Turbo C®++ for Windows
Version 3.0

Programmer's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

Copyright © 1991 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Windows, as used in this manual, shall refer to Microsoft's implementation of a windows system.

R1

10 9 8 7 6 5 4 3 2 1

CONTENTS

Introduction 1
What's in this book 1
An introduction to the formal definitions . 2
Syntax and terminology 3

Chapter 1 Lexical elements 5
Whitespace 6
Line splicing with \ 6
Comments 7
C comments 7
Nested comments 7
C++ comments 8
Comment delimiters and whitespace . 8
Tokens 8
Keywords 9
Identifiers 10
Naming and length restrictions 10
Identifiers and case sensitivity 10
Uniqueness and scope 11
Constants 11
Integer constants 11
Decimal constants 11
Octal constants 12
Hexadecimal constants 13
Long and unsigned suffixes 13
Character constants 14
Escape sequences 14
Turbo C++ special two-character constants 15
Signed and unsigned char 15
Wide character constants 16
Floating-point constants 16
Floating-point constants—data types 16
Enumeration constants 17
String literals 18
Constants and internal representation 19
Constant expressions 20
Punctuators 21
Brackets 21
Parentheses 21
Braces 21
Comma 22
Semicolon 22
Colon 23
Ellipsis 23
Asterisk (pointer declaration) 23
Equal sign (initializer) 23
Pound sign (preprocessor directive) . 24

Chapter 2 Language structure 25
Declarations 25
Objects 25
Lvalues 26
Rvalues 27
Types and storage classes 27
Scope 27
Block scope 28
Function scope 28
Function prototype scope 28
File scope 28
Class scope (C++) 28
Scope and name spaces 28
Visibility 29
Duration 29
Static duration 29
Local duration 30
Dynamic duration 30
Translation units 31
Linkage 31
Name mangling 32

Declaration syntax 33
Tentative definitions 33
Possible declarations 34
External declarations and definitions . 36
Type specifiers 38
Type taxonomy 38
Type void 39
The fundamental types 39
Integral types 40
Floating-point types 41
Standard conversions 41
Special char, int, and enum conversions 42
Initialization 42
Arrays, structures, and unions 43
Simple declarations 44
Storage class specifiers 45
Use of storage class specifier auto .. 45
Use of storage class specifier extern . 45
Use of storage class specifier register 45
Use of storage class specifier static .. 46
Use of storage class specifier typedef 46
Modifiers 47
The const modifier 47
The interrupt function modifier 49
The volatile modifier 49
The cdecl and pascal modifiers 50
pascal 50
cdecl 50
The pointer modifiers 51
Function type modifiers 52
Complex declarations and declarators . 53
Pointers 54
Pointers to objects 55
Pointers to functions 55
Pointer declarations 56
Pointers and constants 57
Pointer arithmetic 58
Pointer conversions 59
C++ reference declarations 59
Arrays 59
Functions 60
Declarations and definitions 60
Declarations and prototypes 61
Definitions 63
Formal parameter declarations 64
Function calls and argument conversions 64
Structures 65
Untagged structures and typedefs 66
Structure member declarations 66
Structures and functions 67
Structure member access 67
Structure word alignment 69
Structure name spaces 69
Incomplete declarations 70
Bit fields 70
Unions 71
Anonymous unions (C++ only) 72
Union declarations 73
Enumerations 73
Expressions 75
Expressions and C++ 78
Evaluation order 78
Errors and overflows 79
Operator semantics 79
Operator descriptions 79
Unary operators 81
Binary operators 81
Additive operators 81
Multiplicative operators 81
Shift operators 81
Bitwise operators 81
Logical operators 81
Assignment operators 81
Relational operators 82
Equality operators 82
Component selection operators 82
Class-member operators 82
Conditional operator 82
Comma operator 82
Postfix and prefix operators 82
Array subscript operator [ ] 82
Function call operators ( ) 83
Structure/union member operator . (dot) 83

