a BazList class and an instance
                      named zList.
```

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. Because there's no code generated by the type conversions from each wrapper class, there's no run-time overhead imposed by this type safety.

Eliminating pointers

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is beneficial if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

Template definition that eliminates pointers

```
template <class T> aBase
{
      ⋮
 private:
  T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf>
{
      ⋮
};
```

All the functions in *aBase* can call functions defined in *aFilebuf* directly, without having to go through a pointer. And if any of the functions in *aFilebuf* can be inlined, you'll get a speed improvement, because templates allow them to be inlined.

Template compiler switches

The **–Jg** family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler. For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

See the *User's Guide*, Chapter 3, for a summary of template options and switches.

–Jg Default value of the switch. All template instances first encountered when this switch value is in effect will be generated, such that if several compilation units generate the same template instance, the linker will merge them to produce a single copy of the instance. This is the most convenient approach to generating template instances because it's almost entirely automatic. Note, though, that to be able to generate the template instances, the compiler must have the function body (in case of a template function) or bodies of member functions and definitions for static data members (in case of a template class).

–Jgd Instructs the compiler to generate public definitions for template instances. This is similar to **–Jg**, but if more than one compilation unit generates a definition for the same template instance, the linker will report public symbol redefinition errors.

–Jgx Instructs the compiler to generate external references to template instances. Some other compilation unit must generate a public definition for that template instance (using the **–Jgd** switch) so that the external references can be satisfied.

Using template switches

When using the **–Jg** family of switches, there are two basic approaches for generating template instances:

The first approach is to include the function body (for a function template) or member function and static data member definitions (for a template class) in the header file that defines the particular template, and use the default setting of the template switch (**-Jg**). If some instances of the template are user-defined, the declarations (prototypes, for example) for them should be included in the same header but preceded by **#pragma option –Jgx**. This lets the compiler know it should not generate those particular instances.

Here's an example of a template function header file:

```
// Declare a template function along with its body

template<class T> void sort(T* array, int size)
{
   ⋮
body of template function goes here
   ⋮
}

// Sorting of 'int' elements done by user-defined instance

#pragma option -Jgx

extern void  sort(int* array, int size);

// Restore the template switch to its original state

#pragma option -Jg.
```

If the preceding header file is included in a C++ source file, the *sort* template can be used without worrying about how the various instances are generated (with the exception of *sort* for **int** arrays, which is declared as a user-defined instance, and whose definition must be provided by the user).

The second approach is to compile all of the source files comprising the program with the **–Jgx** switch (causing external references to templates to be generated); this way, template bodies don't need to appear in header files. To provide the definitions for all of the template instances, add a file (or files) to the program that includes the template bodies (including any user-defined instance definitions), and list all the template instances needed in the rest of the program to provide the necessary public symbol definitions. Compile the file (or files) with the **–Jgd** switch.

Here's an example:

```
//   vector.h

template <class elem, int size> class vector
{
   elem * value;

public:

   vector();

   elem & operator[](int index) { return value[index]; }
};


//   MAIN.CPP

#include "vector.h"

// Tell the compiler that the template instances that follow
// will be defined elsewhere.

#pragma option -Jgx

// Use two instances of the 'vector' template class.

vector<int,100> int_100;
vector<char,10> char_10;

int main()
{
   return int_100[0] + char_10[0];
}

//   TEMPLATE.CPP

#include <string.h>

#include "vector.h"
```

```
// Define any template bodies

template <class elem, int size> vector<elem, size>::vector()
{
   value = new elem[size];
   memset(value, 0, size * sizeof(elem));
}

// Generate the necessary instances

#pragma option -Jgd

typedef vector<int,100> fake_int_100;
typedef vector<char,10> fake_char_10;
```

CHAPTER 4

Exception handling

This chapter describes the Borland C++ error-handling mechanisms generally referred to as *exception handling*. All exception handling constructs are available for 16- and 32-bit implementations. The Borland C++ implementation of C++ exception handling is consistent with the proposed ANSI specification. The exception-handling mechanisms that are available in C programs are referred to as *structured exceptions*. Borland C++ provides full compiling, linking, and debugging support for C programs with structured exceptions. See the section "C-based structured exceptions" on page 181, and the *User's Guide*, Chapter 3, for a discussion of compiler options for programming with exceptions.

C++ exception handling

C++ exceptions can be handled only in a **try/catch** construct.

The C++ language defines a standard for exception handling. The standard ensures that the power of object-oriented design is supported throughout your program.

In accordance with the specifications of the ANSI/ISO C++ working paper, Borland C++ supports the termination exception-handling model. When an abnormal situation arises at run time, the program could terminate. However, throwing an exception lets you gather information at the throw point that could be useful in diagnosing the causes that led to failure. You can also specify in the exception handler the actions to be taken before the program terminates. Only synchronous exceptions are handled, meaning that the cause of failure is generated from within the program. An event such as *Ctrl*-C (which is generated from outside the program) is not considered to be an exception.

The **catch** and **throw** keywords are not allowed in a C program.

Syntax:

---

try-block:
  try *compound-statement handler-list*

handler-list:
  handler handler-list $_{opt}$

handler:
  catch *(exception-declaration) compound-statement*

exception-declaration:
  type-specifier-list declarator
  type-specifier-list abstract-declarator
  type-specifier-list
  ...

throw-expression:
  throw *assignment-expression* $_{opt}$

The *try-block* is a statement that specifies the flow of control as the program executes. The try-block is designated by the **try** keyword. Braces after the keyword surround a program block that can generate exceptions. The language structure specifies that any exceptions that occur should be raised within the *try-block*. See page 101 for a discussion about statements.

The handler is a block of code designed to handle an exception. The C++ language requires that at least one handler be available immediately after the try-block. There should be a handler for each exception that the program can generate.

When the program encounters an abnormal situation for which it is not designed, you can transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.

The exception-handling mechanism requires the use of three keywords: **try**, **catch**, and **throw**. The *try-block* specified by **try** must be followed immediately by the *handler* specified by **catch**. If an exception is thrown in the *try-block*, program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so could result in abnormal termination of the program.

Exception declarations

Although C++ allows an exception to be of almost any type, it is useful to make exception classes. The exception object is treated exactly the way any object would be treated. An exception carries information from the point where the exception is thrown to the point where the exception is caught. This is information that the program user will want to know when the program encounters some anomaly at run time.

Predefined exceptions, specified by the C++ language, are documented in the *Library Reference*, Chapter 10. Borland C++ provides additional support for exceptions. These extensions are documented in the *Library Reference*, Chapter 4. See also page 120 for a discussion of the **new** operator and the predefined *xalloc* exception.

Throwing an exception

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the **try** keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

1. The program searches for a matching handler
2. If a handler is found, the stack is unwound to that point
3. Program control is transferred to the handler

If no handler is found, the program will call the *terminate* function. If no exceptions are thrown, the program executes in the normal fashion.

A *throw expression* is also referred to as a throw-point. You can specify whether an exception can be thrown by using one of the following syntax specifications:

```
1. throw throw_expression;
2. throw;
3. void my_func1() throw (A, B)
   {
   // Body of function.
   }
4. void my_func2() throw ()
   {
   // Body of this function.
   }
```

The first case specifies that *throw_expression* is to be passed to a handler.

The second case specifies that the exception currently being handler is to be thrown again. An exception must currently exist. Otherwise, *terminate* is called.

The third case specifies a list of exceptions that *my_func1* can throw. No other exceptions should propagate out of *my_func1*. If an exception other than *A* or *B* is generated within *my_func1*, it is considered to be an unexpected exception and program control will be transferred to the *unexpected* function. By default, the *unexpected* function ends with a call to *abort* but it can throw an exception. See the *Library Reference*, Chapter 10, for a description of *unexpected*.

The final case specifies that *my_func2* should throw no exceptions. If some other function (for example, **operator new**) in the body of *my_func2* throws an exception, such an exception should be caught and handled within the body of *my_func2*. Otherwise, such an exception is a violation of *my_func2* exception specification. The *unexpected* function is then called.

When an exception occurs, the throw expression initializes a temporary object of the type ***T*** (to match the type of argument *arg*) used in *throw(**T** arg)*. ➡ Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

Handling an exception

The exception handler is indicated by the **catch** keyword. The handler must be placed immediately after the try-block. The keyword **catch** can also occur immediately after another **catch**. Each handler will only handle an exception that matches, or can be converted to, the type specified in its argument list. The possible conversions are listed after the try-block syntaxes.

The following syntaxes, following the try-block, are valid:

```
try {
   // Include any code that might throw an exception
}


1. catch (T X)
   {
   // Take some actions
   }
2. catch ( ... )
   {
   // Take some actions
    }
```

The first statement is specifically defined to handle an object of type ***T***. If the argument is ***T***, ***T&***, **const** ***T***, or **const** ***T&***, the handler will accept an object of type ***X*** if any of the following are true:

- ***T*** and ***X*** are of the same type
- ***T*** is an accessible base class for ***X*** in the throw expression

- ***T*** is a pointer type and ***X*** is a pointer type that can be converted to ***T*** by a standard pointer conversion at the throw point

The statement **catch** (...) will handle any exception, regardless of type. This statement, if used, must be the last handler for its try-block.

Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call *terminate.*

Exception handlers are evaluated in the order that they are encountered. An exception is caught when its type matches the type in the **catch** statement. Once a type match is made, program control is transferred to the handler. The stack will have been unwound upon entering the handler. The handler specifies what actions should be taken to deal with the program anomaly.

A **goto** statement can be used to transfer program control out of a handler or try-block but such a statement can never be used to enter a handler or try-block.

After the handler has executed, the program can continue at the point after the last handler for the current try-block. No other handlers are evaluated for the current exception.

Exception specifications

The C++ language makes it possible for you to specify any exceptions that a function can throw. This *exception specification* can be used as a suffix to the function declaration. The syntax for exception specification is as follows:

exception-specification:
    throw (*type-id-list* $_{opt}$)

    type-id-list:
    type-id
    type-id-list, type-id

The function suffix is not considered to be part of the function's type. Consequently, a pointer to a function is not affected by the function's exception specification. Such a pointer checks only the function's return and argument types. Therefore, the following is legal:

```
void f2(void) throw();      // Should not throw exceptions
void f3(void) throw (BETA); // Should only throw BETA objects
void (* fptr)();            // Pointer to a function returning void
fptr = f2;
fptr = f3;
```

Extreme care should be taken when overriding virtual functions. Again, because the exception specification is not considered part of the function type, it is possible to violate the program design. In the following example, the derived class *BETA::vfunc* is defined so that it throws an exception—a departure from the original function declaration.

```
class ALPHA {
public:
   virtual void vfunc(void) throw () {};  // Exception specification
};

class BETA : public ALPHA {
   struct BETA_ERR {};
   void vfunc(void) throw( BETA_ERR ) {}; // Exception specification is changed
};
```

The following are examples of functions with exception specifications.

```
void f1();                // The function can throw any exception

void f2() throw();        // Should not throw any exceptions

void f3() throw( A, B* ); // Can throw exceptions publicly derived from A,
                          // or a pointer to publicly derived B
```

The definition and all declarations of such a function must have an exception specification containing the same set of type-id's. If a function throws an exception not listed in its specification, the program will call *unexpected*. This is a run-time issue—it will not be flagged at compile time. Therefore, care must be taken to handle any exceptions that can be thrown by elements called within a function.

Source

```
// HOW TO MAKE EXCEPTION-SPECIFICATIONS AND HANDLE ALL EXCEPTIONS
#include <iostream.h>

class ALPHA{};                // EXCEPTION DECLARATION
ALPHA _a;
void f3(void) throw (ALPHA) { // WILL THROW ONLY TYPE-ALPHA OBJECTS
   cout << "f3() was called" << endl;
   throw(_a);
   }
```

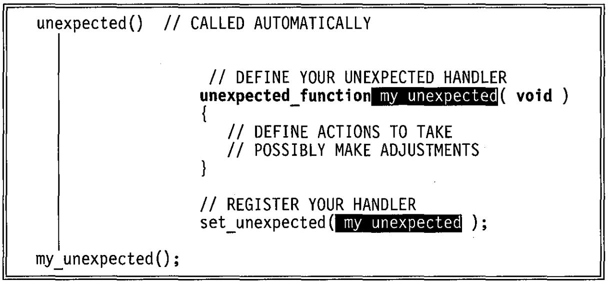
```
void f2(void) throw() {       // SHOULD NOT THROW EXCEPTIONS
   try {                      // WRAP ALL CODE IN A TRY-BLOCK
      cout << "f2() was called" << endl;
      f3();
      }
   /* IF MORE FUNCTIONS ARE ADDED, ANY OF WHICH THROW EXCEPTIONS, THE FOLLOWING
      HANDLER WILL CATCH ALL OF THEM. */
   catch ( ... ) {            // TRAP ALL EXCEPTIONS
      cout << "An exception was caught in f2()!" << endl;
      }
   }

int main(void) {
   try {
      f2();
      return 0;
      }
   catch ( ... ) {
      cout << "Need more handlers!";
      return 1;
      }
   }
```

Output

```
f2() was called
f3() was called
An exception was caught in f2()!
```

If an exception is thrown that is not listed in the exception specification, the *unexpected* function will be called. The following diagrams illustrate the sequence of events that can occur when *unexpected* is called. See the *Library Reference,* Chapter 10, for a description of the *set_terminate, set_unexpected,* and unexexpected functions. The chapter also describes the *terminate_function* and *unexpected_function* types.

Program behavior when a function is registered with *set_unexpected()*



```
unexpected()  // CALLED AUTOMATICALLY

                    // DEFINE YOUR UNEXPECTED HANDLER
                   unexpected_function my unexpected( void )
                   {
                      // DEFINE ACTIONS TO TAKE
                      // POSSIBLY MAKE ADJUSTMENTS
                   }

                   // REGISTER YOUR HANDLER
                   set_unexpected( my unexpected );

my_unexpected();
```

Program behavior when no function is registered with *set_unexpected()* but there is a function registered with *set_terminate()*

```
unexpected()  // CALLED AUTOMATICALLY
 |
terminate()
 |
 |               // DEFINE YOUR TERMINATION SCHEME
 |               terminate_function my_terminate( void )
 |                  {
 |                  // TAKE ACTIONS BEFORE TERMINATING
 |                  // SHOULD NOT THROW EXCEPTIONS
 |                  exit(1); // MUST END SOMEHOW.
 |                  }
 |
 |               // REGISTER YOUR TERMINATION FUNCTION
 |               set_terminate( my_terminate )
 |
 |
my_terminate()
// PROGRAM ENDS.
```

Constructors and destructors

When an exception is thrown, the copy constructor is called for the thrown value. The copy constructor is used to initialize a temporary object at the throw point. Other copies can be generated by the program. See page 3 for a discussion of the copy constructor.

Destructors are called by default. See the *User's Guide*, Chapter 3, for information about exception-handling switches.

When program flow is interrupted by an exception, destructors are called for all automatic objects that were constructed since the beginning of the try-block was entered. If the exception was thrown during construction of some object, destructors will be called only for those objects that were fully constructed. For example, if an array of objects was under construction when an exception was thrown, destructors will be called only for the array elements that were already fully constructed.

When a C++ exception is thrown, the stack is unwound. By default, during stack unwinding, destructors are called for automatic objects. You can use the **-xd** compiler option to switch the default off.

Unhandled exceptions

If an exception is thrown and no handler is found it, the program will call the *terminate* function. The following diagram illustrates the series of events that can occur when the program encounters an exception for which no handler can be found. See the *Library Reference,* Chapter 10, for a description of the functions named in the diagram.

Default program behavior for unhandled exceptions

```
terminate();
 |
 |
abort();
// PROGRAM ENDS.
```

C-based structured exceptions

For portability, you can use the *try* and *except* macros defined in excpt.h.

Borland C++ provides support for program development that makes use of structured exceptions. You can compile and link a C source file that contains an implementation of structured exceptions. In a C program, the ANSI-compatible keywords used to implement structured exceptions are **_ _except**, **_ _finally**, and **_ _try**. Note that the **_ _finally** and **_ _try** keywords can appear only in C programs.

For try-except exception-handling implementations the syntax is as follows:

try-block:
  _ _try *compound-statement* (in a C module)
  try *compound-statement* (in a C++ module)

handler:
  _ _except *(expression) compound-statement*

For try-finally termination implementations the syntax is as follows:

try-block:
  _ _try *compound-statement*

termination:
  _ _finally *compound-statement*

See your Win32 documentation for additional details on the implementation of structured exceptions for 16- and 32-bit platforms.

Using C-based exceptions in C++

Borland C++ allows substantial interaction between C and C++ error handling mechanisms. The following interactions are supported:

- C structured exceptions can be used in C++ programs.
- C++ exceptions cannot be used in a C program because C++ exceptions require that their handler be specified by the **catch** keyword and **catch** is not allowed in a C program.
- An exception generated by a call to the *RaiseException* function is handled by a **try/_ _except** or **_ _try/_ _except** block. All handlers of **try/catch** blocks are ignored when *RaiseException* is called.
- The following C exception helper functions can be used in C and C++ programs:
  - *GetExceptionCode*
  - *GetExceptionInformation*

- *SetUnhandledExceptionFilter*
- *UnhandledExceptionFilter*
- Borland C++ does not require that the *UnhandledExceptionFilter* function be used only in the except filter of **_ _try/_ _except** or **try/_ _except** blocks. However, program behavior is undefined when this function is called outside of the **_ _try/_ _except** or **try/_ _except** block.

See your Win32 documentation for a description of the exception helper functions and *RaiseException*.

The full functionality of an **_ _except** block is allowed in C++. If an exception is generated in a C module, it is possible to provide a handler-block in a separate calling C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

These actions are consistent with the design of structured exceptions.

```
/* In PROG.C */
void func(void) {
      ⋮
   /* generate an exception */
   RaiseException( /* specify your arguments */ );
      ⋮
}

// In CALLER.CPP
// How to test for C++ or C-based exceptions.
#include <excpt.h>
#include <iostream.h>

int main(void) {
   try
   {            // test for C++ exceptions
      try
      {         // test for C-based structured exceptions
         func();
      }
      _ _except( /* filter-expression */ )
      {
      cout << "A structured exception was generated.";
      ⋮
      /* specify actions to take for this structured exception */
```

```
    return -1;
    }
    return 0;
  }
  catch ( ... )
  {
  //.handler for any C++ exception
  cout << "A C++ exception was thrown.";
  return 1;
  }
}
```

The **__try/__finally** ensures that the code in the **__finally** block is executed no matter how the flow within the **__try** exits. The **__finally** keyword is not allowed in a C++ program and the **__try/__finally** block is not supported in a C++ program.

Even though the **__try/__finally** block is not supported in a C++ program, a C-based exception generated by the operating system or the program can still result in proper stack unwinding by using local objects within destructors. Any module compiled with the **–xd** compiler option will have destructors invoked for all objects with **auto** storage. Stack unwinding occurs from the point where the exception is thrown to the point where the exception is caught.

Destructors are called by default. See the *User's Guide*, Chapter 3, for information about exception-handling switches.

CHAPTER 5

The preprocessor

Although Borland C++ uses an integrated single-pass compiler for its IDE and command-line versions, it is useful to retain the terminology associated with earlier multipass compilers.

With a multipass compiler, a first pass of the source text pulls in any include files, tests for any conditional compilation directives, expands any macros, and produces an intermediate file for further compiler passes. Since the IDE and command-line versions of the Borland C++ compiler perform this first pass with no intermediate output, Borland C++ provides independent preprocessors, CPP.EXE and CPP32.EXE, that produce such an output file. The independent preprocessor is useful as a debugging aid because it lets you see the net result of include directives, conditional compilation directives, and complex macro expansions.

The following discussion, therefore, applies both to the CPP and CPP32 preprocessors and to the preprocessor functionality built into the Borland C++ compiler.

The preprocessor detects *preprocessor directives* (also known as *control lines*) and parses the tokens embedded in them.

The Borland C++ preprocessor includes a sophisticated macro processor that scans your source code before the compiler itself gets to work. The preprocessor gives you great power and flexibility in the following areas:

- Defining macros that reduce programming effort and improve your source code legibility. Some macros can also eliminate the overhead of function calls.
- Including text from other files, such as header files containing standard library and user-supplied function prototypes and manifest constants.
- Setting up conditional compilations for improved portability and for debugging sessions.

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program.

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

The full syntax for Borland C++'s preprocessor directives is given in the next table.

Table 5.1: Borland C++ preprocessing directives syntax

preprocessing-file:
    group

group:
    group-part
    group group-part

group-part:
    <pp-tokens> newline
    if-section
    control-line

if-section:
    if-group <elif-groups> <else-group> endif-line

if-group:
    #if *constant-expression newline <group>*
    #ifdef *identifier newline <group>*
    #ifndef *identifier newline <group>*

elif-groups:
    elif-*group*
    elif-*groups elif-group*

elif-group:
    #elif *constant-expression newline <group>*

else-group:
    #else *newline <group>*

endif-line:
    #endif *newline*

control-line:
    #include *pp-tokens newline*
    #define *identifier replacement-list newline*
    #define *identifier lparen <identifier-list>) replacement-list newline*
    #undef *identifier newline*
    #line *pp-tokens newline*
    #error *<pp-tokens> newline*
    #pragma *<pp-tokens> newline*
    #pragma **warn** *action abbreviation newline*
    #pragma **inline** *newline*
    # *newline*

action: one of
    + - .

abbreviation:
    nondigit nondigit nondigit

lparen:
    the left parenthesis character without preceding whitespace

replacement-list:
    <pp-tokens>

pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token

preprocessing-token:
    header-name (only within an #include directive)
    identifier (no keyword distinction)
    constant
    string-literal
    operator
    punctuator
    each non-whitespace character that cannot be one of the preceding

header-name:
    <h-char-sequence>

h-char-sequence:
    h-char
    h-char-sequence h-char

h-char:
    any character in the source character set except the newline (\n) or greater than (>) character

newline:
    the newline character

Null directive

The null directive consists of a line containing the single character #. This directive is always ignored.

The #define and #undef directives

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Simple #define macros

In the simple case with no parameters, the syntax is as follows:

#define *macro_identifier <token_sequence>*

Each occurrence of *macro_identifier* in your source code following this control line will be replaced in the original position with the possibly empty *token_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.

An empty token sequence results in the effective removal of each affected macro identifier from the source code:

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
   ⋮
puts(HI);         /* expands to puts("Have a nice day!"); */
puts(NIL);        /* expands to puts(""); */
puts("empty");    /* NO expansion of empty! */
/* NOR any expansion of the empty within comments! */
```

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: the expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor:

```
#define GETSTD #include <stdio.h>
   ⋮
GETSTD      /* compiler error */
```

GETSTD will expand to `#include <stdio.h>`. However, the preprocessor itself will not obey this apparently legal directive, but will pass it verbatim to the compiler. The compiler will reject `#include <stdio.h>` as illegal input. A macro won't be expanded during its own expansion (so `#define A A` won't expand indefinitely).