Structure/union pointer operator -> 83
Postfix increment operator ++ 84
Postfix decrement operator -- 84
Increment and decrement operators .. 84
Prefix increment operator 84
Prefix decrement operator 84
Unary operators 85
Address operator & 85
Indirection operator * 86
Unary plus operator + ... 86
Unary minus operator - 86
Bitwise complement operator ~ 86
Logical negation operator ! 86
The sizeof operator 87
Multiplicative operators 87
Additive operators 88
The addition operator + ... 88
The subtraction operator - 89
Bitwise shift operators 89
Bitwise shift operators (<< and >>) . 89
Relational operators 90
The less-than operator < 90
The greater-than operator > 91
The less-than or equal-to operator <= 91
The greater-than or equal-to operator >= 91
Equality operators 91
The equal-to operator == 91
The inequality operator != 92
Bitwise AND operator & 92
Bitwise exclusive OR operator ^ 93
Bitwise inclusive OR operator | 93
Logical AND operator && 93
Logical OR operator || 94
Conditional operator ? : 94
Assignment operators 95
The simple assignment operator = .. 95
The compound assignment operators 96
Comma operator 96
C++ operators 97
Statements 97
Blocks 98
Labeled statements 98
Expression statements 99
Selection statements 99
if statements 99
switch statements 100
Iteration statements 101
while statements 101
do while statements 101
for statements 102
Jump statements 103
break statements 103
continue statements 103
goto statements 103
return statements 104

Chapter 3 C++ specifics 105

Referencing 105
Simple references 106
Reference arguments 106
Scope access operator 108
The new and delete operators 108
Handling errors 109
The operator new with arrays 109
The operator delete with arrays 109
The ::operator new 110
Initializers with the new operator ... 110
Classes 111
Class names 111
Class types 111
Class name scope 112
Class objects 113
Class member list 113
Member functions 113
The keyword this 113
Inline functions 114
Static members 115
Member scope 116
Nested types 117
Member access control 118
Base and derived class access 120
Virtual base classes 122
Friends of classes 122
Constructors and destructors 124

Constructors 125
Constructor defaults 126
The copy constructor 127
Overloading constructors 127
Order of calling constructors 128
Class initialization 129
Destructors 132
When destructors are invoked 132
atexit, #pragma exit, and destructors . 133
exit and destructors 133
abort and destructors 133
Virtual destructors 134
Overloaded operators 135
Operator functions 136
Overloaded operators and inheritance 136
Overloading **new** and **delete** 137
Overloading unary operators 138
Overloading binary operators 139
Overloading the assignment operator = ... 139
Overloading the function call operator () 140
Overloading the subscript operator .. 140
Overloading the class member access operator 140
Virtual functions 140
Abstract classes 142
C++ scope 143
Class scope 144
Hiding 144
C++ scoping rules summary 144
Templates 145
Function templates 146
Overriding a template function ... 148
Implicit and explicit template functions 148
Class templates 149
Arguments 150
Angle brackets 150
Type-safe generic lists 151
Eliminating pointers 152

Chapter 4 The preprocessor 153

Null directive # 154
The #define and #undef directives 155
Simple #define macros 155
The #undef directive 156
The Define option 157
Keywords and protected words 157
Macros with parameters 157
File inclusion with #include 160
Header file search with <header_name> 161
Header file search with "header_name" 161
Conditional compilation 162
The #if, #elif, #else, and #endif conditional directives 162
The operator defined 163
The #ifdef and #ifndef conditional directives 163
The #line line control directive 164
The #error directive 165
The #pragma directive 165
#pragma argsused 166
#pragma exit and #pragma startup .. 166
#pragma hdrfile 167
#pragma hdrstop 167
#pragma saveregs 167
Predefined macros 168
__CDECL__ 168
__cplusplus 168
__DATE__ 169
__DLL__ 169
__FILE__ 169
__LINE__ 169
__MSDOS__ 169
__PASCAL__ 169
__STDC__ 170
__TCPLUSPLUS__ 170
__TEMPLATES__ 170
__TIME__ 170
__TURBOC__ 170
_Windows 170