The #undef directive

You can undefine a macro using the **#undef** directive:

#undef *macro_identifier*

This line detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined.

No macro expansion occurs within **#undef** lines.

The state of being *defined* or *undefined* is an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

```
#define BLOCK_SIZE 512
   ⋮
buff = BLOCK_SIZE*blks;  /* expands as 512*blks *
   ⋮
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
   ⋮
#define BLOCK_SIZE 128   /* redefinition */
   ⋮
buf = BLOCK_SIZE*blks;   /* expands as 128*blks */
   ⋮
```

Attempting to redefine an already defined macro identifier results in a warning unless the new definition is *exactly* the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
   #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE isn't currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Assembly language programmers must resist the temptation to write:

```
#define BLOCK_SIZE = 512 /* ?? token sequence includes the = */
```

The –D and –U options

Identifiers can be defined and undefined using the command-line compiler options **–D** and **–U**. See the *User's Guide*, Chapter 3.

The command line

```
BCc -Ddebug=1; paradox=0; X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
#define X
#undef mysym
```

in the program.

The Define option

Identifiers can be defined, but not explicitly undefined, from the IDE. Use the Define option to explicitly define a macro.

Identifiers can be defined, but not explicitly undefined, from the IDE. See the *User's Guide*, Chapter 3.

Keywords and protected words

It is legal but not recommended to use Borland C++ keywords as macro identifiers:

```
#define int long     /* legal but probably catastrophic */
#define INT long     /* legal and possibly useful */
```

The following predefined global identifiers *cannot* appear immediately following a **#define** or **#undef** directive:

Note the double underscores, leading and trailing.

__STDC__
__FILE__
__LINE__
__DATE__
__TIME__

Macros with parameters

The following syntax is used to define a macro with parameters:

#define *macro_identifier*(*<arg_list>*) *token_sequence*

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the **(**. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *placeholder*.

Such macros are called by writing

macro_identifier<whitespace>(*<actual_arg_list>*)

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls (see page 192).

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the **#define** line: there must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*. For example,

```
#define CUBE(x) ((x)*(x)*(x))
   ⋮
int n,y;
n = CUBE(y);
```

results in the following replacement:

$$n = ((y) * (y) * (y));$$

Similarly, the last line of

```
#define SUM (a,b) ((a) + (b))
   ⋮
int i,j,sum;
sum = SUM(i,j);
```

expands to $sum = ((i) + (j))$. The reason for the apparent glut of parentheses will be clear if you consider the call

```
n = CUBE(y+1);
```

Without the inner parentheses in the definition, this would expand as $n = y+1*y+1 *y+1$, which is parsed as

```
n = y + (1*y) + (1*y) + 1;  // != (y+1) cubed unless y=0 or y = -3!
```

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Note the following points when using macros with argument lists:

- **Nested parentheses and commas.** The *actual_arg_list* can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters:

```
#define ERRMSG(x, str) showerr("Error",x,str)
#define  SUM(x,y) ((x) + (y))
   ⋮
ERRMSG(2, "Press Enter, then Esc");
/* expands to showerr("Error",2,"Press Enter, then Esc");
return SUM(f(i,j), g(k,l));
/* expands to return ((f(i,j)) + (g(k,l))); */
```

- **Token pasting with ##.** You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers; for example, given the definition

  ```
  #define VAR(i,j) (i##j)
  ```

  the call `VAR(x,6)` would expand to (*x*6). This replaces the older (nonportable) method of using `(i/**/j)`.

- **Converting to strings with #.** The # symbol can be placed in front of a formal macro argument to convert the actual argument to a string after replacement. So, given the following macro definition:

  ```
  #define TRACE(flag) printf(#flag "=%d\n",flag)
  ```

  the code fragment

  ```
  int highval = 1024;
  TRACE(highval);
  ```

  becomes

  ```
  int highval = 1024;
  printf("highval" "= %d\n", highval);
  ```

  which, in turn, is treated as

  ```
  int highval = 1024;
  printf("highval=%d\n", highval);
  ```

- **The backslash for line continuation.** A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions:

  ```
  #define WARN "This is really a single-\
  line warning"
     ⋮
  puts(WARN);
  /* screen will show: This is really a single-line warning */
  ```

- **Side effects and other dangers.** The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once. Compare **CUBE** and **cube** in the following example:

Final value of b depends on what your compiler does to the expanded expression.

```
int cube(int x) {
   return x*x*x;
}
#define CUBE(x) ((x)*(x)*(x))
   :
int b = 0, a = 3;
b = cube(a++);
/* cube() is passed actual arg = 3; so b = 27; a now = 4 */
a = 3;
b = CUBE(a++);
/* expands as ((a++)*(a++)*(a++)); a now = 6 */
```

File inclusion with #include

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three versions:

The angle brackets are real tokens, not metasymbols that imply header_name is optional.

#include *<header_name>*

#include *"header_name"*

#include *macro_identifier*

The first and second versions imply that no macro expansion will be attempted; in other words, *header_name* is never scanned for macro identifiers. *header_name* must be a valid file name with an extension (traditionally .h for header) and optional path name and path delimiters.

The third version assumes that neither **<** nor " appears as the first non-whitespace character following **#include**. Further, it assumes the existence of a macro definition that will expand the macro identifier into a valid delimited header name with either of the *<header_name>* or *"header_name"* formats.

The preprocessor removes the **#include** line and replaces it with the entire text of the header file at that point in the source code. The source code itself isn't changed, but the compiler "sees" the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header_name*, only that directory will be searched.

The difference between the <*header_name*> and "*header_name*" formats lies in the searching algorithm employed in trying to locate the include file; these algorithms are described in the following two sections.

Header file search with <header_name>

The <*header_name*> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file isn't located in any of the default directories, an error message is issued.

Header file search with "header_name"

The "*header_name*" version specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file isn't found there, the search continues in the include directories as in the <*header_name*> situation.

The following example clarifies these differences:

```
#include <stdio.h>
/* header in standard include directory */

#define myinclud "C:\BC4\INCLUDE\MYSTUFF.H"
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\\BC4\\INCLUDE\\MYSTUFF.H" */

#include  myinclud
/* macro expansion */

#include "myinclud.h"
/* no macro expansion */
```

After expansion, the second **#include** statement causes the preprocessor to look in C:\BC4\INCLUDE\MYSTUFF.H and nowhere else. The third **#include** causes it to look for MYINCLUD.H in the current directory, then in the default directories.

Conditional compilation

Borland C++ supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The #if, #elif, #else, and #endif conditional directives

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. They are used as follows:

#if *constant-expression-1*
<*section-1*>
<**#elif** *constant-expression-2 newline section-2*>
⋮
<**#elif** *constant-expression-n newline section-n*>

<**#else** <*newline*> *final-section*>

#endif
⋮

If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the Borland C++ compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with *next-section*. In the *false* case, control passes to the next **#elif** line (if any) where *constant-expression-2* is evaluated. If true, *section-2* is processed, after which control moves on to the matching **#endif**. Otherwise, if *constant-expression-2* is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be carefully balanced with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

The operator defined

The **defined** operator offers an alternative, more flexible way of testing if combinations of identifiers are defined. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) or **defined** identifier (the parentheses are optional) evaluates to 1 (true) if the symbol has been previously defined (using **#define**) and has not been subsequently undefined (using **#undef**); otherwise, it evaluates to 0 (false). The following two directives are therefore the same:

```
#if defined(mysym)
```

```
#ifdef mysym
```

The advantage is that you can use **defined** repeatedly in a complex expression following the **#if** directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

The #ifdef and #ifndef conditional directives

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

#ifndef tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif** given in the previous section.

An identifier defined as NULL is considered to be defined.

The #line line control directive

You can use the **#line** command to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program. The syntax

#line *integer_constant* <"*filename*">

indicates that the following source line originally came from line number *integer_constant* of *filename*. Once the *filename* has been registered, subsequent **#line** commands relating to that file can omit the explicit *filename* argument.

The inclusion of stdio.h means that the preprocessor output will be somewhat large.

```
/* TEMP.C: An example of the #line directive */

#include <stdio.h>

#line 4 "junk.c"
void main()
{

    printf(" in line %d of %s",_ _LINE_ _,_ _FILE_ _);
#line 12 "temp.c"
    printf("\n");
    printf(" in line %d of %s",_ _LINE_ _,_ _FILE_ _);
#line 8
    printf("\n");
    printf(" in line %d of %s",_ _LINE_ _,_ _FILE_ _);
}
```

If you run TEMP.C through CPP (`cpp temp.c`), you'll get an output file TEMP.I; that looks something like this:

Most of the stdio.h portion has been eliminated.

```
temp.c 1:
C:\BC4\INCLUDE\STDIO.H 1:
C:\BC4\INCLUDE\STDIO.H 2:
C:\BC4\INCLUDE\STDIO.H 3:
   :
C:\BC4\INCLUDE\STDIO.H 212:
C:\BC4\INCLUDE\STDIO.H 213:
temp.c 2:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6: printf(" in line %d of %s",6,"junk.c");
junk.c 7:
temp.c 12: printf("\n");
```

```
temp.c 13: printf(" in line %d of %s",13,"temp.c");
temp.c 14:
temp.c 8: printf("\n");
temp.c 9: printf(" in line %d of %s",9,"temp.c");
temp.c 10: }
temp.c 11:
```

If you then compile and run TEMP.C, you'll get this output:

```
in line 6 of junk.c
in line 13 of temp.c
in line 9 of temp.c
```

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

The #error directive

The **#error** directive has the following syntax:

#error *errmsg*

This generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional statement that is true for the undesired case.

For example, suppose you **#define** *MYVAL*, which must be either 0 or 1. You could then include the following conditional statement in your source code to test for an incorrect value of *MYVAL*:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

The #pragma directive

The **#pragma** directive permits implementation-specific directives of the form:

#pragma *directive-name*

With **#pragma**, Borland C++ can define the directives it wants without interfering with other compilers that support **#pragma**. If the compiler doesn't recognize *directive-name*, it ignores the **#pragma** directive without any error or warning message.

Borland C++ supports the following **#pragma** directives:

- **#pragma argsused**
- **#pragma codeseg**
- **#pragma comment**
- **#pragma exit**
- **#pragma hdrfile**
- **#pragma hdrstop**
- **#pragma inline**
- **#pragma intrinsic**
- **#pragma option**
- **#pragma saveregs**
- **#pragma startup**
- **#pragma warn**

#pragma argsused

The **argsused** pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message

```
"Parameter name is never used in function func-name"
```

#pragma codeseg

The **codeseg** directive lets you name the segment, class, or group where functions are allocated.

The syntax is as follows:

#pragma codeseg *<seg_name> <"seg_class"> <group>*

If the pragma is used without any of its optional arguments, the default code segment is used for function allocation.

#pragma comment

The **comment** directive lets you write a comment record into an OBJ file. A library module that is not specified in the linker's response-file can be specified by the **comment** LIB directive.

Use the following syntax to make comment records:

#pragma comment(LIB, *"lib_module_name"*)

This causes the linker to include the *lib_module_name* module as the last library.

#pragma exit and #pragma startup

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the *main* function is called) or upon program exit (just before the program terminates through **_exit**).

The syntax is as follows:

#pragma startup *function-name <priority>*
#pragma exit *function-name <priority>*

The specified *function-name* must be a previously declared function taking no arguments and returning **void**:

```
void func(void);
```

Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100. For example,

Note that the function name used in **pragma startup** or **exit** must be defined (or declared) before the pragma line is reached.

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

void startFunc(void)
{
   printf("Startup function.\n");
}

#pragma startup startFunc 64
/* priority 64 --> called first at startup */

void exitFunc(void)
{
   printf("Wrapping up execution.\n");
}

#pragma exit exitFunc
/* default priority is 100 */

void main(void)
{
#if defined(_Windows)
   _InitEasyWin();
#endif
   printf("This is main.\n");
}
```

#pragma hdrfile

This directive sets the name of the file in which to store precompiled headers. The syntax is

#pragma hdrfile *"FILENAME*.CSM"

See Appendix C in the *User's Guide* for more details.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option **–H=*filename*** to change the name of the file used to store precompiled headers.

#pragma hdrstop

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers. See the *User's Guide*, Appendix C for more on precompiled headers.

#pragma inline

This directive is equivalent to the **–B** command-line compiler option or the IDE inline option. It tells the compiler there is inline assembly language code in your program (see Chapter 10). The syntax is

#pragma inline

This is best placed at the top of the file, because the compiler restarts itself with the **–B** option when it encounters #**pragma inline**.

#pragma intrinsic

#pragma intrinsic is documented in Chapter 3 of the *User's Guide*.

#pragma option

Use #**pragma option** to include command-line options within your program code. The syntax is

#pragma option [*options...*]

The command-line compiler options are defined in Chapter 3 in the *User's Guide*.

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. Any of the toggle options (such as **–a** or **–K**) can be turned on and off (as on the command line). For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This lets you temporarily change an option, then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include

–B
–c
–d*name*
–D*name* = *string*
–e*filename*
–E
–F*x*
–H
–I*filename*
–L*filename*
–l*xset*
–M
–o
–P
–Q
–S
–T
–U*name*
–V
–X
–Y

You can use #**pragma**s, #**include**s, #**define**, and some #**if**s in the following cases:

- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an #**if**, #**ifdef**, #**ifndef** or #**elif** directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear *only* in a #**pragma option** command before these events. These options are

–E*filename*
–f*
–i#
–m*
–n*path*
–o*filename*
–u
–z*

Other options can be changed anywhere. The following options affect the compiler only if they get changed between functions or object declarations:

–1
–2
–3
–4
–5
–a
–ff
–G
–h
–k
–N
–O
–p
–r
–rd
–v
–y
–Z

The following options can be changed at any time and take effect immediately:

The options can appear followed by a dot (.) to reset the option to its command-line state.

–A (see Note)
–b
–C
–d
–g*n*
–j*n*
–K
–w*xxx*
–zE
–zF
–zH

Note The #**pragma option –A** statement isn't equivalent to the command-line option **–A**. The command-line option recognizes only ANSI-specified keywords. The #**pragma option –A** prefixes non-ANSI keywords with double underscores. In effect, this causes such keywords to comply with ANSI requirements.

#pragma saveregs

The **saveregs** pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

#pragma warn

The **warn** pragma lets you override specific **–w*xxx*** command-line options or check **D**isplay Warnings settings in the Options | Compiler | Messages dialog boxes.

For example, if your source code contains the directives

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned on, the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began.

A complete list of the three-letter abbreviations and the warnings to which they apply is given in Chapter 3 in the *User's Guide*. Note that you must use only the three letters that identify warning; do not use the prefix **–w**, which is intended for the command-line option.

Predefined macros

Borland C++ predefines certain global identifiers, each of which is discussed in this section. These predefined macros are also known as *manifest constants*. Except for _ _cplusplus and _Windows, each of the global identifiers starts and ends with two underscore characters (_ _).

The following symbols are defined based on the memory model chosen at compile time:

_ _COMPACT_ _
_ _HUGE_ _
_ _LARGE_ _
_ _MEDIUM_ _
_ _SMALL_ _
_ _TINY_ _

Only one symbol is defined for any given compilation; the others, by definition, are undefined. For example, if you compile with the small model, the _ _SMALL_ _ macro is defined and the rest are not, so that the directive

```
#if defined(_ _SMALL_ _)
```

will be true, while

```
#if defined(_ _LARGE_ _)
```

(or any of the others) will be false. The actual value for any of these defined macros is 1.

__BCOPT__	This macro is defined (to the string "1") in any compiler that has an optimizer.
__BCPLUSPLUS__	This macro is specific to Borland's C and C++ family of compilers. It is defined for C++ compilation only. If you've selected C++ compilation, it is defined as 0x320, a hexadecimal constant. This numeric value will increase in later releases.
__BORLANDC__	This macro is specific to Borland's C and C++ family of compilers. It is defined as 0x400, a hexadecimal constant. This numeric value will increase in later releases.
__CDECL__	This macro is specific to Borland's C and C++ family of compilers. It signals that the Pascal calling convention isn't being used. The macro is set to the integer constant 1 if calling was not used; otherwise, it is undefined.
__CONSOLE__	This macro is available only for the 32-bit compiler. When defined, the macro indicates that the program is a console application.
__cplusplus	This macro is defined as 1 if in C++ mode; otherwise it is undefined. This lets you write a module that will be compiled sometimes as C and sometimes as C++. Using conditional compilation, you can control which C and C++ parts are included.
__DATE__	This macro provides the date the preprocessor began processing the current source file (as a string literal). Each inclusion of __DATE__ in a given file contains the same value, regardless of how long the processing takes. The date appears in the format *mmm dd yyyy*, where *mmm* equals the month (Jan, Feb, and so forth), *dd* equals the day (1 to 31, with the first character of *dd* a blank if the value is less than 10), and *yyyy* equals the year (1990, 1991, and so forth).
__DLL__	This macro is specific to Borland's C and C++ family of compilers. It is defined as 1 if you compile a module to generate code for Windows DLLs; otherwise it remains undefined.
__FILE__	This macro provides the name of the current source file being processed (as a string literal). This macro changes whenever the compiler processes an #**include** directive or a #**line** directive, or when the include file is complete.

__LINE__

This macro provides the number of the current source-file line being processed (as a decimal constant). Normally, the first line of a source file is defined as 1, though the **#line** directive can affect this. See page 196 for information on the **#line** directive.

__MSDOS__

This macro is true for the 16-bit compiler and always false for the 32-bit compiler.

__MT__

This macro is available only for the 32-bit compiler. The macro is defined as 1 if **–WM** option is used. It specifies that the multithread library is to be linked.

__OVERLAY__

This macro is specific to Borland's C and C++ family of compilers. It is predefined as 1 if you compile a module with the **–Y** option (enable overlay support). If you don't enable overlay support, this macro is undefined.

__PASCAL__

This macro is specific to Borland's family of compilers. It signals that the Pascal calling convention has been used. The macro is set to the integer constant 1 if used; otherwise, it remains undefined.

__STDC__

This macro is defined as the constant 1 if you compile for ANSI compatibility; otherwise, it is undefined.

__TCPLUSPLUS__

This macro is specific to Borland's family of compilers. It is defined for C++ compilation only. If you've selected C++ compilation, it is defined as 0x0320, a hexadecimal constant. This numeric value will increase in later releases.

__TEMPLATES__

This macro is specific to Borland's family of compilers. It is defined as 1 for C++ files (meaning that Borland C++ supports templates); otherwise, it is undefined.

__TIME__

This macro keeps track of the time the preprocessor began processing the current source file (as a string literal).

As with __DATE__, each inclusion of __TIME__ contains the same value, regardless of how long the processing takes. It takes the format *hh:mm:ss*, where *hh* equals the hour (00 to 23), *mm* equals minutes (00 to 59), and *ss* equals seconds (00 to 59).

__TLS__	The thread local storage macro is always true when the 32-bit compiler is used. See page 53 for a discussion of the **__thread** keyword.
__TURBOC__	This macro is specific to Borland's C and C++ family of compilers. It is defined as 0x0400, a hexadecimal constant. This numeric value will increase in later releases.
__WIN32__	This macro is always defined for the 32-bit compiler. It is defined for console and GUI applications.
_Windows	This macro indicates that Windows-specific code is being generated. It is defined by default for Borland C++. The macro is always defined for console and GUI applications.

CHAPTER 6

Using C++ iostreams

This chapter provides a brief, practical overview of how to use C++ stream I/O. For specific details on the C++ stream classes and their member functions, see the *Library Reference*.

Stream input/output in C++ (commonly referred to as *iostreams*, or just *streams*) provide all the functionality of the *stdio* library in ANSI C. Iostreams are used to convert typed objects into readable text, and vice versa. Streams can also read and write binary data. The C++ language lets you define or overload I/O functions and operators that are then called automatically for corresponding user-defined types.

What is a stream?

A stream is an abstraction referring to any flow of data from a source (or *producer*) to a *sink* (or *consumer*). We also use the synonyms *extracting*, *getting*, and *fetching* when speaking of inputting characters from a source; and *inserting*, *putting*, or *storing* when speaking of outputting characters to a sink. Classes are provided that support console output (constrea.h), memory buffers (iostream.h), files (fstream.h), and strings (strstrea.h) as sources or sinks (or both).

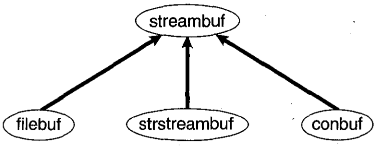
The iostream library

The *iostream* library has two parallel families of classes: those derived from *streambuf*, and those derived from *ios*. Both are low-level classes, each doing a different set of jobs. All stream classes have at least one of these two classes as a base class. Access from *ios*-based classes to *streambuf*-based classes is through a pointer.

The streambuf class

The *streambuf* class provides an interface to memory and physical devices. *streambuf* provides underlying methods for buffering and handling streams when little or no formatting is required. The member functions of the *streambuf* family of classes are used by the *ios*-based classes. You can also derive classes from *streambuf* for your own functions and libraries. The buffering classes *conbuf*, *filebuf*, and *strstreambuf* are derived from *streambuf*.

Figure 6.1
Class streambuf and its derived classes



The ios class

The class *ios* (and hence any of its derived classes) contains a pointer to a *streambuf*. It performs formatted I/O with error-checking using a *streambuf*.

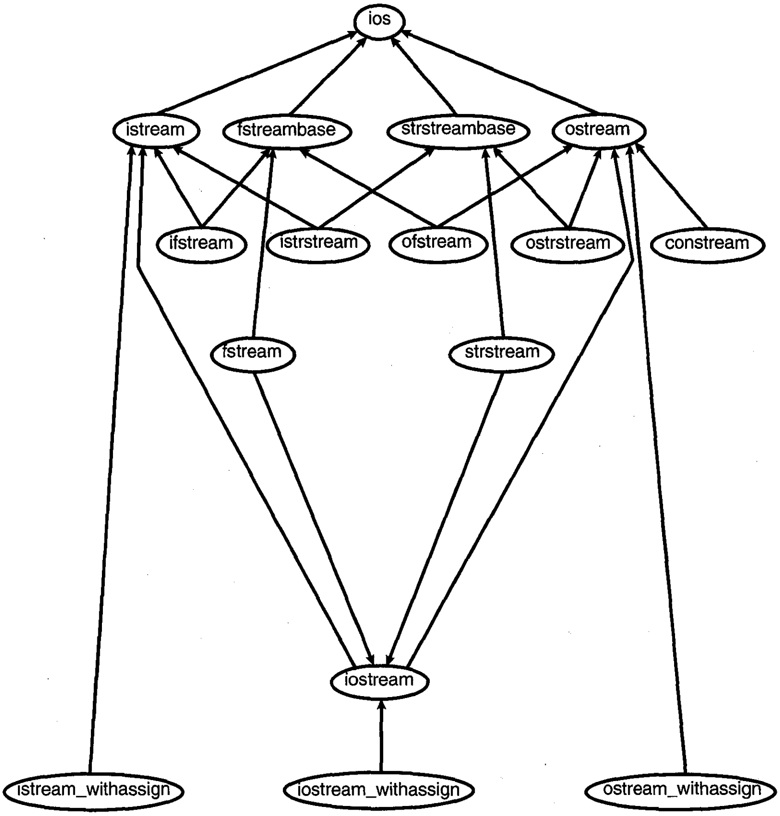
An inheritance diagram for all the *ios* family of classes is found in Figure 6.2. For example, the *ifstream* class is derived from the *istream* and *fstreambase* classes, and *istrstream* is derived from *istream* and *strstreambase*. This diagram is not a simple hierarchy because of the generous use of *multiple inheritance*. With multiple inheritance, a single class can inherit from more than one base class. (The C++ language provides for *virtual inheritance* to avoid multiple declarations.) This means, for example, that all the members (data and functions) of *iostream*, *istream*, *ostream*, *fstreambase*, and *ios* are part of objects of the *fstream* class. All classes in the *ios*-based tree use a *streambuf* (or a *filebuf* or *strstreambuf*, which are special cases of a *streambuf*) as its source and/or sink.

C++ programs start with four predefined open streams, declared as objects of *withassign* classes as follows:

```
extern istream_withassign cin;   // Corresponds to stdin; file descriptor 0.
extern ostream_withassign cout;  // Corresponds to stdout; file descriptor 1.
extern ostream_withassign cerr;  // Corresponds to stderr; file descriptor 2.
extern ostream_withassign clog;  // A buffered cerr; file descriptor 2.
```

Figure 6.2
Class ios and its derived classes

By accepted practice, the arrows point **from** the derived class **to** the base class.



Stream output

Stream output is accomplished with the *insertion* (or *put to)* operator, **<<**. The standard left shift operator, **<<**, is overloaded for output operations. Its left operand is an object of type *ostream*. Its right operand is any type for which stream output has been defined (that is, fundamental types or any types you have overloaded it for). For example,

```
cout << "Hello!\n";
```

writes the string "Hello!" to *cout* (the standard output stream, normally your screen) followed by a new line.

The **<<** operator associates from left to right and returns a reference to the *ostream* object it is invoked for. This allows several insertions to be cascaded as follows:

```
int i = 8;
double d = 2.34;
   cout << "i = " << i << ", d = " << d << "\n";
```

This will write the following to standard output:

```
i = 8, d = 2.34
```

Fundamental types

The fundamental data types directly supported are **char, short, int, long, char*** (treated as a string), **float, double, long double,** and **void***. Integral types are formatted according to the default rules for *printf* (unless you've changed these rules by setting various *ios* flags). For example, the following two output statements give the same result:

```
int i;
long l;
cout << i << " " << l;
printf("%d %ld", i, l);
```

The pointer (**void** *) inserter is used to display pointer addresses:

```
int i;
cout << &i;         // display pointer address in hex
```

Read the description of *ostream* in the *Library Reference* for other output functions.

I/O formatting

Formatting for both input and output is determined by various *format state* flags contained in the class *ios*. The flags are read and set with the *flags, setf,* and *unsetf* member functions.

Output formatting can also be affected by the use of the *fill, width,* and *precision* member functions of **class** *ios*.

The format flags are detailed in the description of **class** *ios* in the *Library Reference.*

Manipulators

A simple way to change some of the format variables is to use a special function-like operator called a *manipulator*. Manipulators take a stream reference as an argument and return a reference to the same stream. You can embed manipulators in a chain of insertions (or extractions) to alter stream states as a side effect without actually performing any insertions (or extractions). For example,

Parameterized manipulators must be called for each stream operation.

```
#include <iostream.h>
#include <iomanip.h>  // Required for parameterized manipulators.

int main(void) {
int i = 6789, j = 1234, k = 10;
```

```
   cout << setw(6) << i << j << i << k << j;
   cout << "\n";
   cout << setw(6) << i << setw(6) << j << setw(6) << k;
   return(0);
   }
```

produces this output:

```
678912346789101234
6789  1234    10
```

setw is a parameterized manipulator declared in iomanip.h. Other parameterized manipulators, *setbase, setfill, setprecision, setiosflags* and *resetiosflags,* work in the same way. To make use of these, your program must include iomanip.h. You can write your own manipulators without parameters:

```
#include <iostream.h>

// Tab and prefix the output with a dollar sign.
ostream& money( ostream& output) {
   return output << "\t$";
   }

int main(void) {
   float owed = 1.35, earned = 23.1;
   cout << money << owed << money << earned;
   return(0);
   }
```

produces the following output:

```
$1.35   $23.1
```

The non-parameterized manipulators *dec, hex,* and *oct* (declared in iostream.h) take no arguments and simply change the conversion base (and leave it changed):

```
int i = 36;
cout << dec << i << " " << hex << i << " " << oct << i << endl;
cout << dec;  // Must reset to use decimal base.
// displays 36 24 44
```

Table 6.1
Stream manipulators

Manipulator	**Action**
dec	Set decimal conversion base format flag.
hex	Set hexadecimal conversion base format flag.
oct	Set octal conversion base format flag.
ws	Extract whitespace characters.
endl	Insert newline and flush stream.

Table 6.1: Stream manipulators (continued)

ends	Insert terminal null in string.
flush	Flush an ostream.
setbase(**int** *n*)	Set conversion base format to base *n* (0, 8, 10, or 16). 0 means the default: decimal on output, ANSI C rules for literal integers on input.
resetiosflags(**long** *f*)	Clear the format bits specified by *f*.
setiosflags(**long** *f*)	Set the format bits specified by *f*.
setfill(**int** *c*)	Set the fill character to *c*.
setprecision(**int** *n*)	Set the floating-point precision to *n*.
setw(**int** *n*)	Set field width to *n*.

The manipulator *endl* inserts a newline character and flushes the stream. You can also flush an *ostream* at any time with

```
ostream << flush;
```

Filling and padding

The fill character and the direction of the padding depend on the setting of the fill character and the left, right, and internal flags.

The default fill character is a space. You can vary this by using the function *fill*:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i;          // display ***123
```

The default direction of padding gives right-alignment (pad on the left). You can vary these defaults (and other format flags) with the functions *setf* and *unsetf*:

```
int i = 56;
   ⋮
cout.width(6);
cout.fill('#');
cout.setf(ios::left,ios::adjustfield);
cout << i;          // display 56####
```

The second argument, *ios*::*adjustfield*, tells *setf* which bits to set. The first argument, *ios*::*left*, tells *setf* what to set those bits to. Alternatively, you can use the manipulators *setfill*, *setiosflags*, and *resetiosflags* to modify the fill character and padding mode. See *ios* data members in the *Library Reference* for a list of masks used by *setf*.

Stream input

Stream input is similar to output but uses the overloaded right shift operator, **>>**, known as the *extraction* (get from) operator or *extractor*. The left operand of **>>** is an object of type **class** *istream*. As with output, the right operand can be of any type for which stream input has been defined.

By default, **>>** skips whitespace (as defined by the *isspace* function in ctype.h), then reads in characters appropriate to the type of the input object. Whitespace skipping is controlled by the **ios::***skipws* flag in the format state's enumeration. The *skipws* flag is normally set to give whitespace skipping. Clearing this flag (with *setf*, for example) turns off whitespace skipping. There is also a special "sink" manipulator, *ws*, that lets you discard whitespace.

Consider the following example:

```
int i;
double d;
cin >> i >> d;
```

When the last line is executed, the program skips any leading whitespace. The integer value (*i*) is then read. Any whitespace following the integer is ignored. Finally, the floating-point value (*d*) is read.

For type **char** (**signed** or **unsigned**), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) character. If you need to read the next character, whether it is whitespace or not, you can use one of the *get* member functions (see the discussion of *istream* in the *Library Reference*).

For type **char*** (treated as a string), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) characters until another whitespace character is found. A final null character is then appended. Care is needed to avoid "overflowing" a string. You can alter the default width of zero (meaning no limit) using *width* as follows:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array;           // Avoids overflow.
```

For all input of fundamental types, if only whitespace is encountered, nothing is stored in the target, and the istream state is set to *fail*. The target will retain its previous value; if it was uninitialized, it remains uninitialized.

I/O of user-defined types

To input or output your own defined types, you must overload the extraction and insertion operators. Here is an example:

```
#include <iostream.h>

struct info {
   char *name;
   double val;
   char *units;
   };

// You can overload << for output as follows:
ostream& operator << (ostream& s, info& m) {
   s << m.name << " " << m.val << " " << m.units;
   return s;
   };

// You can overload >> for input as follows:
istream& operator >> (istream& s, info& m) {
   s >> m.name >> m.val >> m.units;
   return s;
   };

int main(void) {
   info x;
   x.name = new char[15];
   x.units = new char[10];

   cout << "\nInput name, value and units:";
   cin >> x;
   cout << "\nMy input:" << x;
   return(0);
   }
```

Simple file I/O

The class *ofstream* inherits the insertion operations from *ostream,* while *ifstream* inherits the extraction operations from *istream*. The file-stream classes also provide constructors and member functions for creating files and handling file I/O. You must include fstream.h in all programs using these classes.

Consider the following example that copies the file FILE.IN to the file FILE.OUT:

```
#include <fstream.h>

int main(void) {
   char ch;
   ifstream f1("FILE.IN");
   ofstream f2("FILE.OUT");

   if (!f1) cerr << "Cannot open FILE.IN for input";
   if (!f2) cerr << "Cannot open FILE.OUT for output";
   while (f2 && f1.get(ch))
      f2.put(ch);
   return(0);
   }
```

Note that if the *ifstream* or *ofstream* constructors are unable to open the specified files, the appropriate stream error state is set.

The constructors let you declare a file stream without specifying a named file. Later, you can associate the file stream with a particular file:

```
ofstream ofile;         // creates output file stream
   ⋮
ofile.open("payroll");  // ofile connects to file "payroll"
// do some payrolling...

ofile.close();          // close the ofile stream
ofile.open("employee"); // ofile can be reused...
```

By default, files are opened in text mode. This means that on input, carriage-return/linefeed sequences are converted to the '\n' character. On output, the '\n' character is converted to a carriage-return/linefeed sequence. These translations are not done in binary mode. The file-opening mode is set with an optional second parameter to the *open* function or in some file-stream constructors. The file opening-mode constants can be used alone or they can be logically ORed together. See the description of **class** *ios* data members in the *Library Reference.*

String stream processing

The functions defined in strstrea.h support in-memory formatting, similar to *sscanf* and *sprintf,* but much more flexible. All of the *istream* member functions are available for **class** *istrstream* (*i*nput *str*ing *stream*). This is the same for output: *ostrstream* inherits from *ostream.*

Given a text file with the following format:

```
101 191 Cedar Chest
102 1999.99 Livingroom Set
```

Each line can be parsed into three components: an integer ID, a floating-point price, and a description. The output produced is

```
1: 101  191.00  Cedar Chest
2: 102  1999.99 Livingroom Set
```

Here is the program:

```
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <string.h>

int main(int argc, char **argv) {
   int id;
   float amount;
   char description[41];

   if (argc == 1) {
      cout << "\nInput file name required.";
      return (-1);
      }

   ifstream inf(argv[1]);

   if (inf) {
      char inbuf[81];
      int lineno = 0;

      // Want floats to print as fixed point
      cout.setf(ios::fixed, ios::floatfield);

      // Want floats to always have decimal point
      cout.setf(ios::showpoint);

      while (inf.getline(inbuf,81)) {
         // 'ins' is the string stream:
         istrstream ins(inbuf,strlen(inbuf));
         ins >> id >> amount >> ws;
         ins.getline(description,41);  // Linefeed not copied.
         cout << ++lineno << ": "
            << id << '\t'
            << setprecision(2) << amount << '\t'
            << description << "\n";
      }
   }
   return(0);
}
```

Note the use of format flags and manipulators in this example. The calls to *setf* coupled with *setprecision* allow floating-point numbers to be printed in a money format. The manipulator *ws* skips whitespace before the description string is read.

Screen output streams

As with conio.h functions, constreams are not available for GUI applications. The screen area created by *constream* is not bordered or otherwise distinguished from the surrounding screen.

The class *constream*, derived from *ostream* and defined in constrea.h, provides the functionality of conio.h for use with C++ streams. This lets you create output streams that write to specified areas of the screen, in specified colors, and at specific locations.

Console stream manipulators are provided to facilitate formatting of console streams. These manipulators work in the same way as the corresponding function provided by conio.h. For a detailed description of the manipulators' behavior and valid arguments, see the *Library Reference*.

Table 6.2 Console stream manipulators

Manipulator	**conio function**	**Action**
clreol	*clreol*	Clears to end of line in text window.
delline	*delline*	Deletes line in the text window.
highvideo	*highvideo*	Selects high-intensity characters.
insline	*insline*	Inserts a blank line in the text window.
lowvideo	*lowvideo*	Selects low-intensity characters.
normvideo	*normvideo*	Selects normal-intensity characters.
setattr(**int**)	*textattr*	Sets screen attributes.
setbk(**int**)	*textcolor*	Sets new character color.
setclr(**int**)	*textcolor*	Sets the color.
setcrsrtype(**int**)	*_setcursortype*	Selects cursor appearance.
setxy(**int**, **int**)	*gotoxy*	Positions the cursor at the specified position.

Typical use of parameterized manipulators. See the *Library Reference* for a description of **class** *constream.*

```
#include <constrea.h>

int main(void) {
   constream win1;

   win1.window(1, 1, 40, 20); // Initialize the desired space.
   win1.clrscr();             // Clear this rectangle.

   // Use the parameterized manipulator to set screen attributes.
   win1 << setattr((BLUE<<4) | WHITE)
        << "This text is white on blue.";

   // Use this parameterized manipulator to specify output area.
   win1 << setxy(10, 10)
        << "This text is in the middle of the window.";
   return(0);
   }
```

You can create multiple constreams, each writing to its own portion of the screen. Then, you can output to any of them without having to reset the window each time.

```
#include <constrea.h>

int main(void) {
   constream demo1, demo2;

   demo1.window( 1, 2, 40, 10 );
   demo2.window( 1, 12, 40, 20 );

   demo1.clrscr();
   demo2.clrscr();

   demo1 << "Text in first window" << endl;
   demo2 << "Text in second window" << endl;
   demo1 << "Back to the first window" << endl;
   demo2 << "And back to the second window" << endl;
   return(0);
   }
```

CHAPTER 7

Using Borland class libraries

This chapter describes Borland's container class library and persistent streams class library. Reference material for each of these classes can be found in the *Library Reference*.

The container class library

This section describes the Borland International Data Structures (BIDS), also known as the container class library.

Containers are objects that implement common data structures, offering member functions for adding and accessing each container's data elements while hiding the inner details from the user. Containers can hold integers, real numbers, strings, structures, classes, user-defined types, or any C++ object.

Containers and templates

Borland containers are implemented using templates. This means you pass in to the template the *type* of the object you want the container to hold. For example, an array container that holds **float**s would be instantiated like this:

```
TArrayAsVector<float> FloatArray(10);
```

See Chapter 3 for a description of templates.

FloatArray can hold 10 **float**s. The *TArrayAsVector* template class describes the member functions for accessing and maintaining the array. Most containers have *Add* and *Detach* member functions, and the array classes also have the usual **[]** operators for indexing into the array.

Here's another example of an array container that holds a class object:

```
class Myclass {

// class description

};

TArrayAsVector<MyClass> MyClassArray(10);
```

ADTs and FDSs

The container class library can be divided into two categories: Fundamental Data Structures (FDS) and Abstract Data Types (ADT).

Table 7.1
Borland containers and header files

Container	Header file
Borland FDSs	
Binary tree	binimp.h
Hashtable	hashimp.h
Linked list	listimp.h
Double-linked list	dlistimp.h
Vector	vectimp.h
Borland ADTs	
Array	arrays.h
Association	assoc.h
Dequeue	deques.h
Dictionary	dict.h
Queue	queues.h
Set	sets.h
Stack	stacks.h

FDSs are lower-level containers that implement storage constructs. Each FDS has fundamental add and detach member functions. ADTs (for example *TArrayAsVector*) are commonly used data-processing constructs. They are higher-level containers that implement more abstract constructs than lists and vectors, such as stacks and sets. Each ADT has operations (methods) that are particular to that ADT; for example, the stack containers have *Push* and *Pop* member functions.

Each ADT is based on an FDS. For example, *TArrayAsVector* implements an array, using a vector as the underlying FDS. Here is an example of a stack ADT implemented with a linked-list FDS:

```
TStackAsList<int> IntStack(10);
```

Here, a stack ADT is implemented using a vector FDS:

```
TStackAsVector<int> IntStack(10);
```

ADT containers use the storage characteristics of the underlying FDS, and add the specific access methods that make each ADT unique (for example *Push* and *Pop* for stacks).

Choosing an FDS

A vector-based stack is appropriate when the maximum number of elements to be stored on the stack is known in advance, and when speed is critical. A vector allocates space for all its elements when it is created, and the operations on a vector-based stack are simple and fast. The list-based stack is appropriate when there is no reasonable upper bound to the size of the stack, and speed is not as critical.

Direct and indirect containers

Containers can store copies of objects (direct containers) or pointers to objects (indirect containers). *TArrayAsVector<char>* is a direct array that stores a copy of a character. The following container is an indirect array that stores pointers to **float**s:

```
TIArrayAsVector<float> FloatPtrArray(10);
```

The *I* in a template name indicates an indirect container.

The type of object you need to store helps determine whether you need to use a direct or indirect container. A stack of **float**s, for example, would probably use a direct container. A stack of large **struct**s would probably use an indirect container to reduce copying time. This choice, though, is not often easy. Performance tuning requires the comparison of different container implementations. Traditionally this entails drastic recoding. Using containers makes it much easier.

For direct object storage, the contained type must have a valid == operator, a default constructor, and meaningful copy semantics. Indirect containers also need a valid == operator and a default constructor; because indirect containers hold pointers to objects, and pointers always have good copy semantics, indirect containers also always have meaningful copy semantics. This means that indirect containers can contain objects of any type.

Sorted containers

Several containers keep their contents in sorted order. For example,

```
TSArrayAsVector<MyClass> SortMyClassArray(10);
```

instantiates a sorted array of *MyClass* objects, with a vector as the underlying FDS.

Sorted containers (both direct and indirect) require that the type of object passed into the container must have a valid < operator so that the containers' *add* functions can determine the ordering of the elements. These operations are provided for predefined types; for user-defined types, such as classes, you must provide this operator. Here's a simple example of a class with the == and < operators overloaded:

```
class MyClass {

private:
   int a;

// ...

public:


// ...