Chapter 5 Using C++ streams 171
What is a stream? 171
The iostream library 172
  The streambuf class 172
  The ios class 172
Output 173
  Fundamental types 174
  Output formatting 174
  Manipulators 175
  Filling and padding 177
Input 177
I/O of user-defined types 178
Simple file I/O 179
String stream processing 180
Stream class reference 182
filebuf 182
  Member functions 182
fstream 183
  Member functions 184
fstreambase 184
  Member functions 184
ifstream 185
  Member functions 185
ios 185
  Data members 186
  Member functions 186
iostream 188
iostream_withassign 188
  Member functions 189
istream 189
  Member functions 189
istream_withassign 190
  Member functions 191
istrstream 191
ofstream 191
  Member functions 191
ostream 192
  Member functions 192
ostream_withassign 192
  Member functions 193
ostrstream 193
  Member functions 193
streambuf 193
  Member functions 194
strstreambase 196
  Member functions 196
strstreambuf 196
  Member functions 197
strstream 198
  Member function 198

Chapter 6 Math 199
Floating-point options 199
  Emulating the 80x87 chip 200
  Using 80x87 code 200
  No floating-point code 200
  Fast floating-point option 200
  Registers and the 80x87 201
  Disabling floating-point exceptions 201
Using complex math 202
  Using BCD math 203
    Converting BCD numbers 204
    Number of decimal digits 204

Chapter 7 BASM and inline assembly 207
  Inline assembly language 207
    BASM 207
    Inline syntax 208
    Opcodes 209
      String instructions 210
      Prefixes 211
      Jump instructions 211
      Assembly directives 211
    Inline assembly references to data and functions 211
      Inline assembly and register variables 212
      Inline assembly, offsets, and size overrides 212
    Using C structure members 212
    Using jump instructions and labels 213
  Interrupt functions 214
  Using low-level practices 215

Chapter 8 Building a Windows application 217
Compiling and linking within the IDE 218
  Understanding resource files 219

Understanding module definition files 219
Compiling and linking WHELLO 219
Using the project manager 220
Setting compile and link options 221
WinMain 221
Prologs and epilogs 222
Windows All Functions Exportable 222
Windows Explicit Functions Exported 223
Windows Smart Callbacks 223
Windows DLL All Functions Exportable 224
Windows DLL Explicit Functions Exported 224
The _export keyword 224
Prologs, epilogs, and exports: a summary 224
Memory models 226
Module definition files 226
A quick example 226
Linking for Windows 228
Linking in the IDE 228
Dynamic link libraries 228
Compiling and linking a DLL within the IDE 229
Import libraries 229
Creating an import library 229
Select an import library 230
From a DLL 230
From a module definition file 230
Creating the import library 230
Creating DLLs 230
LibMain and WEP 230
Pointers and memory 232
Static data in DLLs 233
C++ classes and pointers 233

Chapter 9 Error messages 235

Finding a message in this chapter 235
Types of messages 236
Compile-time messages 236
Help compiler messages 237
Run-time error messages 238
Librarian messages 239
Linker messages 239
Message explanations 240

Appendix A HC: The Windows Help compiler 309

Creating a Help system: The development cycle 309
How Help appears to the user 310
How Help appears to the help writer 311
How Help appears to the help programmer 312
Planning the Help system 312
Developing a plan 312
Defining the audience 312
Planning the contents 313
Planning the structure 314
Displaying context-sensitive Help topics 315
Determining the topic file structure 316
Choosing a file structure for your application 316
Designing Help topics 318
Layout of the Help text 318
Type fonts and sizes 319
Graphic images 320
Creating the Help topic files 321
Choosing an authoring tool 321
Structuring Help topic files 321
Coding Help topic files 322
Assigning build tags 323
Assigning context strings 324
Assigning titles 325
Assigning keywords 326
Creating multiple keyword tables 327
Assigning browse sequence numbers 328
Organizing browse sequences 328
Coding browse sequences 330
Creating cross-references between topics 330
Defining terms 331
Creating definition topics 331