//overloaded operators necessary for use with a sorted container

   int operator<(const MyClass& mc) const {
      return a < mc.a ? 1 : 0;
   }

   int operator==(const MyClass& mc) const {
      return a == mc.a ? 1 : 0;
   }

};//end MyClass
```

For indirect containers the objects are sorted, not the pointers the container holds.

Memory management

Containers have versions that give you control over memory management. Here is a container that lets you pass in a memory-management object of your choice:

```
TMQueueAsVector<MyClass, MyMemManage> MyQueue(100);
```

TMQueueAsVector takes two type parameters. One is the type of object that the queue will hold (*MyClass*), the other is the name of a memory-management class (*MyMemManag*) that you want to use. The *M* in a template name means that you must specify a memory manager to implement that container. Container template names without the *M* use the standard memory allocator *TStandardAllocator* found in alloctr.h. The following two container declarations are equivalent:

```
TMQueueAsVector<MyClass, TStandardAllocator> MyQueue(100);

TQueueAsVector<MyClass> MyQueue(100);
```

Both use *TStandardAllocator* to manage memory. *TStandardAllocator* provides operators **new**, **new[]**, **delete**, and **delete[]**, which call their global counterparts. No specialized behavior is provided.

User-supplied memory management must provide a class-specific **new** operator, a placement **new** operator that takes a **void** * argument as its second parameter, and a **delete** operator. Use the allocators in alloctr.h as an example for building your own.

Container naming conventions

The characteristics of each container class are encoded in the container name. For example, *TMIArrayAsVector* is a "managed, indirect array implemented as a vector." That is, this template takes a memory management scheme as a parameter, is an indirect container, and is implemented using a vector. *TDequeAsDoubleList* is a direct container that uses the system memory-management scheme and which is implemented as a double-linked list. Table 7.2 summarizes these abbreviations.

Table 7.2
Container name abbreviations

Abbreviation	Description
T	Borland class library prefix
M	User supplied memory-management container
I	Indirect container
C	Counted container
S	Sorted container

ADT/FDS combinations in the library

The BIDS libraries do not contain all possible combinations of ADT/FDS combinations. Table 7.3 lists the ADT/FDS combinations supplied.

Table 7.3
ADT/FDS combinations

	ADT							
FDS	**Stack**	**Queue**	**Dequeue**	**Bag**	**Set**	**Sorted Array**	**Array**	**Dictionary**
Vector	x	x	x	x	x	x	x	
List	x							
DoubleList			x	x				
Hashtable								x
Binary tree								

You can use the template classes to develop your own ADT/FDS implementations.

Container iterators

Each container class has a corresponding container iterator class, which are classes dedicated to iterating over a particular kind of container. For example, *TArrayAsVector* has a corresponding iterator called *TArrayAsVectorIterator* that is responsible for iterating over all the items in the array.

Container iterators implement the **++** pre- and post-increment operators for that container. They also implement the *Current* member function (which returns the current object) and the *Restart* member function (which restarts iteration).

Here is an iterator example:

```
#include <iostream.h>
#include <classlib\arrays.h>

typedef TArrayAsVector<float> floatArray;
typedef TArrayAsVectorIterator<float> floatArrayIterator;

int main(void){

  //create an array of integers
  floatArray FloatArray(10);

  int count = 0;

  //add items to the array using Add member function
  //for (int i=0; i <= FloatArray.ArraySize(); i++)
  while (count <= FloatArray.ArraySize())
    FloatArray.Add(float(count++));


  //create an iterator - the constructor takes the array name
  //as a parameter
  floatArrayIterator nextFloat(FloatArray);

  cout << "FloatArray contents:" << endl;

  while (nextFloat !=0) {
    cout << FloatArray[count++] << "  ";
    cout << endl;
    ++nextFloat;
  }

}
```

Object ownership

Indirect containers inherit the *OwnsElements* member function from *TShouldDelete* (shddel.h). *OwnsElements* lets you indicate whether the default action of the container is to delete objects when using member functions *Detach* and *Flush*. *Detach* and *Flush* each take a parameter that indicates whether or not they should delete the object, use the default.

Using containers

Using templatized containers lets you develop a stack-based application (for example, using vectors as the underlying structure) that you can change to a linked-list implementation without major recoding. Often it involves only a change to a **typedef**.

For example:

```
//Create a stack of integers, load the stack, and output contents
#include <classlib\stacks.h>
#include <iostream.h>
```

```
//The recommended way of declaring container types
typedef TStackAsVector<int> IntStack;

int main()
{
   IntStack intStack;
   for ( int i = 0; i < 10; i++ )
       intStack.Push( i );
   for ( i = 0; i < 10; i++ )
       cout << intStack.Pop() << "  ";
   cout << endl;
   return(0);
}
```

Output

```
9  8  7  6  5  4  3  2  1  0
```

This implements a stack of **int**s using a vector as the underlying FDS. If you later determine that a list would be a more suitable implementation for the stack, you can replace the **typedef** with the following:

```
typedef TStackAsList<int> IntStack;
```

After recompiling, the stack implementation is changed from a vector to a linked list. With only the **typedef** changed, the code continues to work properly.

When changing to an indirect container, a few more changes are required:

```
//Create a stack of integer pointers, load the stack, and output //contents
#include <classlib\stacks.h>
#include <iostream.h>

//Changed typedef as usual
typedef TIStackAsVector<int> IntStack;

int main()
{
  IntStack intStack;

  for ( int i = 0; i < 10; i++ )
     //Indirect Push takes pointer arg
     intStack.Push( &i );

  for ( i = 0; i < 10; i++ )
    //Indirect Pop returns pointer - need to dereference
    cout << *intStack.Pop() << "  ";

  cout << endl;
  return(0);
}
```

Output

```
9  8  7  6  5  4  3  2  1  0
```

A sorted array example

The following example uses a sorted, indirect array containing strings.

If you used *TIArrayAsVector<String>*, the elements would appear in the order they were added to the array.

```
#include <iostream.h>
#include <strstrea.h>
#include <classlib\arrays.h>
#include <cstring.h>

int main()
{
   typedef TISArrayAsVector<string> lArray;
   lArray a(2);
   for (int i = a.ArraySize(); i; i--)
   {
      ostrstream os;
      os << "string " << (10 - i) << ends;
      a.Add( (new string(os.str())));
   }
   cout << "array elements:\n";

//In the sorted array container, the index of a particular array
//element depends on its value, not on the order it was entered

   for (i = 0; i < a.ArraySize(); ++i)
      cout<< *a[i] << endl;

   return(0);
}
```

Output

```
array elements:
string 7
string 8
string 9
```

A dequeue example

The following example illustrates an indirect dequeue, implemented as a double-linked list.

```
#include <iostream.h>
#include <strstrea.h>
#include <classlib\deques.h>
#include <cstring.h>
```

Pointers to string objects in the dequeue container must be dereferenced when extracting from the dequeue.

```
typedef TIDequeAsDoubleList<string> lDeque;

int main()
{
   lDeque d;
   for (int i = 1; i < 5; i++)
   {
      ostrstream os;
      os << "string " << i << ends;
      // use alternating left, right insertions
      if(i&1)
        d.PutLeft(new string(os.str()));
      else
        d.PutRight(new string(os.str()));
   }
   cout << "Dequeue Contents:" << endl;
   while (!d.IsEmpty())
      //Must dereference when using indirect container
      cout << *d.GetLeft() << endl;

   return(0);
}
```

Output

```
Dequeue Contents:
string 3
string 1
string 2
string 4
```

Container directories

The libraries for the template-based container classes are distinguished by the prefix BIDS: BIDS*x*.LIB, where *x* represents the memory model, and BIDSDB*x*.LIB. for the diagnostic version.

Container class support includes directories containing:

To use the BIDS libraries you must explicitly add the appropriate BIDS[DB]x.LIB library to your project or makefile.

- Header files
- Libraries
- Source files
- Examples

The following sections describe the directories containing each.

The LIBS and BIN directories

The following table lists the container libraries:

File name	Description
BIDSF.LIB	32-bit (flat model)
BIDSDF.LIB	32-bit (flat model) diagnostic version
BIDS40F.DLL	32-bit (flat model) DLL
BIDS40DF.DLL	32-bit (flat model) DLL diagnostic version
BIDSFI.LIB	32-bit (flat model) import library
BIDSDFI.LIB	32-bit (flat model) import library diagnostic version
BIDSS.LIB	16-bit small model
BIDSDBS.LIB	16-bit small model diagnostic version
BIDSM.LIB	16-bit medium model
BIDSDBM.LIB	16-bit medium model diagnostic version
BIDSC.LIB	16-bit compact model
BIDSDBC.LIB	16-bit compact model diagnostic version
BIDSL.LIB	16-bit large model
BIDSDBL.LIB	16-bit large model diagnostic version
BIDS40.DLL	16-bit DLL
BIDS40D.DLL	16-bit DLL diagnostic version
BIDSI.LIB	16-bit import library
BIDSDI.LIB	16-bit import library diagnostic version

The INCLUDE directory

The INCLUDE\CLASSLIB directory contains the header files necessary to compile a program that uses container classes. For each ADT or FDS there is a corresponding header file in this directory. Make sure the INCLUDE directory is on your include path, and then reference header files with an explicit CLASSLIB. For example:

```
#include <classlib\stacks.h>
```

The SOURCE directory

The SOURCE\CLASSLIB directory contains the source files that implement many of the member functions of the classes in the library. You will need these source files if you want to build a library. The supplied MAKEFILE builds a class library of the specified memory model and places that library in the LIB directory.

The EXAMPLES directory

The EXAMPLES\CLASSLIB\BIDS directory has several example programs that use container classes. Here is a list of the example programs and the classes they use:

- STRNGMAX: A string collating example.
- REVERSE: An intermediate example using *TStack* (an alias for *TStackAsVector*) and *String*. This example lets the user input strings, then outputs the strings in reverse order.
- LOOKUP: An intermediate example using *TDictionaryAsHashTable* and *TDDAssociation*.
- QUEUETST: An intermediate example using *TQueue* (an alias for *TQueueAsVector*) and a nonhierarchical class, *TTime*.
- DIRECTRY: An advanced example illustrating derived user classes with *TISArrayAsVector*, and using source files FILEDATA.CPP and TESTDIR.CPP.

Debugging containers

Borland provides macros for debugging classes. Chapter 9 of the *Library Reference* describes how to use these class diagnostic macros.

The persistent streams class library

This section describes what's new with Borland's object streaming support, then explains how to make your objects streamable.

Objects that you create when an application runs—windows, dialog boxes, collections, and so on—are temporary. They are constructed, used, and destroyed as the application proceeds. Objects can appear and disappear as they enter and leave their scope, or when the program terminates. By making your objects streamable you save these objects, either in memory or file streams, so that they *persist* beyond their normal lifespan.

See Chapter 6 of the *Library Reference* for more on persistent streams.

There are many applications for persistent objects. When saved in shared memory they can provide interprocess communication. They can be transmitted via modems to other systems. And, most significantly, objects can be saved permanently on disk using file streams. They can then be read back and restored by the same application, by other instances of the same application, or by other applications. Efficient, consistent, and safe streamability is available to all objects.

Building your own streamable classes is straightforward and incurs little overhead. To make your class streamable you need to add specific data members, member functions, and operators. You also must derive your class, either directly or indirectly, from the *TStreamableBase* class. Any derived class is also streamable.

To simplify creating streamable objects, the persistent streams library contains macros that add all the routines necessary to make your classes streamable. The two most important are

- DECLARE_STREAMABLE
- IMPLEMENT_STREAMABLE

These macros add the boilerplate code necessary to make your objects streamable. In most cases you can make your objects streamable by adding these two macros at appropriate places in your code, as explained later.

What's new with streaming

Object streaming has been significantly changed for Borland C++ 4.0 to make it easier to use and more powerful. These changes are compatible with existing ObjectWindows and Turbo Vision code.

The new streaming code is easier to use because it provides macros that relieve the programmer of the burden of remembering most of the details needed to create a streamable class. Its other new features include support for multiple inheritance, class versioning, and better system isolation. In addition, the streaming code has been reorganized to make it easier to write libraries that won't force streaming code to be linked in if it isn't used.

Streaming has been moved from the ObjectWindows library to the class library. This makes streaming more easily usable in applications that don't use ObjectWindows.

There have been several additions to the streaming capabilities. These changes are intended to be backward compatible, so if you compile a working application with the new streaming code, your application should be able to read streams that were written with the old code. There is no provision for writing the old stream format, however. We assume that you'll like the new features so much that you won't want to be without them.

The following sections describe the changes and new capabilities of streaming. Each of these changes is made for you when you use the DECLARE_STREAMABLE and IMPLEMENT_STREAMABLE macros.

Object versioning

Objects in streams now have a version number associated with them. An object version number is a 32-bit value that should not be 0. Whenever an object is written to a stream, its version number will also be written. With versioning you can recognize if there's an older version of the stream you're reading in, so you can interpret the stream appropriately.

Reading and writing base classes

In your current code, you might be reading and writing base classes directly, as shown here:

```
void Derived::write( opstream& out )
{
   Base::write( out );
// ...
}

void *Derived::read( ipstream& in )
{
   Base::read( in );
// ...
}
```

This method will continue to work, but it won't write out any version numbers for the base class. To take full advantage of versioning, you should change these calls to use the new template functions that understand about versions:

```
void Derived::Write( opstream& out )
{
   WriteBaseObject( (Base *)this, out );
// ...
}

void *Derived::Read( ipstream& in, uint32 ver )
{
   ReadBaseObject( (Base *)this, in );
// ...
}
```

➡ The cast to a pointer to the base class is essential. If you leave it out your program may crash.

Reading and writing integers

Old streams wrote **int** and **unsigned** data types as 2-byte values. To move easily to 32-bit platforms, the new streams write **int** and **unsigned** values as 4-byte values. The new streams can read old streams, and will handle the 2-byte values correctly.

The old streams provide two member functions for reading and writing integer values:

```
void writeWord(unsigned);

unsigned readWord();
```

These have been changed in the new streams:

```
void writeWord(uint32);
```

```
uint32 readWord();
```

Existing code that uses these functions will continue to work correctly if it is recompiled and relinked, although calls to *readWord* will generate warnings about a loss of precision when the return value is assigned to an **int** or **unsigned** in a 16-bit application. But in new code all of these functions should be avoided. In general, you probably know the true size of the data being written, so the streaming library now provides separate functions for each data size:

Use of these four functions is preferred.

```
void writeWord16(uint16);

void writeWord32(uint32);

uint16 readWord16(uint16);

uint32 writeWord32(uint32);
```

Multiple inheritance and virtual base support

The streaming code now provides four function templates that support virtual base classes and multiple inheritance. The following sections describe these functions.

The ReadVirtualBase and WriteVirtualBase function templates

Any class that has a direct virtual base should use the new *ReadVirtualBase* and *WriteVirtualBase* function templates:

```
void Derived::Write( opstream& out )
{
   WriteVirtualBase( (VirtualBase *)this, out );
// ...
}

void *Derived::Read( ipstream& in, uint32 ver )
{
   ReadVirtualBase( (VirtualBase *)this, in );
// ...
}
```

A class derived from a class with virtual bases does not need to do anything special to deal with those virtual bases. Each class is responsible only for its direct bases.

The ReadBaseObject and WriteBaseObject function templates

Object streams now support multiple inheritance. To read and write multiple bases, use the new *WriteBaseObject* and *ReadBaseObject* function templates for each base:

```
void Derived::Write( opstream& out )
{
   WriteBaseObject( (Base1 *)this, out );
   WriteBaseObject( (Base2 *)this, out ):
// ...
}

void *Derived::Read( ipstream& in, uint32 ver )
{
   ReadBaseObject( (Base1 *)this, in );
   ReadBaseObject( (Base2 *)this, in );
// ...
}
```

Creating streamable objects

The easiest way to make a class streamable is by using the macros supplied in the persistent streams library. The following steps will work for most classes:

1. Make *TStreamableBase* a virtual base of your class, either directly or indirectly.
2. Add the DECLARE_STREAMABLE macro to your class definition.
3. Add the IMPLEMENT_STREAMABLE macro to one of your source files. Adding the IMPLEMENT_CASTABLE macro is also recommended.
4. Write the *Read* and *Write* member function definitions in one of your source files.

The following sections provide details about defining and implementing streamable classes.

Defining streamable classes

To define a streamable class you need to

- Include objstrm.h
- Base your class on the *TStreamableBase* class

- Include macro DECLARE_STREAMABLE into your class definition. For example,

```
#include <objstrm.h>

class Sample : public TStreamableBase
{
public:
   // member functions, etc.
private:
   int i;
DECLARE_STREAMABLE(IMPEXPMACRO, Sample, 1 );
};
```

Header file objstrm.h provides the classes, templates, and macros that are needed to define a streamable class.

Every streamable class must inherit, directly or indirectly, from the class *TStreamableBase*. In this example, the class *Sample* inherits directly from *TStreamableBase*. A class derived from *Sample* would not need to explicitly inherit from *TStreamableBase* because *Sample* already does. If you are using multiple inheritance, you should make *TStreamableBase* a virtual base instead of a nonvirtual base as shown here. This will make your classes slightly larger, but won't have any other adverse affect on them.

In most cases the DECLARE_STREAMABLE macro is all you need to use when you're defining a streamable class. This macro takes three parameters. The first parameter is used when compiling DLLs. This parameter takes a macro that is meant to expand to either **_ _import**, **_ _export**, or nothing, depending on how the class is to be used in the DLL. See Chapters 6 and 9 of the *Library Reference* for further explanation. The second parameter is the name of the class that you're defining, and the third is the version number of that class. The streaming code doesn't pay any attention to the version number, so it can be anything that has some significance to you. See the discussion of the nested class *Streamer* for details.

DECLARE_STREAMABLE adds a constructor to your class that takes a parameter of type *StreamableInit*. This is for use by the streaming code; you won't need to use it directly. DECLARE_STREAMABLE also creates two inserters and two extractors for your class so that you can write objects to and read them from persistent streams. For the class *Sample* (shown earlier in this section), these functions have the following prototypes:

```
opstream& operator << ( opstream&, const Sample& );
opstream& operator << ( opstream&, const Sample* );
ipstream& operator >> ( ipstream&, Sample& );
ipstream& operator >> ( ipstream&, Sample*& );
```

The first inserter writes out objects of type *Sample*. The second inserter writes out objects pointed to by a pointer to *Sample*. This inserter gives you the full power of object streaming, because it understands about polymorphism. That is, it will correctly write objects of types derived from *Sample*, and when those objects are read back in using the pointer extractor (the last extractor) they will be read in as their actual types. The extractors are the inverse of the inserters.

Finally, DECLARE_STREAMABLE creates a nested class named *Streamer*, based on the *TStreamer* class, which defines the core of the streaming code.

Implementing streamable classes

Most of the members added to your class by the DECLARE_STREAMABLE macro are inline functions. There are a few, however, that aren't inline; these must be implemented outside of the class. Once again, there are macros to handle these definitions.

The IMPLEMENT_CASTABLE macro provides a rudimentary typesafe downcast mechanism. If you are building with Borland C++ 4.0 you don't need to use this because Borland C++ 4.0 supports run-time type information. However, if you need to build your code with a compiler that does not support run-time type information, you will need to use the IMPLEMENT_CASTABLE macro to provide the support that object streaming requires. Although it isn't necessary to use IMPLEMENT_CASTABLE when using Borland C++ 4.0, you ought to do so anyway if you're concerned about being able to compile your code with another compiler.

IMPLEMENT_CASTABLE has several variants:

```
IMPLEMENT_CASTABLE( cls )
IMPLEMENT_CASTABLE1( cls, base1 )
IMPLEMENT_CASTABLE2( cls, base1, base2 )
IMPLEMENT_CASTABLE3( cls, base1, base2, base3 )
IMPLEMENT_CASTABLE4( cls, base1, base2, base3, base4 )
IMPLEMENT_CASTABLE5( cls, base1, base2, base3, base4, base5)
```

At some point in your source code you should invoke this macro with the name of your streamable class as its first parameter and the name of all its streamable base classes other than *TStreamableBase* as the succeeding parameters. For example,

```
class Base1 : public virtual TStreamableBase
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, Base1, 1 );
};
IMPLEMENT_CASTABLE( Base1 );          // no streamable bases
```

```
class Base2 : public virtual TStreamableBase
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, Base2, 1 );
};
IMPLEMENT_CASTABLE( Base1 );          // no streamable bases

class Derived : public Base1, public virtual Base2
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, Derived, 1 );
};
IMPLEMENT_CASTABLE2( Derived, Base1, Base2 ); //two streamable bases

class MostDerived : public Derived
{
DECLARE_STREAMABLE( IMPEXPMACRO, MostDerived, 1 );
};
IMPLEMENT_CASTABLE1( MostDerived, Derived ); //one streamable base
```

The class *Derived* uses IMPLEMENT_CASTABLE2 because it has two streamable base classes.

In addition to the IMPLEMENT_CASTABLE macros, you should invoke the appropriate IMPLEMENT_STREAMABLE macro somewhere in your code. The IMPLEMENT_STREAMABLE macro looks like the IMPLEMENT_CASTABLE macros:

```
IMPLEMENT_STREAMABLE( cls )
IMPLEMENT_STREAMABLE1( cls, base1 )
IMPLEMENT_STREAMABLE2( cls, base1, base2 )
IMPLEMENT_STREAMABLE3( cls, base1, base2, base3 )
IMPLEMENT_STREAMABLE4( cls, base1, base2, base3, base4 )
IMPLEMENT_STREAMABLE5( cls, base1, base2, base3, base4, base5 )
```

The IMPLEMENT_STREAMABLE macros have one important difference from the IMPLEMENT_CASTABLE macros: when using the IMPLEMENT_STREAMABLE macros you must list all the streamable base classes of your class in the parameter list, and you must list all virtual base classes that are streamable. This is because the IMPLEMENT_STREAMABLE macros define the special constructor that the object streaming code uses; that constructor must call the corresponding constructor for all of its direct base classes and all of its virtual bases. For example,

```
class Base1 : public virtual TStreamableBase
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, Base1, 1 );
};
IMPLEMENT_CASTABLE( Base1 );           // no streamable bases
IMPLEMENT_STREAMABLE( Base1 ); // no streamable bases

class Base2 : public virtual TStreamableBase
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, Base2, 1 );
};
IMPLEMENT_CASTABLE( Base1 );           // no streamable bases
IMPLEMENT_STREAMABLE( Base1 ); // no streamable bases

class Derived : public Base1, public virtual Base2
{
// ..
DECLARE_STREAMABLE( IMPEXPMACRO, Derived, 1 );
};
IMPLEMENT_CASTABLE2( Derived, Base1, Base2 );
IMPLEMENT_STREAMABLE2( Derived, Base1, Base2 );

class MostDerived : public Derived
{
// ...
DECLARE_STREAMABLE( IMPEXPMACRO, MostDerived, 1 );
};
IMPLEMENT_CASTABLE1( MostDerived, Derived );
IMPLEMENT_STREAMABLE2( MostDerived, Derived, Base2 );
```

The nested class Streamer

The nested class *Streamer* is the core of the streaming code for your objects. The DECLARE_STREAMABLE macro creates *Streamer* inside your class. It is a protected member, so classes derived from your class can access it. *Streamer* inherits from *TNewStreamer,* which is internal to the object streaming system. It inherits the following two pure virtual functions:

```
virtual void Write( opstream& ) const = 0;
virtual void *Read( ipstream&, uint32 ) const = 0;
```

Streamer overrides these two functions, but does not provide definitions for them. You must write these two functions: *Write* should write any data that needs to be read back in to reconstruct the object, and *Read* should read that data. *Streamer::GetObject* returns a pointer to the object being streamed. For example,

```
class Demo : public TStreamableBase
{
   int i;
   int j;
public:
   Demo( int ii, int jj ) : i(ii), j(jj) {}
DECLARE_STREAMABLE( IMPEXPMACRO, Demo, 1 );
};
IMPLEMENT_CASTABLE( Demo );
IMPLEMENT_STREAMABLE( Demo );

void *Demo::Streamer::Read( ipstream& in, uint32 ) const
{
   in >> GetObject()->i >> GetObject()->j;
   return GetObject();
}

void Demo::Streamer::Write( opstream& out ) const
{
   out << GetObject()->i << GetObject()->j;
}
```

Writing the Read and Write functions

It is usually easiest to implement the *Read* function before implementing the *Write* function. To implement *Read* you need to

- Know what data you need in order to reconstruct the new streamable object
- Devise a sensible way of reading that data into the new streamable object.

Then implement *Write* to work in parallel with *Read* so that it sets up the data that *Read* will later read. The streaming classes provide several operators to make this easier. For example, *opstream* provides inserters for all the built-in types, just as *ostream* does. So all you need to do to write out any of the built-in types is to insert them into the stream.

You also need to write out base classes. In the old ObjectWindows and Turbo Vision streaming, this was done by calling the base's *Read* and *Write* functions directly. This doesn't work with code that uses the new streams, because of the way class versioning is handled.

The streaming library provides template functions to use when reading and writing base classes. *ReadVirtualBase* and *WriteVirtualBase* are used for virtual base classes, and *ReadBaseObject* and *WriteBaseObject* are used for nonvirtual bases. Just like IMPLEMENT_CASTABLE, you only need to deal with direct bases. Virtual bases of your base classes will be handled by the base class, as shown in this example:

```
class Base1 : public virtual TStreamableBase
{
int i;
DECLARE_STREAMABLE( IMPEXPMACRO, Base1, 1 );
};
IMPLEMENT_CASTABLE( Base1 );           // no streamable bases
IMPLEMENT_STREAMABLE( Base1 ); // no streamable bases
void Base1::Streamer::Write( opstream& out ) const
{
   out << GetObject()->i;
}

class Base2 : public virtual TStreamableBase
{
int j;
DECLARE_STREAMABLE( IMPEXPMACRO, Base2, 1 );
};
IMPLEMENT_CASTABLE( Base1 );           // no streamable bases
IMPLEMENT_STREAMABLE( Base1 ); // no streamable bases
void Base2::Streamer::Write( opstream& out ) const
{
   out << GetObject()->j;
}

class Derived : public Base1, public virtual Base2
{
int k;
DECLARE_STREAMABLE( IMPEXPMACRO, Derived, 1 );
};
IMPLEMENT_CASTABLE2( Derived, Base1, Base2 );
IMPLEMENT_STREAMABLE2( Derived, Base1, Base2 );
void Derived::Streamer::Write( opstream& out ) const
{
   WriteBaseObject( (Base1 *)this, out );
   WriteVirtualBase( (Base2 *)this, out );
   out << GetObject()->k;
}

class MostDerived : public Derived
{
int m;
DECLARE_STREAMABLE( IMPEXPMACRO, MostDerived, 1 );
};
IMPLEMENT_CASTABLE1( MostDerived, Derived );
IMPLEMENT_STREAMABLE2( MostDerived, Derived, Base2 );
void MostDerived::Streamer::Write( opstream& out ) const
{
   WriteBaseObject( (Derived *)this, out );
   out << GetObject()->m;
}
```

When you're writing out a base class, don't forget to cast the **this** pointer. Without the cast, the template function will think it's writing out your class and not the base class. The result will be that it calls your *Write* or *Read* function rather than the base's. This results in a lengthy series of recursive calls, which will eventually crash.

Object versioning

You can assign version numbers to different implementations of the same class as you change them in the course of maintenance. This doesn't mean that you can use different versions of the same class in the same program, but it lets you write your streaming code in such a way that a program using the newer version of a class can read a stream that contains the data for an older version of a class. For example:

```
class Sample : public TStreamableBase
{
int i;
DECLARE_STREAMABLE( IMPEXPMACRO, Sample, 1 );
};
IMPLEMENT_CASTABLE( Sample );
IMPLEMENT_STREAMABLE( Sample );
void Sample::Streamer::Write( opstream& out ) const
{
   out << GetObject()->i;
}
void *Sample::Streamer::Read( ipstream& in, uint32 ) const
{
   in >> GetObject()->i;
   return GetObject();
}
```

Suppose you've written out several objects of this type into a file and you discover that you need to change the class definition. You'd do it something like this:

```
class Sample : public TStreamableBase
{
int i;
int j;         // new data member
DECLARE_STREAMABLE( IMPEXPMACRO, Sample, 2 );// new version number
};
IMPLEMENT_CASTABLE( Sample );
IMPLEMENT_STREAMABLE( Sample );
void Sample::Streamer::Write( opstream& out ) const
{
   out << GetObject()->i;
   out << GetObject()->j;
}
```

```
void *Sample::Streamer::Read( ipstream& in, uint32 ver ) const
{
   in >> GetObject()->i;
   if( ver > 1 )
            in >> GetObject()->j;
   else
            GetObject()->j = 0;
   return GetObject();
}
```

Streams written with the old version of *Sample* will have a version number of 1 for all objects of type *Sample*. Streams written with the new version will have a version number of 2 for all objects of type *Sample*. The code in *Read* checks that version number to determine what data is present in the stream.

The streaming library used in the previous versions of ObjectWindows and Turbo Vision doesn't support object versioning. If you use the new library to read files created with that library, your *Read* function will be passed a version number of 0. Other than that, the version number has no significance to the streaming library, and you can use it however you want.

CHAPTER 8

Windows programming

See page 253 for 32-bit Windows (Win32) information.

This chapter describes a variety of 16-bit and 32-bit Windows programming topics, including

- Resource script files
- Module definition files
- Import libraries
- Project files and the project manager
- The Borland heap manager
- 32-bit Windows programming

The intricacies of designing and developing Windows applications go beyond the scope of this chapter.

In addition to compiling source code and linking .OBJ files, a Windows programmer must compile resource script files and bind resources to an executable. A Windows programmer must also know about dynamic linking, dynamic link libraries (DLLs), and import libraries. Also, if you're using the Borland C++ IDE, it's helpful to know how to use the Borland project manager, which uses project files to automate and manage application building. Figure 8.1 illustrates the process of building a Windows application.

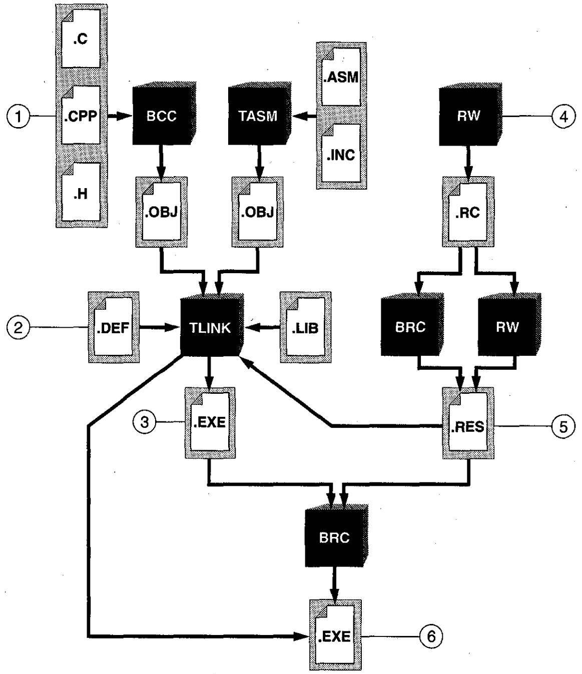


Figure 8.1
Compiling and linking a Windows program

These are the steps to compiling and linking a Windows program:

1. Source code is compiled or assembled, producing .OBJ files.
2. Module definition files (.DEF) tell the linker what kind of executable you want to produce.
3. Resource Workshop (or some other resource editor) creates resources, like icons or bitmaps. A resource file (.RC) is produced. See the *User's Guide* documentation for more information on using Resource Workshop, and Chapter 10 in the *User's Guide* for more information on using resource tools.
4. The .RC file is compiled by a resource compiler or Resource Workshop, and a binary .RES file is output.
5. Linking produces an .EXE file with bound resources.

Resource script files

Windows applications typically use *resources*. Resources are icons, menus, dialog boxes, fonts, cursors, bitmaps, or other user-defined resources.

Resources are defined in a file called a resource script file, also known as a resource file. These files have the file-name extension .RC.

See Chapter 10 in the *User's Guide* for a complete discussion of BRC.

To use resources, you must use the Borland Resource Compiler (BRC or BRC32) to compile your .RC file into a binary format. Resource compilation creates a .RES file. TLINK or TLINK32 then binds the .RES file to the .EXE file output by the linker. This process also marks the .EXE file as a Windows executable.

Module definition files

Module definition files are described in detail in Chapter 9 of the *User's Guide.*

A module definition (.DEF) file provides information to the linker about the contents and system requirements of a Windows application. This information includes heap and stack size, and code and data characteristics. .DEF files also list functions that are to be made available for other modules (export functions), and functions that are needed from other modules (import functions). Because Borland linkers have other ways of finding out the information contained in a module definition file, module definition files are not always required for Borland C++'s linker to create a Windows application.

Here's the module definition file for the WHELLO example:

```
NAME           WHELLO
DESCRIPTION    'C++ Windows Hello World'
EXETYPE        WINDOWS
CODE           PRELOAD MOVEABLE
DATA           PRELOAD MOVEABLE MULTIPLE
HEAPSIZE       1024
STACKSIZE      5120
```

Let's take this file apart, statement by statement:

- NAME specifies a name for a program. If you want to build a DLL instead of a program, you would use the LIBRARY statement. Every module definition file should have either a NAME statement or a LIBRARY statement, but never both. The name specified must be the same name as the executable file. WINDOWAPI identifies this program as a Windows executable.
- DESCRIPTION lets you specify a string that describes your application or library.
- EXETYPE marks the executable as a Windows executable. This is necessary for all Windows executables.
- CODE describes attributes of the executable's code segment. The PRELOAD option instructs the loader to load this portion of the image

when the application is loaded into memory. The MOVEABLE option means Windows can move the code around in memory.

- DATA defines the default attributes of data segments. The MULTIPLE option ensures that each instance of the application has its own data segment.
- HEAPSIZE specifies the size of the application's local heap.
- STACKSIZE specifies the size of the application's local stack. You can't use the STACKSIZE statement to create a stack for a DLL.

Two important statements not used in this .DEF file are the EXPORTS and IMPORTS statements.

The EXPORTS statement lists functions in a program or DLL that will be called by other applications or by Windows. These functions are known as export functions, callbacks, or callback functions. Exported functions are identified by the linker and entered into an export table.

The **_export** keyword should immediately precede the function name.

To help you avoid having to create and maintain long EXPORTS sections in your module definition files, Borland C++ provides the **_export** keyword. Functions flagged with **_export** will be identified by the linker and entered into the export table for the module. This is why the WHELLO example has no EXPORT statement in its module definition file.

The WHELLO application doesn't have an IMPORTS statement either because the only functions it calls from other modules are those from the Windows Application Program Interface (API); those functions are imported via the automatic inclusion of the IMPORT.LIB, IMPORT32.LIB, or IMPRTW32.LIB import libraries. When an application needs to call other external functions, these functions must be listed in the IMPORTS statement or included via an import library.

Import libraries

When you use DLLs, you must give the linker definitions of the functions you want to import from DLLs. This information temporarily satisfies the external references to the functions called by the compiled code, and tells the Windows loader where to find the functions at run time.

There are two ways to tell the linker about import functions:

- You can add an IMPORTS section to the module definition file and list every DLL function that the module will use.
- You can include an import library for the DLLs when you link the module.

See Chapter 11 in the *User's Guide* for an explanation of how to use import library tools.

An import library contains import definitions for some or all of the exported functions for one or more DLLs. A utility called IMPLIB creates import libraries for DLLs. IMPLIB creates import libraries directly from DLLs or from a DLL's module definition files, or from a combination of the two.

Import libraries can be substituted for all or part of the IMPORTS section of a module definition file.

WinMain

➡ You must supply the *WinMain* function as the main entry point for Windows applications. Some applications, the Borland ObjectWindows library for example, encapsulate the *WinMain* entry point.

The following parameters are passed to *WinMain*:

WINAPI, HINSTANCE and LPSTR are defined in windows.h.

```
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine, int nCmdShow)
```

- *hInstance* is the instance handle of the application. Each instance of a Windows application has a unique instance handle that's used as an argument to several Windows functions and can be used to distinguish between multiple instances of a given application.
- *hPrevInstance* is the handle of the previous instance of this application. *hPrevInstance* is NULL if this is the first instance, and is always 0 under Win32.
- *lpCmdLine* is a pointer (a far pointer in 16-bit Windows) to a null-terminated command-line string. This value can be specified when invoking the application from the program manager or from a call to **WinExec**.
- *nCmdShow* is an integer that specifies how to display the application's window (for example iconized).

The return value from *WinMain* is not currently used by Windows. However, it can be useful during debugging because Turbo Debugger can display this value when your program terminates.

Prologs and epilogs

When you compile a module for Windows, the compiler needs to know what kind of prolog and epilog to create for each of a module's functions.

IDE settings and command-line compiler options control the creation of the prolog and epilog. The prolog and epilog perform several functions, including ensuring that the correct data segment is active during callback functions, and marking stack frames for the Windows stack-crawling mechanism.

The prolog and epilog code is automatically generated by the compiler, though various compiler options or IDE settings dictate which sets of instructions are contained in the generated code.

The following sections describe the **_export** and **_import** keywords, and the effects of the different prolog/epilog code-generation options and their corresponding command-line compiler options.

The _export keyword

For 16-bit Windows, exported functions must be declared far; you can use the **FAR** type, defined in windows.h.

The keyword **_export** in a function definition tells the compiler to compile the function as exportable and tells the linker to export the function. In the function or class declaration, **_export** immediately precedes the function or class name; for example,