Coding definitions 332
Inserting graphic images 332
Creating and capturing bitmaps ... 332
Placing bitmaps using a graphical word processor 333
Placing bitmaps by reference 333
Managing topic files 334
Keeping track of files and topics .. 335
Creating a help tracker 335
Building the Help file 337
Creating the Help project file 337
Specifying topic files 338
Specifying build tags 339
Specifying options 339
Specifying error reporting 340
Specifying build topics 340
Specifying the root directory 341
Specifying the index 342
Assigning a title to the Help system 342
Converting fonts 342
Changing font sizes 343
Multiple keyword tables 344
Compressing the file 344
Specifying alternate context strings .. 345
Mapping context-sensitive topics 346
Including bitmaps by reference 348
Compiling Help files 348
Using the Help Compiler 349
Programming the application to access Help 349
Calling WinHelp from an application 350
Getting context-sensitive Help 351
Shift+F1 support 352
F1 support 353
Getting help on items on the Help menu 355
Accessing additional keyword tables 355
Canceling Help 356
Help examples 357
The Helpex project file 359
Index 361

TABLES

1.1: All Turbo C++ keywords 9
1.2: Turbo C++ extensions to C 9
1.3: Keywords specific to C++ 9
1.4: Turbo C++ register pseudovariables . 10
1.5: Constants—formal definitions 12
1.6: Turbo C++ integer constants without L or U 13
1.7: Turbo C++ escape sequences 15
1.8: Turbo C++ floating constant sizes and ranges 17
1.9: Data types, sizes, and ranges 19
2.1: Turbo C++ declaration syntax 35
2.2: Turbo C++ declarator syntax 36
2.3: Turbo C++ class declarations (C++ only) 37
2.4: Declaring types 39
2.5: Integral types 40
2.6: Methods used in standard arithmetic conversions 42
2.7: Turbo C++ modifiers 47
2.8: Complex declarations 54
2.9: External function definitions 63
2.10: Associativity and precedence of Turbo C++ operators 76
2.11: Turbo C++ expressions 77
2.12: Bitwise operators truth table 93
2.13: Turbo C++ statements 98
4.1: Turbo C++ preprocessing directives syntax 154
5.1: Stream manipulators 176
5.2: File modes 180
7.1: Opcode mnemonics 210
7.2: String instructions 210
7.3: Jump instructions 211
8.1: Compiler options and the _export keyword 225
9.1: Compile-time message variables 237
9.2: Help message variables 238
9.3: Librarian message variables 239
9.4: Linker error message variables 239
A.1: Your application audience 313
A.2: Help design issues 318
A.3: Windows fonts 320
A.4: Help control codes 322
A.5: Restrictions of Help titles 326
A.6: Help keyword restrictions 327
A.7: Help project file sections 337
A.8: The Help [Options] options 339
A.9: WARNING levels 340
A.10: Build tag order of precedence 341
A.11: Build expression examples 341
A.12: *wCommand* values 350
A.13: *dwData* formats 351
A.14: MULTIKEYHELP structure formats 356

FIGURES

1.1: Internal representations of data types .20
5.1: Class **streambuf** and its derived classes 172
5.2: Class **ios** and its derived classes 173
8.1: Compiling and linking a Windows program 218
A.1: Helpex help window 311
A.2: Topic file 311
A.3: Example of a help hierarchy 314
A.4: Basic help file structure 316
A.5: Help file structure showing hypertext jumps 317
A.6: Help topic display showing bitmaps by reference 334
A.7: Help tracker text file example 336
A.8: Help tracker worksheet example 336
A.9: Word for Windows topic 357
A.10: Help topic display 358
A.11: Bitmap by reference in topic 358
A.12: Help topic display 359

INTRODUCTION

To get an overview of the Turbo C++ documentation set, start with the User's Guide. Read the introduction and Chapter 1 in that book for information on how to most effectively use the Turbo C++ manuals.

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 provides a language reference, and programming information on C++ streams, memory models, floating point, BASM, inline assembly, error messages, and Windows Help compiler.

Code examples have a **main** function. EasyWin makes all these examples work in Windows so you don't need **WinMain** and its complicated parameters.

Before you read the *Programmer's Guide,* read the *User's Guide* if

1. You have never programmed in any language.
2. You have programmed, but not in C, C++, or Windows, and you would like more introductory material.
3. You are looking for information on how to install Turbo C++.
4. You want general information on Turbo C++'s Integrated Development Environment (IDE) (including the editor), as well as information on how to use the project manager, convert from Microsoft C, or update DOS programs using EasyWin.