```
LRESULT CALLBACK _export MainWindowProc( HWND hWnd, UINT iMessage,
                                          WPARAM wParam, LPARAM lParam )
```

For more information on **_export**, see Chapter 2. See Chapter 9 for information on using **_export** with C++ class definitions.

The _import keyword

The keyword **_import** in a function or class tells the compiler that the function or class will be imported from a DLL. Like **_export**, **_import** is used before the function or class name. For more information on **_import**, see Chapter 2. See Chapter 9 for information on using **_import** with C++ class definitions.

Windows All Functions Exportable (-W, -WC)

This option creates a Windows application object module with all functions exportable.

The **–W** option is the most general kind of Windows application module, although not necessarily the most efficient. The compiler generates a prolog and epilog for every function, making each function exportable. This doesn't mean that all the functions actually *will* be exported; it only means that all the functions *can* be exported. To actually export one of these functions, you must either use the **_export** keyword or add an entry for the function name in the EXPORTS section of the module definition file. **–WC** is the equivalent switch for 32-bit console mode applications.

Windows Explicit Functions Exported (-WE)

The **–WE** option creates an object module containing only those functions marked as **_export** exportable.

Since, in any given application module, many of the functions won't be exported, it isn't necessary for the compiler to include the special prolog and epilog for exportable functions unless a particular function is known to be exported. The **_export** keyword in a function definition tells the compiler to use the special prolog and epilog required for exported functions. All functions not flagged with **_export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

➡ The **–WE** option works *only* in conjunction with the **_export** keyword. This option doesn't export those functions listed in the EXPORTS section of a module definition file. In fact, you can't use this option and provide the names of the exported functions in the EXPORTS section. If you do, the compiler will generate prolog and epilog code that is incompatible with exported functions; incorrect behavior will result when these functions are called.

Windows Smart Callbacks (-WS)

BCC32 doesn't use the –WS switch.

The **–WS** option creates an object module with functions using smart callbacks.

This form of prolog and epilog assumes that DS == SS; that is, it assumes the default data segment is the same as the stack segment. This eliminates the need for the special Windows code (called a *thunk*) created for exported functions. Using smart callbacks can improve performance because calls to functions in the module don't have to be redirected through the thunks.

Exported functions that use this option don't need the **_export** keyword or need to be listed in the EXPORTS section of the module definition file, because the linker doesn't need to create an export entry for them in the executable.

When you use functions compiled and linked with smart callbacks, you don't need to precede them with a call to *MakeProcInstance* (which rewrites the function's prolog in such a way that it uses a smart callback).

There are no smart callbacks for DLLs because DLLs assume DS != SS.

Because of the assumption that DS == SS, you can use this option only for applications, not for DLLs. Furthermore, you must not explicitly change DS in your program (which is an unsafe practice under Windows in any circumstance).

Windows Smart Callbacks and Explicit Functions Exported (-WSE)

The **–WSE** option creates an object module with functions that use smart callbacks, and with explicit functions exported. This is a BCC (16-bit) only switch.

Windows DLL All Functions Exportable (-WD, -WCD)

This option creates a DLL object module with all functions exportable. This prolog and epilog code is used for functions that will reside in a DLL. It also supports the exporting of these functions. This is similar to the corresponding non-DLL option. **–WCD** is the equivalent for 32-bit console mode applications.

Windows DLL Explicit Functions Exported (-WDE, -WCDE)

This prolog and epilog code is also used for functions that will reside in a DLL. However, any functions that will be exported must explicitly specify **_export** in the function definition. This is similar to the corresponding non-DLL option. **–WCDE** is the equivalent switch for 32-bit console mode applications.

Prologs, epilogs, and exports: A summary

There are two steps to exporting a function. First, the compiler must create the correct prolog and epilog for the function; at this point, the function is called exportable. Second, the linker must create an entry for every export function in the header section of the executable. This occurs in 16-bit Windows so that the correct data segment can be bound to the function at runtime. In 32-bit Windows the binding of data segments does not apply. However, DLLs must have entries in the header so the loader can find the function to link to when an .EXE loads the DLL.

If a function is flagged with the **_export** keyword and any of the Windows compiler options are used, the function is compiled as exportable and linked as an export.

If a function is *not* flagged with the **_export** keyword, Borland C++ will take one of the following actions:

- If you compile with the **–W**/**-WC** or **–WD**/**-WCD** option (or with the IDE equivalent of either option), the function will be compiled as exportable.

  If the function is listed in the EXPORTS section of the module definition file, the function will be linked as an export. If it is not listed in the module definition file, or if no module definition file is linked, then it won't be linked as an export.
- If you compile with the **–WE** or **–WDE**/**-WCDE** option (or with the IDE equivalent of either option), the function will *not* be compiled as

exportable. Including this function in the EXPORTS section of the module definition file will cause it be exported, but, because the prolog is incorrect, the program will run incorrectly. You might get a Windows error message in the 16-bit environment.

Table 8.1 summarizes the effect of the combination of the Windows compiler options and the **_export** keyword:

Table 8.1: Compiler options and the _export keyword

Function flagged with _export?	Yes	Yes	Yes	Yes	No	No	No	No
Function listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
The compiler option[1] is:	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**	**–W** or **–WD**	**–WE** or **–WDE**
Is function exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	Yes	Yes	Yes	Yes	Yes	Yes[2]	No[3]	No

[1] Or the 32-bit console mode application equivalents.

[2] The function will be exported in some sense, but because the prolog and epilog won't be correct, the function won't work as expected.

[3] This combination also makes little sense. It's inefficient to compile all functions as exportable if you don't actually export some of them.

Project files

Project files automate the process of building Windows applications when you're using the Borland C++ IDE. Project files, which have the file-name extension .PRJ, contain information about how to build a particular application. Using a tool called the project manager, you can create and maintain project files that describe each of the applications you are developing, and that build the projects into applications. Project files contain a list of the files to be processed and the switch settings for each tool used. This information is used by the project manager to automatically build the application. Project files and the project manager are the IDE equivalent of makefiles and the make utility, but project files are easier to maintain and use than makefiles.

For example, if you enter HELLO.CPP, HELLO.RC, and HELLO.DEF into a project file, the Borland C++ project manager

- Creates HELLO.OBJ by compiling HELLO.CPP with the C++ compiler
- Creates HELLO.RES by compiling HELLO.RC with the resource compiler (BRC or BRC32) or Resource Workshop

- Creates HELLO.EXE by linking HELLO.OBJ with appropriate libraries, using information contained in HELLO.DEF, and by linking the resources contained in HELLO.RES.

Use the Project Options dialog box in the IDE to set project options. This is fully explained in the *User's Guide*.

The Borland heap manager

Windows supports dynamic memory allocations on two different heaps: the *global heap* and the *local heap*.

The global heap is a pool of memory available to all applications. Although global memory blocks of any size can be allocated, the global heap is intended only for large memory blocks (256 bytes or more). Each global memory block carries an overhead of at least 20 bytes. Under the Windows standard and 386 enhanced modes, there is a system-wide limit of 8192 global memory blocks, only some of which are available to any given application.

The local heap is a pool of memory available only to your application. It exists in the upper part of an application's data segment. The total size of local memory blocks that can be allocated on the local heap is 64K minus the size of the application's stack and static data. For this reason, the local heap is best suited for small memory blocks (256 bytes or less). The default size of the local heap is 4K, but you can change this in your application's .DEF file.

Borland C++ includes a *heap manager* that implements the *new*, *delete*, *malloc*, and *free* functions. The heap manager uses the global heap for all allocations. Because the global heap has a system-wide limit of 8192 memory blocks, Borland C++'s heap manager includes a *sub-allocator* algorithm to enhance performance and allow a substantially larger number of blocks to be allocated.

This is how the segment sub-allocator works: when allocating a large block, the heap manager allocates a global memory block using the Windows *GlobalAlloc* routine. When allocating a small block, the heap manager allocates a larger global memory block and then divides (sub-allocates) that block into smaller blocks as required. Allocations of small blocks reuse all available sub-allocation space before the heap manager allocates a new global memory block, which, in turn, is further sub-allocated.

The *HeapLimit* variable defines the threshold between small and large heap blocks. *HeapLimit* is set at 64K bytes. The *HeapBlock* variable defines the size the heap manager uses when allocating blocks to be assigned to the suballocator. *HeapBlock* is set at 4096 bytes.

32-bit Windows programming

The following sections briefly describe the Win32 and Windows programming environment, and explain how to port your code to this environment. This port makes your code compilable to run on both 16 and 32-bit versions of Windows, and compilable for future processors hosting Windows.

See page 263 for 32-bit tool names, options, and libraries.

Borland C++ 32-bit tools support the production of 32-bit .OBJ and .EXE files in the portable executable (PE) file format, which is the executable file format for Win32 and Windows NT programs. Win32 conforming executables will run unchanged on Windows NT.

Win32

Win32 is an operating-system extension to Windows 3.1 that provides support for developing and running 32-bit Windows executables. Win32 is a set of DLLs that handle mapping 32-bit application program interface (API) calls to their 16-bit counterparts, a virtual device driver (VxD) to handle memory management, and a revised API called the Win32 API. The DLL and VxD are transparent.

To make sure your code will compile and run under Win32, you should

- Make sure your code adheres to the Win32 API.
- Write portable code using types and macros provided in the windows.h, and windowsx.h files.

The following sections describe the Win32 API, and explain how to write portable Windows code.

The Win32 API

For complete descriptions of Win32 API functions, see online help.

The Win32 API widens most of the existing 16-bit Windows API to 32 bits, and adds new API calls compatible with Windows NT. The Win32s API is a subset of the Win32 API for Windows NT. The Win32 API is made up of 16-bit API calls that have been converted to and are callable in the 32-bit environment, and 32-bit API calls that are implementable in the 16-bit Windows environment.

If a Win32 executable calls any of the Win32 API functions not supported under Win32, appropriate error codes are returned at run time. If you write applications that conform to the Win32 API and use the porting tips described in the next section, your application will be portable across 16 and 32-bit Windows environments.

Writing portable Windows code

This section discusses portability constructs (which were introduced in Windows 3.1) that will assist you in producing portable Windows code. Existing 16-bit Windows code can be ported to Win32 and Windows NT with minimal changes. Most changes involve substituting new macros and types for old, and replacing any 16-bit specific API calls with analogous Win32 API calls. Once these changes have been made, your code can compile and run under 16 or 32-bit Windows.

A compile-time environment variable, STRICT, has been provided to assist you in producing portable code.

STRICT

Windows 3.1 introduced support in windows.h for defining STRICT. Defining STRICT enables strict compiler error checking. For example, if STRICT is not defined, passing an *HWND* to a function that requires an *HDC* will not cause a compiler warning. If you define STRICT, you get a compiler error.

Using STRICT enables

- Strict handle type checking
- Correct and consistent parameter and return-value type declarations
- Fully prototyped type definitions for callback function types (window, dialog, and hook procedures)
- ANSI-compliant declaration of COMM, DCB, and COMSTAT structures

STRICT is Windows 3.0 backward compatible. It can be used with the 3.1 WINDOWS.H for creating applications that will run under Windows 3.0.

Defining STRICT will help you locate and correct type incompatibilities that arise when migrating your code to 32 bits, and will aid portability between 16 and 32-bit Windows.

New types, constants, and macros have been provided so you can change your source code to be STRICT compliant. Table 8.2 lists the types, macros, and handle types you can use to make your application STRICT compliant.

Table 8.2: STRICT compliant types, constants, helper macros and handles

Types and constants	**Description**
CALLBACK	Use instead of FAR PASCAL in your callback routines (for example, window and dialog procedures).
LPARAM	Declares all 32-bit polymorphic parameters.
LPCSTR	Same as *LPSTR*, except that it is used for read-only string pointers.
LRESULT	Declares all 32-bit polymorphic return values.
UINT	Portable unsigned integer type whose size is determined by the targeted environment. Represents a 16-bit value on Windows 3.1, and a 32-bit value on Win32.
WINAPI	Use instead of FAR PASCAL for API declarations. If you are writing a DLL with exported API entry points, you can use this for the API declarations.
WPARAM	Declares all 16-bit polymorphic parameters.

Macros	**Description**
FIELDOFFSET(*type,field*)	Calculates the field offsets in a structure. *type* is the structure type, and *field* is the field name.
MAKELP(*sel,off*)	Takes a selector and offset and produces a *FAR VOID**.
MAKELPARAM(*low,high*)	Makes an *LPARAM* out of two 16-bit values.
MAKELRESULT(*low,high*)	Makes an *LRESULT* out of two 16-bit values.
OFFSETOF(*lp*)	Extracts the offset of a far pointer and returns a *UINT*.
SELECTOROF(*lp*)	Extracts the selector for a far pointer and returns a *UINT*.

Handles	**Description**
HACCEL	Accelerator table handle
HDRVR	Driver handle (Windows 3.1 only)
HDWP	*DeferWindowPost()* handle
HFILE	File handle
HGDIOBJ	Generic GDI object handle
HGLOBAL	Global handle
HINSTANCE	Instance handle
HLOCAL	Local handle
HMETAFILE	Metafile handle
HMODULE	Module handle
HRSRC	Resource handle
HTASK	Task handle

Making your code STRICT compliant

To make your application STRICT compliant,

Because of C++ type-safe linking, linking STRICT and non-STRICT modules might cause linker errors in C++ applications.

1. Decide what code you want to be STRICT compliant. Converting your code to STRICT can be done in stages.
2. Turn on the compiler's highest error/warning level. In the IDE, use Options | Compiler | Messages | Display | All. With BCC32 use the **-w** switch. You might want to compile at this stage, before taking the next step.
3. `#define` STRICT before including windows.h and compile, or use -DSTRICT on the command line.

STRICT conversion hints

This section describes some common coding practices you should use when converting your code to STRICT compliance.

- Change *HANDLE* to the appropriate specific handle type; for example, *HMODULE*, *HINSTANCE*, and so on.
- Change *WORD* to *UINT* except where you specifically want a 16-bit value on a 32-bit platform.
- Change *WORD* to *WPARAM*.
- Change *LONG* to *LPARAM* or *LRESULT* as appropriate.
- Change *FARPROC* to *WNDPROC*, *DLGPROC*, or *HOOKPROC* as appropriate.
- For 16-bit Windows, always declare function pointers with the proper function type rather than with *FARPROC*. You'll need to cast function pointers to and from the proper function type when using *MakeProcInstance*, *FreeProcInstance*, and other functions that take or return a *FARPROC*; for example,

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT msg,
                      WPARAM wParam,
                      LPARAM lParam);
                      DLGPROC lpfnDlg;

lpfnDlg=(DLGPROC)MakeProcInstance(DlgProc, hinst);
...
FreeProcInstance((FARPROC)lpfnDlg);
```

- Take special care with *HMODULE*s and *HINSTANCE*s. The Kernel module-management functions generally use *HINSTANCE*s, but a few APIs return or accept only *HMODULE*s.

- If you've copied any API function declarations from WINDOWS.H, they might have changed, and your local declaration might be out of date. Remove your local declarations.
- Cast the results of *LocalLock* and *GlobalLock* to the proper kind of data pointer. Parameters to these and other memory management functions should be cast to *LOCALHANDLE* or *GLOBALHANDLE*, as appropriate.
- Cast the result of *GetWindowWord* and *GetWindowLong* and the parameters to *SetWindowWord* and *SetWindowLong*.
- When casting *SendMessage*, *DefWindowProc*, and *SendDlgItemMsg* or any other function that returns an *LRESULT* or *LONG* to a handle of some kind, you must first cast the result to a *UINT*:

```
HBRUSH hbr;
hbr = (HBRUSH)(UINT)SendMessage(hwnd, WM_CTLCOLOR, ..., ...);
```

- The *CreateWindow* and *CreateWindowEx* hmenu parameter is sometimes used to pass an integer control ID. In this case you must cast this to an *HMENU*:

```
HWND hwnd;
int id;
hwnd = CreateWindow("Button", "Ok", BS_PUSHBUTTON,
                    x, y, cx, cy, hwndParent,
                    (HMENU)id, //Cast required here
                    hinst, NULL);
```

See page 259 for a description of message crackers.

- Polymorphic data types (*WPARAM*, *LPARAM*, *LRESULT*, **void** *FAR**) should be assigned to variables as soon as possible. Avoid using them in your own code when the type of the value is known; this will minimize the number of potentially unsafe and non-32-bit-portable casting you will have to do in your code. The macro APIs and message cracker mechanisms provided in windowsx.h will take care of almost all packing and unpacking of these data types, in a 32-bit portable way.
- Become familiar with the common compiler warnings and errors that you're likely to encounter as you convert to STRICT.

Some of the most common compiler errors and warnings you might encounter are described on page 260.

The UINT and WORD types

The type *UINT* has been created and used extensively in the API to create a data type portable from Windows 3.x. *UINT* is defined as

```
typedef unsigned int UINT;
```

UINT is needed because of the difference in **int** sizes between 16-bit Windows and Win32. For 16-bit Windows, **int** is a 16-bit unsigned integer;

for Win32, **int** is a 32-bit unsigned integer. Use *UINT* to declare integer objects expected to widen from 16 to 32 bits when compiling 32-bit applications.

The type *WORD* is defined as

```
typedef unsigned short WORD;
```

WORD declares a 16-bit value on both 16-bit Windows and Win32. Use *WORD* to create objects that will remain 16 bits wide across both platforms. Note that because Win32 handles are widened to 32 bits, *WORD* can no longer be used for handles.

The WINAPI and CALLBACK calling conventions

The windows.h macro WINAPI defines the calling convention. WINAPI resolves to the appropriate calling convention for the targeted platform. WINAPI should be used in place of FAR PASCAL.

For example, here is an important change necessary for window procedure definitions. The following is code as it would appear in 16-bit Windows:

```
LONG FAR PASCAL WindowProc(HANDLE hWnd, unsigned message
                WORD wParam, LONG lParam)
```

Here is the Win32 version:

```
LONG WINAPI WindowProc(HWND hWnd, UINT message
          UINT wParam, LONG lParam)
```

Using WINAPI allows specifying alternative calling conventions. Currently Win32 uses **__stdcall**. The fundamental type **unsigned** is changed to the more portable *UINT*. *WORD* is also changed to *UINT*, in this case illustrating the expansion of *wParam* to 32 bits. If this change to *wParam* isn't made, the result will be application failure during initial window creation.

Use the CALLBACK calling convention in your callback function declarations. This replaces FAR PASCAL.

Extracting message data

In 32-bit Windows code you need to change the way you unpack message data from *lParam* and *wParam*. In Win32, *wParam* grows from 16 to 32 bits in size, while *lParam* remains 32 bits wide. But since *lParam* frequently contains a handle and another value in 16-bit Windows, and a handle grows to 32 bits under Win32, a new *wParam* and *lParam* packing scheme was necessary.

For example, WM_COMMAND is one of the messages affected by the changes to extra parameter size. Under Windows 3.*x*, *wParam* contains a 16-bit identifier and *lParam* contains both a 16-bit window handle and a 16-bit command.

Under Win32, *lParam* contains only the window handle because window handles are now 32 bits. Therefore, the 16-bit command is moved from *lParam* to the low-order 16 bits of *wParam* (now 32 bits), with the high-order 16 bits of *wParam* containing the identifier. This repacking means you need to change the way you extract information from these parameters; for example, by using *message crackers*, which are described in the next section.

Message crackers

Message crackers are a portable way of extracting messages from *wParam* and *lParam*. Depending on your environment (16-bit Windows or Win32) message crackers use a different technique for extracting the message data. Each Windows message has a set of message crackers.

For example, here is the 32-bit version of the WM_COMMAND message crackers:

```
#define GET_WM_COMMAND_ID(wp, lp)               LOWORD(wp)
#define GET_WM_COMMAND_HWND(wp, lp)             (HWND)(lp)
#define GET_WM_COMMAND_CMD(wp, lp)              HIWORD(wp)
#define GET_WM_COMMAND_MPS(id, hwnd, cmd)     \
        (WPARAM)MAKELONG(id, cmd), (LONG)(hwnd)
```

And here is the 16-bit version of the WM_COMMAND message crackers:

```
#define GET_WM_COMMAND_ID(wp, lp)                    (wp)
#define GET_WM_COMMAND_HWND(wp, lp)                  (HWND)LOWORD(lp)
#define GET_WM_COMMAND_CMD(wp, lp)                  HIWORD(lp)
#define GET_WM_COMMAND_MPS(id, hwnd, cmd)     \
        (WPARAM)(id), MAKELONG(hwnd, cmd)
```

Using these message-cracker macros ensures that your message extraction code is portable to either platform.

Porting DOS system calls

Windows 3.0 provided the *DOS3Call* API function for calling DOS file I/O functions. This API function, and other INT 21H DOS functions, are replaced in Win32 by named 32-bit calls. Table 8.3 lists DOS INT 21H calls and their equivalent Win32 API function.

Table 8.3
Int 21 and Win32 equivalent functions

INT 21H function	**DOS operation**	**Win32 API equivalent**
0EH	Select disk	*SetCurrentDirectory*
19H	Get current disk	*GetCurrentDirectory*
2AH	Get date	*GetDateAndTime*
2BH	Set date	*SetDateAndTime*
2CH	Get time	*GetDateAndTime*
2DH	Set time	*SetDateAndTime*
36H	Get disk free space	*GetDiskFreeSpace*

Table 8.3: Int 21 and Win32 equivalent functions (continued)

39H	Create directory	*CreateDirectory*
3AH	Remove directory	*RemoveDirectory*
3BH	Set current directory	*SetCurrentDirectory*
3CH	Create handle	*CreateFile*
3DH	Open handle	*CreateFile*
3EH	Close handle	*CloseHandle*
3FH	Read handle	*ReadFile*
40H	Write handle	*WriteFile*
41H	Delete file	*DeleteFile*
42H	Move file pointer	*SetFilePointer*
43H	Get file attributes	*GetAttributesFile*
43H	Set file attributes	*SetAttributesFile*
47H	Get current directory	*GetCurrentDirectory*
4EH	Find first file	*FindFirstFile*
4FH	Find next file	*FindNextFile*
56H	Change directory entry	*MoveFile*
57H	Get file date/time	*GetDateAndTimeFile*
57H	Set file date/time	*SetDateAndTimeFile*
59H	Get extended error	*GetLastError*
5AH	Create unique file	*GetTempFileName*
5BH	Create new file	*CreateFile*
5CH	Lock file	*LockFile*
5CH	Unlock file	*UnlockFile*
67H	Set handle count	*SetHandleCount*

Common compiler errors and warnings

This section describes some of the common compiler errors and warnings you might get when trying to compile your application with all messages enabled and with or without STRICT defined.

Call to function *funcname* with no prototype (warning)
A function was used before it was prototyped or declared. This warning can also arise when a function that takes no arguments is not prototyped with **void**:

```
void bar(); /* Should be: bar(void) */
void main(void)
{
bar();
}
```

Conversion may lose significant digits (warning)
This warning results when a value is converted by the compiler, such as from LONG to **int**. You're being warned because you might lose information from this cast. If you're sure there are no information-loss problems, you can suppress this warning with the appropriate explicit cast to the smaller type.

Function should return a value (warning)
A function declared to return a value does not return a value. In older, non-ANSI C code, it was common for functions that didn't return a value to have no return type:

```
foo(i)
int i;
{
...
}
```

Functions declared in this manner are treated by the compiler as being declared to return an **int**. If the function doesn't return anything, it should be declared **void**:

```
void foo(int i)
{
...
}
```

Lvalue required (error)
Type mismatch in parameter (error)
These errors indicate that you are trying to assign or pass a nonpointer type when a pointer type is required. With STRICT defined, all handle types as well as *LRESULT*, *WPARAM*, and *LPARAM* are internally declared as pointer types, so trying to pass an **int**, *WORD*, or *LONG* as a handle will result in these errors.

These errors should be fixed by properly declaring the nonpointer values you're assigning or passing. In the case of special constants such as `(HWND)1` to indicate "insert at bottom" to the window-positioning functions, you should use the new macros (such as HWND_BOTTOM). Only in rare cases should you suppress a type-mismatch error with a cast (because this can often generate incorrect code).

Non-portable pointer conversion (warning)
You cast a near pointer or a handle to a 32-bit value such as *LRESULT*, *LPARAM*, *LONG*, or *DWORD*. This warning almost always represents a bug, because the high-order 16 bits of the value will contain a nonzero value. The compiler first converts the 16-bit near pointer to a 32-bit far pointer by placing the current data segment value in the high 16 bits; it then converts this far pointer to the 32-bit value.

To avoid this warning and ensure that a 0 is placed in the high 16 bits, you must first cast the handle to a *UINT*:

```
HWND hwnd;
LRESULT result = (LRESULT)(UINT)hwnd;
```

In cases where you do want the 32-bit value to contain a far pointer, you can avoid the warning with an explicit cast to a far pointer:

```
char near* pch;
LPARAM lParam = (LPARAM)(LPSTR)pch;
```

Not an allowed type (error)
This error typically results from trying to dereference a void pointer. It usually results from directly using the return value of GlobalLock or LocalLock as a pointer. To solve this problem, assign the return value to a variable of the appropriate type (with the appropriate cast) before using the pointer:

```
BYTE FAR* lpb = (BYTE FAR*)GlobalLock(h);
*lpb = 0;
```

Parameter *paramname* is never used (warning)
This message can result in callback functions when your code does not use certain parameters. You can either turn off this warning, use **#pragma argsused** to suppress it, or omit the name of the parameter in the function definition.