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

What's in this book

Chapters 1 through 4: Lexical elements, Language structure, C++ specifics, and **The preprocessor**, describe the Turbo 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

Turbo C++. Some overall information for Chapters 1 through 4 is included in the next section of this introduction.

Chapter 5: Using C++ streams tells you how to use the C++ version 2.1 stream library.

Chapter 6: Math covers floating point and BCD math.

Chapter 7: BASM and inline assembly tells how to write assembly language programs so they work well when called from Turbo C++ programs. It includes information on the built-in assembler in the IDE.

Chapter 8: Building a Windows application shows you how to get started programming in Windows.

Chapter 9: Error messages lists and explains all errors, fatal errors, and warnings, and suggests possible solutions. Run-time, compile-time, Librarian, Linker, and Help messages are alphabetized together.

Appendix A: HC: The Windows Help compiler introduces the Windows Help Compiler.

An introduction to the formal definitions

Chapters 1 through 4 constitute a formal description of the C and C++ languages as implemented in Turbo C++. Together, these chapters describe the Turbo C++ language; they provide a formal language definition, reference, and syntax for both the C++ and C aspects of Turbo C++. These chapters do not provide a language tutorial. We've used a modified Backus-Naur form notation to indicate syntax, supplemented where necessary by brief explanations and program examples. They are organized in this manner:

- Chapter 1, "Lexical elements," shows how the lexical tokens for Turbo C++ are categorized. Lexical elements is concerned with 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 Turbo C++. Language structure details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.
- Chapter 3, "C++ specifics," covers those aspects specific to C++.

- Chapter 4, "The preprocessor," covers the preprocessor, including macros, includes, and pragmas, as well as many other easy yet useful items.

Turbo C++ is a full implementation of AT&T's C++ version 2.1, 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++ 2.0. In addition to offering many new features and capabilities, C++ often veers from C by small or large amounts. We've made note of these differences throughout these chapters. All the Turbo C++ language features derived from C++ are discussed in greater detail in Chapter 3.

Turbo 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 Turbo C++ extension keywords as normal identifiers.

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 these chapters, 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 Turbo C++ lexical elements. It is concerned with the different categories of word-like units, known as *tokens*, recognized by a language. By contrast, language structure (covered in Chapter 2) details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

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

A Turbo C++ program starts life as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Turbo C++ editor). The basic program unit in Turbo C++ is the file. This usually corresponds to a named DOS 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 page 153). 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:

1. **int**
2. **i**
3. **;**
4. **float**
5. **f**
6. **;**

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process; in other words, they remain as part of the string:

```
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 literals," 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 Turbo C++, with an additional, optional extension permitting nested comments. You can mix and match either kind of comment 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 159 for a description of token pasting.

Turbo C++ does not support the nonportable *token pasting* strategy using **/**/**. Token pasting in Turbo 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 Turbo C++ */
#define VAR(i,j)  (i ## j)     /* Also OK */
```

In Turbo C++,

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

parses as

```
int   i  ;
```

to give the three tokens: **int i ;**

Nested comments

ANSI C doesn't allow nested comments. Attempting to comment out the preceding line with

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

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

```
i ; */
```

which would generate a syntax error.

By default, Turbo C++ won't allow nested comments, but you can override this with compiler options. You can enable nested comments via the Source Options dialog box (O | C | Source) in the IDE.