Size of the type is unknown or zero (error)
You are trying to change the value of a void pointer with + or +=. This error typically results from the fact that certain Windows functions that return pointers to arbitrary types (such as *GlobalLock* and *LocalLock*) are defined to return **void** *FAR** rather than LPSTR.

To solve these problems, assign the **void*** value to a properly declared variable (with the appropriate cast):

```
BYTE FAR* lpb = (BYTE FAR*)GlobalLock(h);
lpb += sizeof(DWORD);
```

Type mismatch in redeclaration of *paramname* (error)
You have inconsistent declarations of a variable, parameter, or function in your source code.

By adhering to the Win32 API, and using STRICT to make code changes you will make your Windows code portable. The next few sections describe some of the new types and macros and how to use them.

Building Win32 executables

You must use the proper tools, switches, libraries, and start-up code to build a Win32 application. Table 8.4 lists the compiler (BCC32) and linker (TLINK32) switches, libraries and start-up code needed when linking, and the resulting executable type (.DLL or .EXE).

Table 8.4
Win32 options, start-up code, and libraries

BCC32 options	TLINK32 option	Libraries	Start-up code	Creates this executable type
–W, –WE	/Tpe	cw32.lib import32.lib	c0w32.obj	GUI .EXE
–WD, –WDE	/Tpd	cw32.lib imprtw32.lib	c0d32.obj	GUI .DLL
–WC	/Tpe /ap	cx32.lib import32.lib	c0x32.obj	Console .EXE
–WCD, –WCDE	/Tpd /ap	cx32.lib imprtw32.lib	c0d32.obj	Console .DLL

CHAPTER 9

Writing dynamic-link libraries

See the *ObjectWindows Programmer's Guide* for more information on DLLs.

This chapter defines dynamic-link libraries (DLLs) and describes how to write them. Using DLLs in your applications reduces .EXE file size, conserves system memory, and provides more flexibility in changing, extending, or upgrading your applications.

What is a DLL?

A DLL is an executable library module containing functions or resources for use by applications or other DLLs. DLLs have no *main* function, which is the usual entry point for an application. Instead, DLLs have multiple entry points, one for each exported function.

When a DLL is loaded by the operating system, the DLL can be shared among multiple applications; one loaded copy of the DLL is all that's necessary.

To fully understand DLLs, it is helpful to understand how dynamic linking and static linking differ.

Dynamic linking

When an application uses a function from a static-link library (for example, the C run-time library), a copy of that function is bound to your application by TLINK at link time. Two simultaneously running applications that are using the same function would each have their own copy of that function. It would be more efficient, however, if the applications shared a single copy of the function. Dynamic linking provides this capability by resolving your application's references to external functions at run time.

Windows supports both dynamic and static linking.

When a program uses a function from a DLL, the function code isn't linked into the .EXE. Dynamic linking uses a different, two-step method:

1. At link time, TLINK binds import records (which contain DLL and procedure-location information) to your .EXE. This temporarily satisfies any external references to DLL functions in your code. These import records are supplied by module-definition files or import libraries.

2. At run time, the import-record information is used to locate and bind the DLL functions to your program.

With dynamic linking, your applications are smaller because copies of the function's code aren't linked into your application. And because the DLL's code and resources are shared among applications, system memory is conserved.

Creating a DLL

DLLs are created similarly to .EXEs: source files containing your code are compiled, then the .OBJs are linked together. The DLL, however, has no *main* function, and is therefore linked differently. The following sections describe how to write a DLL.

LibMain, DllEntryPoint, and WEP

You must supply the *LibMain* function (for 16-bit programs) or the *DllEntryPoint* function (for 32-bit programs) as the main entry point for a DLL.

For 16-bit programs, Windows calls *LibMain* once, when the library is first loaded. *LibMain* performs initialization for the DLL. For 32-bit programs, Windows calls *DllEntryPoint* each time the DLL is loaded and unloaded (it replaces WEP for 32-bit applications), each time an additional process attaches to or detaches from the DLL, or each time a thread within the process is created or destroyed. For more on *DllEntryPoint*, see online Help.

DLL initialization depends almost entirely on the function of the particular DLL, but might include the following typical tasks:

- Unlocking the data segment with *UnlockData*, if it has been declared as MOVEABLE. (16-bit only.)
- Setting up global variables for the DLL, if it uses any.

The initialization code is executed only for the first application using the DLL.

➡ The DLL startup code initializes the local heap automatically; you don't need to include code in *LibMain* to do this. (16-bit only.)

The following parameters are passed to *LibMain*:

HINSTANCE, WORD, and LPSTR are defined in windows.h.

```
int FAR PASCAL LibMain (HINSTANCE hInstance, WORD wDataSeg, WORD cbHeapSize,
                        LPSTR lpCmdLine)
```

- *hInstance* is the instance handle of the DLL.
- *wDataSeg* is the value of the data segment (DS) register.

- *cbHeapSize* is the size of the local heap specified in the module-definition file for the DLL.
- *lpCmdLine* is a far pointer to the command line specified when the DLL was loaded. This is almost always null because DLLs are typically loaded automatically with no parameters. It is possible, however, to supply a command line to a DLL when it is loaded explicitly.

The return value for *LibMain* is either 1 (successful initialization) or 0 (failure in initialization). If *LibMain* is 0, Windows unloads the DLL from memory.

The exit point for a 16-bit DLL is the function *WEP* (Windows Exit Procedure). This function isn't required in a DLL (because the Borland C++ run-time libraries provide a default) but you can supply your own WEP to perform any DLL cleanup before the DLL is unloaded from memory. Windows calls *WEP* just prior to unloading the DLL.

Under Borland C++, *WEP* doesn't need to be exported. Borland C++ defines its own *WEP* that calls your *WEP* (if you've defined one), and then performs system cleanup. This is the prototype for *WEP*:

```
int FAR PASCAL WEP (int nParameter)
```

nParameter is either WEP_SYSTEMEXIT or WEP_FREE_DLL. WEP_SYSTEMEXIT indicates that all of Windows is shutting down and WEP_FREE_DLL indicates that only this DLL is being unloaded.

WEP returns 1 to indicate success. Windows currently doesn't do anything with this return value.

Exporting and importing functions

To make your DLL functions accessible to other applications (.EXEs or other DLLs) the function names must be *exported*. To use exported functions, the function names must be *imported*. The following sections describe how to export and import function names with Borland C++.

Exporting functions

See Chapter 9 in the *User's Guide* for more information on module-definition files.

There are two ways to export functions:

- Create a module-definition file with an EXPORTS section listing all functions that will be used by other applications. The IMPDEF tool can help you do this; see Chapter 11 in the *User's Guide*.
- Precede every function name to be exported in the DLL with the keyword **_export** in the function definition. In addition, when you build or link the DLL, you must choose the correct code-generation option in the IDE (see Chapter 1 of the *User's Guide*), or the correct command-line compiler option (see Chapter 3 of the *User's Guide*).

A function must be exported from a DLL before it can be imported to another DLL or application.

Importing functions

If a Windows application module or another DLL uses functions from a DLL, you must tell the linker that you want to import the functions. There are three ways to do this:

For more information on exporting and importing functions, see Chapter 11 in the *User's Guide.*

- Add an IMPORTS section to the module-definition file and list every DLL function that the module will use.
- Include the import library for the DLLs when you link the module. The IMPLIB tool creates an import library for one or more DLLs.
- Define your function using the **_import** keyword (32-bit applications only).

For more information on the **_export** and **_import** keywords, see Chapter 1.

DLLs and 16-bit memory models

Functions in a DLL are not linked directly into a Windows application; instead, they are called at run time. This means that calls to DLL functions will be far calls, because the DLL will have a different code segment than the application. The data used by called DLL functions will also need to be far.

Let's suppose you have a Windows application called APP1, a DLL defined by LSOURCE1.C, and a header file for that DLL called lsource1.h. Function *f1*, which operates on a string, is called by the application.

If you want the function to work correctly regardless of the memory model the DLL will be compiled under, you need to explicitly make the function and its data far. In the header file lsource1.h, the function prototype would take this form:

```
extern int _export FAR f(char FAR *dstring);
```

In the DLL source LSOURCE1.C, the implementation of the function would take this form:

```
int FAR f1(char far *dstring)
{
⋮
}
```

For the function to be used by the application, the function would also need to be compiled as exportable and then exported. To accomplish this, you can either compile the DLL with all functions exportable (**-WD**) and list *f1* in the EXPORTS section of the module-definition file, or you can flag the function with the **_export** keyword, like this:

```
int FAR _export f1(char far *dstring)
{
:
}
```

If you compile the DLL under the large model (far data, far code), then you don't need to explicitly define the function or its data far in the DLL. In the header file, the prototype would still take the form shown here because the prototype would need to be correct for a module compiled with a smaller memory model:

```
extern int FAR f1(char FAR *dstring);
```

In the DLL, however, the function could be defined like this:

```
int _export f1(char *dstring)
{
:
}
```

See Chapter 11 in the *User's Guide* for more information about import libraries.

Remember that before an application can use *f1*, it has to be imported into the application, either by listing *f1* in the IMPORTS section of a module-definition file or by linking with an import library for the DLL.

Exporting and importing classes

To use classes in a DLL, the class must be exported from the .DLL file and imported by the .EXE file. Conditionalized macro expansion can be used to support both of these circumstances. For example, include something similar to the following code in a header file:

```
#if defined (BUILDING_YOUR_DLL)
   #define _YOURCLASS _export
#elif defined(USING_YOUR_DLL)
   #define _YOURCLASS _import
#else
   #define _YOURCLASS
#endif
```

In your definitions, define your classes like this:

```
class _YOURCLASS class1 {

// ...

};
```

For more on exporting and importing classes, see Chapter 3.

Define BUILDING_YOUR_DLL (with the **-D** option, for example) when you are building your DLL. The _YOURCLASS macro will expand to **_import**. Define USING_YOUR_DLL when you are building the .EXE that will use the DLL. The _YOURCLASS macro will expand to **_import**.

For additional information on exporting and importing, see Chapters 1 and 3 in the *User's Guide*.

Static data in 16-bit DLLs

Through a DLL's functions, all applications using the DLL have access to that DLL's global data. In 16-bit DLLs, a particular function will use the same data, regardless of the application that called it (unlike 32-bit DLLs, where all data is private to the process). If you want a 16-bit DLL's global data to be protected for use by a single application, you need to write that protection yourself. The DLL itself does not have a mechanism for making global data available to a single application. If you need data to be private for a given caller of a DLL, you need to dynamically allocate the data and manage the access to that data manually. Static data in a 16-bit DLL is global to all callers of a DLL. See Chapter 9 for more information on data in DLLs.

Using the Borland DLLs

General forms of compiler and linker command lines that use the DLL versions of the Borland run-time libraries and class libraries are described below.

Here is a 16-bit compile and link using the DLL version of the run-time library:

```
bcc -c -D_RTLDLL -ml source.cpp

tlink -C -Twe c0wl source, source, , import crtldll
```

Note that the macro _RTLDLL and the **-ml** switch are used. Here is the 32-bit version:

```
bcc32 -c -D_RTLDLL source.cpp

tlink32 -Tpe -ap c0x32 source, source, , import32 cw32i
```

Here is a 16-bit compile and link using the DLL version of the class library:

```
bcc -c -D_BIDSDLL -ml source.cpp

tlink -C -Twe c0wl source, source, , import bidsi crtldll
```

Here is a 32-bit compile and link using the DLL version of the class library:

```
bcc32 -c -D_BIDSDLL source.cpp

tlink32 -Tpe -ap c0x32 source, source, , import32 bidsfi cw32i
```

Using inline assembly

Inline assembly is assembly-language instructions embedded within your C or C++ code. Inline assembly instructions are compiled or assembled along with your code rather than being assembled in separate assembly modules.

This chapter describes how to use inline assembly with Borland C++. The following topics are discussed:

- Inline assembly syntax and usage
  - Using the **asm** keyword to place an assembly instruction within your C/C++ code
  - Using C symbols in your **asm** statements to reference data and functions
  - Using register variables, offsets, and size overrides
  - Using C structure members
  - Using jump instructions and labels

See Chapter 3 of the *User's Guide* for the IDE equivalents of command-line options

- Using the **–B** compiler option and #**pragma inline** statement to compile inline assembly
- Using the built-in assembler (BASM)

Inline assembly syntax and usage

This section describes inline assembly syntax, and how to use inline assembly instructions with C++ structures, pointers, and identifiers.

To place an assembly instruction in your C/C++ code, use the **asm** keyword. The format is

asm *opcode operands* ; or *newline*

where

- *opcode* is a valid 80x86 instruction.

- *operands* contains the operand(s) acceptable to the *opcode*, and can reference C constants, variables, and labels.
- The end of the **asm** statement is signaled by either ; (semicolon) or by *newline* (a new line).

A new **asm** statement can be placed on the same line, following a semicolon, but no **asm** statement can continue to the next line.
To include multiple **asm** statements, surround them with braces. The initial brace must appear on the same line as the **asm** keyword.

Three **asm** statements are shown here; two on one line, and one below them.

```
asm {
   pop ax; pop ds
   iret
}
```

Semicolons are not used to start comments (as they are in TASM). When commenting **asm** statements, use C-style comments, like this:

```
asm mov ax,ds;                    /* This comment is OK */
asm {pop ax; pop ds; iret;}       /* This comment is also legal */
asm push ds                       ;THIS COMMENT IS INVALID!!
```

The assembly-language portion of the statement is copied straight to the output, embedded in the assembly language that Borland C++ is generating from your C or C++ instructions. Any C symbols are replaced with appropriate assembly language equivalents.

Each **asm** statement is considered to be a C statement. For example, the following construct is a valid C **if** statement:

```
myfunc()
{
   int  i;
   int x;

   if  (i > 0)
      asm  mov  x,4
   else
      i = 7;
}
```

Note that a semicolon isn't needed after the `mov x,4` instruction. **asm** statements are the only statements in C that depend on the occurrence of a new line to indicate that they have ended. Although this isn't in keeping with the rest of the C language, it is the convention adopted by several UNIX-based compilers.

An **asm** statement can be used as an executable statement inside a function, or as an external declaration outside of a function. **asm** statements located inside functions are placed in the code segment, and **asm** statements located outside functions are placed in the data segment.

Inline assembly references to data and functions

You can use any C symbol in your **asm** statements, including automatic (local) variables, register variables, and function parameters. Borland C++ automatically converts these symbols to the appropriate assembly-language operands and appends underscores onto identifier names.

In general, you can use a C symbol in any position where an address operand would be legal. Of course, you can use a register variable wherever a register would be a legal operand.

If the assembler encounters an identifier while parsing the operands of an inline-assembly instruction, it searches for the identifier in the C symbol table. The names of the 80x86 registers are excluded from this search. Either uppercase or lowercase forms of the register names can be used.

Inline assembly and register variables

Inline assembly code can freely use SI or DI as scratch registers. If you use SI or DI in inline assembly code, the compiler won't use these registers for register variables.

Inline assembly, offsets, and size overrides

When programming, you don't need to be concerned with the exact offsets of local variables: using the variable name will include the correct offsets.

It might be necessary, however, to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. A DWORD PTR override is needed on LES or indirect far call instructions.

Using C structure members

You can reference structure members in an inline-assembly statement in the usual way (that is, with *variable.member*). When you do this, you are working with variables, and you can store or retrieve values in these structure members. However, you can also directly reference the member name (without the variable name) as a form of numeric constant. In this situation, the constant equals the offset (in bytes) from the start of the structure containing that member. Consider the following program fragment:

```
struct myStruct {
   int a_a;
   int a_b;
   int a_c;
} myA ;

myfunc()
{
```

```
   ...
   asm  {mov  ax, WORD PTR myA.a_b
         mov  bx, WORD PTR myA.a_c
        }
   ...
}
```

This fragment declares a structure type named *myStruct* with three members: *a_a*, *a_b*, and *a_c*. It also declares a variable *myA* of type *myStruct*. The first inline-assembly statement moves the value contained in *myA.a_b* into the register AX. The second moves the value at the address *[di]* + *offset(a_c)* into the register BX (it takes the address stored in DI and adds to it the offset of *a_c* from the start of *myStruct*). In this sequence, these assembler statements produce the following code:

```
mov  ax, DGROUP : myA+2
mov  bx, [di+4]
```

This way, if you load a register (such as DI) with the address of a structure of type *myStruct*, you can use the member names to directly reference the members. The member name can be used in any position where a numeric constant is allowed in an assembly-statement operand.

The structure member must be preceded by a dot (.) to signal that a member name, rather than a normal C symbol, is being used. Member names are replaced in the assembly output by the numeric offset of the structure member (the numeric offset of *a_c* is 4), but no type information is retained. Thus members can be used as compile-time constants in assembly statements.

There is one restriction, however: if two structures that you're using in inline assembly have the same member name, you must distinguish between them. Insert the structure type (in parentheses) between the dot and the member name, as if it were a cast. For example,

```
asm   mov    bx,[di].(struct tm)tm_hour
```

Using jump instructions and labels

You can use any of the conditional and unconditional jump instructions, plus the loop instructions, in inline assembly. These instructions are valid only inside a function. Since no labels can be defined in the **asm** statements, jump instructions must use C **goto** labels as the object of the jump. If the label is too far away, the jump will not be automatically converted to a long-distance jump. For this reason, you should be careful when inserting conditional jumps. You can use the **–B** switch to check your jumps. Direct far jumps cannot be generated.

In the following code, the jump goes to the C **goto** label *a*.

```
int     x()
{
a:                           /* This is the goto label "a" */
   ...
   asm  jmp  a               /* Goes to label "a" */
   ...
}
```

Indirect jumps are also allowed. To use an indirect jump, use a register name as the operand of the jump instruction.

Compiling with inline assembly

There are two ways Borland C++ can handle inline assembly code in your C or C++ code.

- Borland C++ can convert your C or C++ code into assembly language, then transfer to TASM to produce an .OBJ file. (This method is described in this section.)
- Borland C++ can use its built-in assembler (BASM) to insert your assembly statements directly into the compiler's instruction stream (16-bit compiler only). (This method is described in the following section.)

By default **–B** invokes TASM or TASM32. You can override it with **–E*xxx***, where xxx is another assembler. See Chapter 3 in the *User's Guide* for details.

You can use the **–B** compiler option for inline assembly in your C or C++ program. If you use this option, the compiler first generates an assembly file, then invokes TASM on that file to produce the .OBJ file.

You can invoke TASM while omitting the **–B** option if you include the **#pragma inline** statement in your source code. This statement enables the **–B** option for you when the compiler encounters it. You will save compile time if you put **#pragma inline** at the top of your source file.

The **–B** option and **#pragma inline** tell the compiler to produce an .ASM file, which might contain your inline assembly instructions, and then transfer to TASM to assemble the .OBJ file. The 16-bit Borland C++ compiler has another method, BASM, that allows the compiler, not TASM, to assemble your inline assembly code.

Using the built-in assembler (BASM)

The 16-bit compiler can assemble your inline assembly instructions using the built-in assembler (BASM). This assembler is part of the compiler, and can do most of the things TASM can do, with the following restrictions:

- It can't use assembler macros.
- It can't handle 80386 or 80486 instructions.
- It doesn't permit Ideal mode syntax.
- It allows only a limited set of assembler directives (see page 278).

Because BASM isn't a complete assembler, it might not accept some assembly-language constructs. If this happens, Borland C++ will issue an error message. You then have two choices: you can simplify your inline assembly-language code so the assembler will accept it, or you can use the **–B** option to invoke TASM to catch whatever errors there might be. TASM might not identify the location of errors, however, because the original C source line number is lost.

Opcodes

You can include any of the 80x86 instruction opcodes as inline-assembly statements. There are four classes of instructions allowed by the Borland C++ compiler:

- Normal instructions—the regular 80x86 opcode set
- String instructions—special string-handling codes
- Jump instructions—various jump opcodes
- Assembly directives—data allocation and definition

All operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

Table 10.1 lists all allowable BASM opcodes. For 80286 instructions, use the **–2** command-line compiler option.

Table 10.1 BASM opcode mnemonics

aaa
aad
aam
aas
adc
add
and
bound
call
cbw
clc
cld
cli
cmc
cmp
cwd
fdivrp
feni
ffree*
fiadd
ficom
ficomp
fidiv
fidivr
fild
fimul
fincstp*
finit
fist
fistp
fisub
fisubr
fpatan
fprem
fptan
frndint
frstor
fsave
fscale
fsqrt
fst
fstcw
fstenv
fstp
fstsw
fsub
fsubp
fsubr
lsl
mov
mul
neg
nop
not
or
out
pop
popa
popf
push
pusha
pushf
rcl
rcr

Table 10.1: BASM opcode mnemonics (continued)

daa	fld	fsubrp	ret
das	fld1	ftst	rol
dec	fldcw	fwait	ror
div	fldenv	fxam	sahf
enter	fldl2e	fxch	sal
f2xm1	fldl2t	fxtract	sar
fabs	fldlg2	fyl2x	sbb
fadd	fldln2	fyl2xp1	shl
faddp	fldpi	hlt	shr
fbld	fldz	idiv	smsw
fbstp	fmul	imul	stc
fchs	fmulp	in	std
fclex	fnclex	inc	sti
fcom	fndisi	int	sub
fcomp	fneni	into	test
fcompp	fninit	iret	verr
fdecstp*	fnop	lahf	verw
fdisi	fnsave	lds	wait
fdiv	fnstcw	lea	xchg
fdivp	fnstenv	leave	xlat
fdivr	fnstsw	les	xor

* Not supported if you're using inline assembly in routines that use floating-point emulation (the command-line compiler option **–f**).

When using 80186 instruction mnemonics in your inline-assembly statements, you must include the **–1** command-line option. This forces appropriate statements into the assembly-language compiler output so that the assembler will expect the mnemonics. If you're using an older assembler, these mnemonics might not be supported.

String instructions

In addition to the opcodes listed in Table 1.1, the string instructions given in Table 1.2 can be used alone or with repeat prefixes.

Table 10.2
BASM string
instructions

cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	scasb	stosw	
lodsw	outsb	scasw		
insb	movs			

The following prefixes can be used with the string instructions:

`lock    rep    repe    repne    repnz    repz`

Jump instructions

Jump instructions are treated specially. Because a label can't be included on the instruction itself, jumps must go to C labels (see the "Using jump instructions and labels" section on page 274). The allowed jump instructions are given in the next table.

Table 10.3
Jump instructions

`ja`
`jae`
`jb`
`jbe`
`jc`
`jcxz`
`je`
`jg`
`jge`
`jl`
`jle`
`jmp`
`jna`
`jnae`
`jnb`
`jnbe`
`jnc`
`jne`
`jng`
`jnge`
`jnl`
`jnle`
`jno`
`jnp`
`jns`
`jnz`
`jo`
`jp`
`jpe`
`jpo`
`js`
`jz`
`loop`
`loope`
`loopne`
`loopnz`
`loopz`

Assembly directives

The following assembly directives are allowed in Borland C++ inline-assembly statements:

`db` `dd` `dw` `extrn`

APPENDIX A

ANSI implementation-specific standards

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually. This chapter tells how Borland has chosen to define these implementation-specific standards. The section numbers refer to the February 1990 ANSI Standard. Remember that there are differences between C and C++; this appendix addresses C only.

2.1.1.3 How to identify a diagnostic.

When the compiler runs with the correct combination of options, any messages it issues beginning with the words *Fatal, Error,* or *Warning* are diagnostics in the sense that ANSI specifies. The options needed to ensure this interpretation are as follows:

Table A.1 Identifying diagnostics in C++

Option	Action
–A	Enable only ANSI keywords.
–C–	No nested comments allowed.
–i32	At least 32 significant characters in identifiers.
–p–	Use C calling conventions.
–w–	Turn off all warnings except the following.
–wbei	Turn on warning about inappropriate initializers.
–wbig	Turn on warning about constants being too large.
–wcpt	Turn on warning about nonportable pointer comparisons.
–wdcl	Turn on warning about declarations without type or storage class.
–wdup	Turn on warning about duplicate nonidentical macro definitions.
–wext	Turn on warning about variables declared both as external and as static.
–wfdt	Turn on warning about function definitions using a typedef.
–wrpt	Turn on warning about nonportable pointer conversion.
–wstu	Turn on warning about undefined structures.
–wsus	Turn on warning about suspicious pointer conversion.
–wucp	Turn on warning about mixing pointers to signed and unsigned char.
–wvrt	Turn on warning about void functions returning a value.

The following options cannot be used:

–ms!	SS must be the same as DS for small data models.
–mm!	SS must be the same as DS for small data models.
–mt!	SS must be the same as DS for small data models.
–zG*xx*	The BSS group name cannot be changed.
–zS*xx*	The data group name cannot be changed.

Other options not specifically mentioned here can be set to whatever you want.

2.1.2.2.1 The semantics of the arguments to main.

The value of *argv*[0] is a pointer to a null byte when the program is run on DOS versions prior to version 3.0. For DOS version 3.0 or later, *argv*[0] points to the program name.

The remaining *argv* strings point to each component of the command-line arguments. Whitespace separating arguments is removed, and each sequence of contiguous non-whitespace characters is treated as a single argument. Quoted strings are handled correctly (that is, as one string containing spaces).

2.1.2.3 What constitutes an interactive device.

An interactive device is any device that looks like the console.

2.2.1 The collation sequence of the execution character set.

The collation sequence for the execution character set uses the signed value of the character in ASCII.

2.2.1 Members of the source and execution character sets.

The source and execution character sets are the extended ASCII set supported by the IBM PC. Any character other than ^Z (Control-Z) can appear in string literals, character constants, or comments.

2.2.1.2 Multibyte characters.

Multibyte characters are supported in Borland C++.

2.2.2 The direction of printing.

Printing is from left-to-right, the normal direction for the PC.

2.2.4.2 The number of bits in a character in the execution character set.

There are 8 bits per character in the execution character set.

3.1.2 The number of significant initial characters in identifiers.

The first 32 characters are significant, although you can use a command-line option (**-i**) to change that number. Both internal and external identifiers use the same number of significant characters. (The number of significant characters in C++ identifiers is unlimited.)

3.1.2 Whether case distinctions are significant in external identifiers.

The compiler will normally force the linker to distinguish between uppercase and lowercase. You can use a command-line option (**-l-c**) to suppress the distinction.

3.1.2.5 The representations and sets of values of the various types of integers.

Type	16-bit minimum value	16-bit maximum value	32-bit minimum value	32-bit maximum value
signed char	–128	127	–128	127
unsigned char	0	255	0	255
signed short	–32,768	32,767	–32,768	32,767
unsigned short	0	65,535	0	65,535
signed int	–32,768	32,767	–2,147,483,648	–2,147,483,647
unsigned int	0	65,5350	0	4,294,967,295
signed long	–2,147,483,648	2,147,483,647	–2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295	0	4,294,967,295

All **char** types use one 8-bit byte for storage.

All **short** and **int** types use 2 bytes (in 16-bit programs).

All **short** and **int** types use 4 bytes (in 32-bit programs).

All **long** types use 4 bytes.

If alignment is requested (**-a**), all non**char** integer type objects will be aligned to even byte boundaries. If the requested alignment is **–a4**, the result is 4-byte alignment. Character types are never aligned.

3.1.2.5 The representations and sets of values of the various types of floating-point numbers.

The IEEE floating-point formats as used by the Intel 8087 are used for all Borland C++ floating-point types. The **float** type uses 32-bit IEEE real format. The **double** type uses 64-bit IEEE real format. The **long double** type uses 80-bit IEEE extended real format.

3.1.3.4 The mapping between source and execution character sets.

Any characters in string literals or character constants will remain unchanged in the executing program. The source and execution character sets are the same.

3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.

Wide characters are supported.

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.

Wide character constants are recognized.

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.

Character constants can contain one or two characters. If two characters are included, the first character occupies the low-order byte of the constant, and the second character occupies the high-order byte.

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.

These conversions are performed by simply truncating the high-order bits. Signed integers are stored as two's complement values, so the resulting number is interpreted as such a value. If the high-order bit of the smaller integer is nonzero, the value is interpreted as a negative value; otherwise, it is positive.

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.

The integer value is rounded to the nearest representable value. Thus, for example, the **long** value ($2^{31} - 1$) is converted to the **float** value 2^{31}. Ties are broken according to the rules of IEEE standard arithmetic.

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.

The value is rounded to the nearest representable value. Ties are broken according to the rules of IEEE standard arithmetic.

3.3 The results of bitwise operations on signed integers.

The bitwise operators apply to signed integers as if they were their corresponding unsigned types. The sign bit is treated as a normal data bit. The result is then interpreted as a normal two's complement signed integer.

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.

The access is allowed and will simply access the bits stored there. You'll need a detailed understanding of the bit encodings of floating-point values in order to understand how to access a floating-type member using a different member. If the member stored is shorter than the member used to access the value, the excess bits have the value they had before the short member was stored.

3.3.3.4 The type of integer required to hold the maximum size of an array.

For a normal array, the type is **unsigned int**, and for huge arrays the type is **signed long**.

3.3.4 The result of casting a pointer to an integer or vice versa.

When converting between integers and pointers of the same size, no bits are changed. When converting from a longer type to a shorter type, the high-order bits are truncated. When converting from a shorter integer type to a longer pointer type, the integer is first widened to an integer type the same size as the pointer type. Thus, signed integers will sign-extend to fill the new bytes. Similarly, smaller pointer types being converted to larger integer types will first be widened to a pointer type as wide as the integer type.

3.3.5 The sign of the remainder on integer division.

The sign of the remainder is negative when only one of the operands is negative. If neither or both operands are negative, the remainder is positive.

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t.

The type is **signed int** when the pointers are near (or the program is a 32-bit application), or **signed long** when the pointers are far or huge. The type of *ptrdiff_t* depends on the memory model in use. In small data models, the type is **int**. In large data models, the type is **long**.

3.3.7 The result of a right shift of a negative signed integral type.

A negative signed value is sign extended when right shifted.

3.5.1 The extent to which objects can actually be placed in registers by using the *register* storage-class specifier

Objects declared with any two-byte integer or pointer types can be placed in registers. The compiler will place any small auto objects into registers, but objects explicitly declared as *register* will take precedence. At least two and as many as six registers are available. The number of registers actually used depends on what registers are needed for temporary values in the function.

3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field.

Plain **int** bit fields are treated as **signed int** bit fields.

3.5.2.1 The order of allocation of bit fields within an int.

Bit fields are allocated from the low-order bit position to the high-order.

3.5.2.1 The padding and alignment of members of structures.

By default, no padding is used in structures. If you use the word alignment option (**-a**), structures are padded to even size, and any members that do not have character or character array type will be aligned to an even multiple offset.

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.

When alignment (**-a**) is not requested, bit fields can straddle word boundaries, but are never stored in more than two adjacent bytes.

3.5.2.2 The integer type chosen to represent the values of an enumeration type.

If all enumerators can fit in an **unsigned char**, that is the type chosen. Otherwise, the type is **signed int**.

3.5.3 What constitutes an access to an object that has volatile-qualified type.

Any reference to a volatile object will access the object. Whether accessing adjacent memory locations will also access an object depends on how the memory is constructed in the hardware. For special device memory, such as video display memory, it depends on how the device is constructed. For normal PC memory, volatile objects are used only for memory that might be accessed by asynchronous interrupts, so accessing adjacent objects has no effect.

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.

There is no specific limit on the number of declarators. The number of declarators allowed is fairly large, but when nested deeply within a set of blocks in a function, the number of declarators will be reduced. The number allowed at file level is at least 50.

3.6.4.2 The maximum number of case values in a switch statement.

There is no specific limit on the number of cases in a switch. As long as there is enough memory to hold the case information, the compiler will accept them.

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.

All character constants, even constants in conditional directives, use the same character set (execution). Single-character character constants will be negative if the character type is signed (default and **–K** not requested).

3.8.2 The method for locating includable source files.

For include file names given with angle brackets, if include directories are given in the command line, then the file is searched for in each of the include directories. Include directories are searched in this order: first, using directories specified on the command line, then using directories specified in TURBOC.CFG or BCC32.CFG. If no include directories are specified, then only the current directory is searched.

3.8.2 The support for quoted names for includable source files.

For quoted file names, the file is first searched for in the current directory. If not found, Borland C++ searches for the file as if it were in angle brackets.

3.8.2 The mapping of source file name character sequences.

Backslashes in include file names are treated as distinct characters, not as escape characters. Case differences are ignored for letters.

3.8.8 The definitions for __DATE__ and __TIME__ when they are unavailable.

The date and time are always available and will use the operating system date and time.

4.1.1 The decimal point character.

The decimal point character is a period (.).

4.1.5 The type of the sizeof operator, *size_t*.

The type *size_t* is **unsigned int**.

4.1.5 The null pointer constant to which the macro NULL expands.

For a 16-bit application, an integer or a long 0, depending on the memory model.

For 32-bit applications, NULL expands to an **int** zero or a **long** zero. Both are 32-bit signed numbers.

4.2 **The diagnostic printed by and the termination behavior of the assert function.**

The diagnostic message printed is "Assertion failed: *expression*, file *filename*, line *nn*", where *expression* is the asserted expression that failed, *filename* is the source file name, and *nn* is the line number where the assertion took place.

abort is called immediately after the assertion message is displayed.

4.3 **The implementation-defined aspects of character testing and case-mapping functions.**

None, other than what is mentioned in 4.3.1.

4.3.1 **The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions.**

First 128 ASCII characters for the default C locale. Otherwise, all 256 characters.

4.5.1 **The values returned by the mathematics functions on domain errors.**

An IEEE NAN (not a number).

4.5.1 **Whether the mathematics functions set the integer expression *errno* to the value of the macro ERANGE on underflow range errors.**

No, only for the other errors—domain, singularity, overflow, and total loss of precision.

4.5.6.4 **Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.**

No; `fmod(x,0)` returns 0.

4.7.1.1 **The set of signals for the signal function.**

SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM.

4.7.1.1 **The semantics for each signal recognized by the signal function.**

See the description of *signal* in the *Library Reference*.

4.7.1.1 **The default handling and the handling at program startup for each signal recognized by the signal function.**

See the description of *signal* in the *Library Reference*.

4.7.1.1 **If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed.**

The equivalent of *signal*(sig, SIG_DFL) is always executed.

4.7.1.1 **Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.**

No, it is not.

4.9.2 **Whether the last line of a text stream requires a terminating newline character.**

No, none is required.

4.9.2 **Whether space characters that are written out to a text stream immediately before a newline character appear when read in.**

Yes, they do.

4.9.2 **The number of null characters that may be appended to data written to a binary stream.**

None.

4.9.3 **Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.**

The file position indicator of an append-mode stream is initially placed at the beginning of the file. It is reset to the end of the file before each write.

4.9.3 **Whether a write on a text stream causes the associated file to be truncated beyond that point.**

A write of 0 bytes *might* or *might not* truncate the file, depending on how the file is buffered. It is safest to classify a zero-length write as having indeterminate behavior.

4.9.3 **The characteristics of file buffering.**

Files can be fully buffered, line buffered, or unbuffered. If a file is buffered, a default buffer of 512 bytes is created upon opening the file.

4.9.3 **Whether a zero-length file actually exists.**

Yes, it does.

4.9.3 **Whether the same file can be open multiple times.**

Yes, it can.

4.9.4.1 **The effect of the remove function on an open file.**

No special checking for an already open file is performed; the responsibility is left up to the programmer.

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.

rename will return a –1 and *errno* will be set to EEXIST.

4.9.6.1 The output for %p conversion in fprintf.

In near data models, four hex digits (*XXXX*). In far data models, four hex digits, colon, four hex digits (*XXXX*:*XXXX*). (For 16-bit programs.)

Eight hex digits (*XXXXXXXX*). (For 32-bit programs.)

4.9.6.2 The input for %p conversion in fscanf.

See 4.9.6.1.

4.9.6.2 The interpretation of a – (hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.

See the description of *scanf* in the *Library Reference*.

4.9.9.1 The value the macro errno is set to by the fgetpos or ftell function on failure.

EBADF Bad file number

4.9.10.4 The messages generated by perror.

Table A.2
Messages generated in both Win 16 and Win 32

- Arg list too big
- Attempted to remove current directory
- Bad address
- Bad file number
- Block device required
- Broken pipe
- Cross-device link
- Error 0
- Exec format error
- Executable file in use
- File already exists
- File too large
- Illegal seek
- Inappropriate I/O control operation
- Input/output error
- Interrupted function call
- Invalid access code
- Invalid argument
- Invalid data
- Invalid environment
- Invalid format
- Is a directory
- Math argument
- Memory arena trashed
- Name too long
- No child processes
- No more files
- No space left on device
- No such device
- No such device or address
- No such file or directory
- No such process
- Not a directory
- Not enough memory
- Not same device
- Operation not permitted
- Path not found
- Permission denied
- Possible deadlock
- Read-only file system
- Resource busy
- Resource temporarily unavailable
- Result too large
- Too many links

Table A.2: Messages generated in both Win 16 and Win 32 (continued)

Invalid function number
Invalid memory block address
Too many open files

Table A.3
Messages generated
only in Win 32

Bad address
Block device required
Broken pipe
Executable file in use
File too large
Illegal seek
Inappropriate I/O control operation
Input/output error
Interrupted function call
Is a directory
Name too long
No child processes
No space left on device
No such device or address
No such process
Not a directory
Operation not permitted
Possible deadlock
Read-only file system
Resource busy
Resource temporarily unavailable
Too many links

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.

calloc and *malloc* will ignore the request and return 0. *realloc* will free the block.

4.10.4.1 The behavior of the abort function with regard to open and temporary files.

The file buffers are not flushed and the files are not closed.

4.10.4.3 The status returned by exit if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.

Nothing special. The status is returned exactly as it is passed. The status is represented as a **signed char**.

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.

The environment strings are those defined in the operating system with the SET command. *putenv* can be used to change the strings for the duration of the current program, but the SET command must be used to change an environment string permanently.

4.10.4.5 The contents and mode of execution of the string by the system function.

The string is interpreted as an operating system command. COMSPEC is used or COMMAND.COM is executed (for 16-bit programs) or CMD.EXE (for 32-bit programs) and the argument string is passed as a command to execute. Any operating system built-in command, as well as batch files and executable programs, can be executed.

4.11.6.2 The contents of the error message strings returned by strerror.

See 4.9.10.4.

4.12.1 The local time zone and Daylight Saving Time.

Defined as local PC time and date.

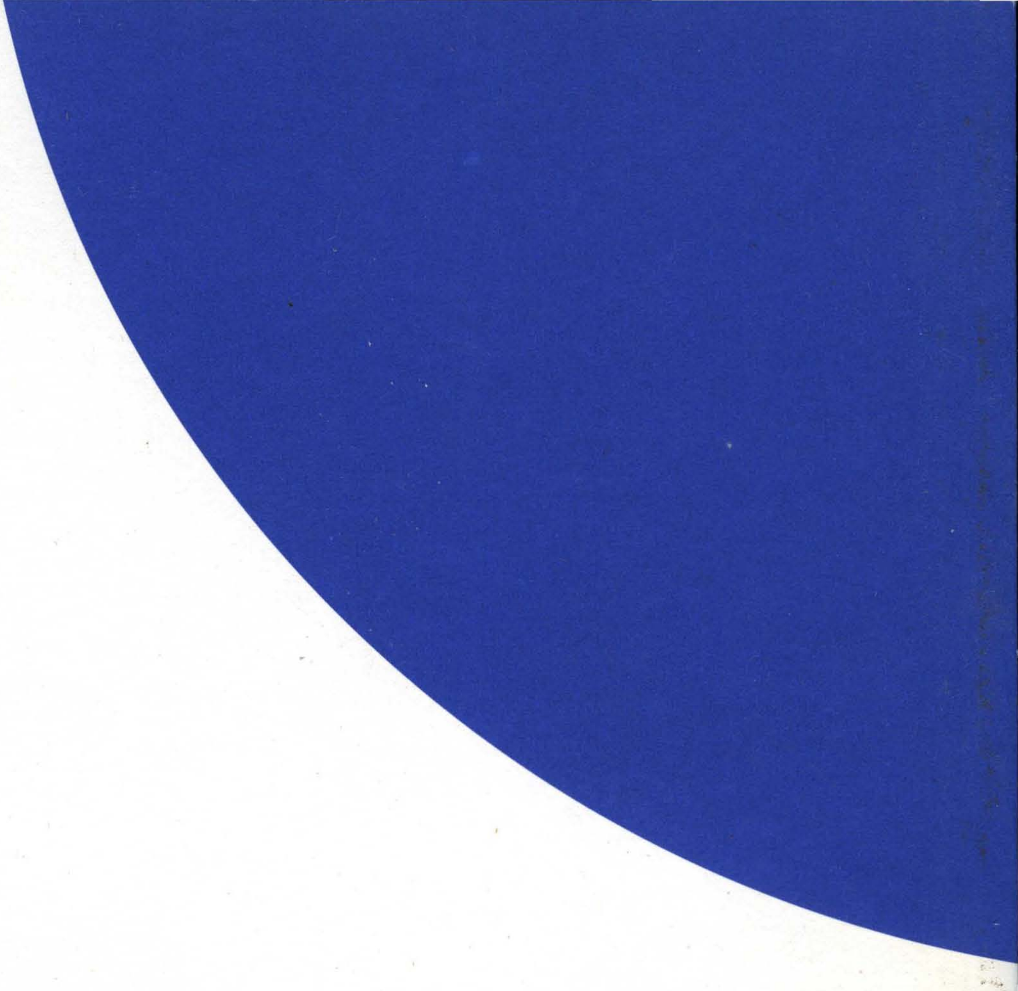
4.12.2.1 The era for clock.

Represented as clock ticks, with the origin being the beginning of the program execution.

4.12.3.5 The formats for date and time.

Borland C++ implements ANSI formats.

Index



Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1240WW21771 • BOR 6271