C++ comments

You can also use // to create comments in C code. This is specific to Turbo 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
... };
```

Comment 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

Turbo C++ recognizes six classes of tokens. The formal definition of a token is as follows:

token:
- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *operator*
- *punctuator*

Punctuators are also known as separators.

As the source code is parsed, 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*.

Keywords

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

Table 1.1
All Turbo C++ keywords

_asm	**_ds**	**_interrupt**	**short**
asm	**else**	**interrupt**	**signed**
auto	**enum**	**_loadds**	**sizeof**
break	**_es**	**long**	**_ss**
case	**_export**	**_near**	**static**
_cdecl	**extern**	**near**	**struct**
cdecl	**_far**	**new**	**switch**
char	**far**	**operator**	**template**
class	**float**	**_pascal**	**this**
const	**for**	**pascal**	**typedef**
continue	**friend**	**private**	**union**
_cs	**goto**	**protected**	**unsigned**
default	**_huge**	**public**	**virtual**
delete	**huge**	**register**	**void**
do	**if**	**return**	**volatile**
double	**inline**	**_saveregs**	**while**
	int	**_seg**	

Table 1.2
Turbo C++ extensions to C

_cdecl	**_es**	**interrupt**	**pascal**
cdecl	**_export**	**_loadds**	**_saveregs**
_cs	**_far**	**_near**	**_seg**
_ds	**far**	**near**	**_ss**
	huge	**_pascal**	

Table 1.3
Keywords specific to C++

asm	**inline**	**public**
class	**new**	**template**
delete	**operator**	**this**
friend	**private**	**virtual**
	protected	

Table 1.4 Turbo C++ register pseudovariables

_AH	**_BP**	**_CX**	**_DX**
_AL	**_BX**	**_DH**	**_ES**
_AX	**_CH**	**_DI**	**_FLAGS**
_BH	**_CL**	**_DL**	**_SI**
_BL	**_CS**	**_DS**	**_SP**
			_SS

Identifiers

The formal definition of an identifier is as follows:

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:

Identifiers in C++ programs are significant to any length.

1. The first character must be a letter or an underscore.
2. By default, Turbo C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu options, but not increased. Use Identifier Length in the Source Options dialog box (O | C | Source).

Identifiers and case sensitivity

Turbo 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, Turbo C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. By checking Case-sensitive Link in the Linker dialog box (Options | Linker | Settings), you can ensure that global identifiers are *case insensitive*. Under this regime, the globals *Sum* and *sum*

are considered identical, resulting in a possible "Duplicate symbol" warning during linking.

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 always legal for *different* name spaces regardless of scope. The rules are covered in the discussion on scope starting on page 27.

Constants

Constants are tokens representing fixed numeric or character values. Turbo C++ supports four classes of constants: floating point, integer, enumeration, and character.

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 1.5.

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 1.6. Note that the rules vary between decimal and nondecimal constants.

Decimal constants

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit will be 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 */
```

Table 1.5: 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...

Octal constants

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 will be truncated.

Hexadecimal constants

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

Long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces it 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 1.6
Turbo 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**
0100000 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 1.6 summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Character constants

A *character constant* is one or more characters enclosed in single quotes, such as `'A'`, `'='`, `'\n'`. In C, single character constants have data type **int**; they are represented internally with 16 bits, with the upper byte zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

Escape sequences

The backslash character (\) is used to introduce an *escape sequence*, allowing 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 Turbo 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 Turbo 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, Turbo C++ for Windows 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 may 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 1.7
Turbo C++ escape sequences

The \\ must be used to represent a real ASCII backslash, as used in DOS 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)
\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

Turbo C++ special two-character constants

Turbo C++ also supports two-character constants (for example, `'An'`, `'\n\t'`, and `'\007\007'`). 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. These constants are not portable to other C compilers.

Signed and unsigned char

In C, one-character constants, such as `'A'`, `'\t'`, and `'\007'`, are also represented as 16-bit **int** values. In this case, the low-order byte is *sign extended* into the high byte; that is, if the value is greater than 127 (base 10), the upper byte is set to –1 (=0xFF). This

can be disabled by declaring that the default **char** type is **unsigned** (choose Unsigned Characters in the Options | Compiler | Code Generation dialog box), which forces the high byte to be zero regardless of the value of the low byte.

Wide character constants

A character constant preceded by an *L* is a wide-character constant of data type **wchar_t** (an integral type defined in stddef.h). For example,

```
x = L 'A';
```

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.

Examples:

Constant	Value
23.45e6	23.45×10^6
.0	0
0.	0
1.	$1.0 \times 10^0 = 1.0$
−1.23	−1.23
2e-5	2.0×10^{-5}
3E+10	3.0×10^{10}
.09E34	0.09×10^{34}

Floating-point constants—data types

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
Turbo 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}

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 73 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 };
```

String literals

String literals, also known as string constants, 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 14). 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.

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>

int main()
{
   char    *p;

   p = "This is an example of how Turbo 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 Turbo 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");
```

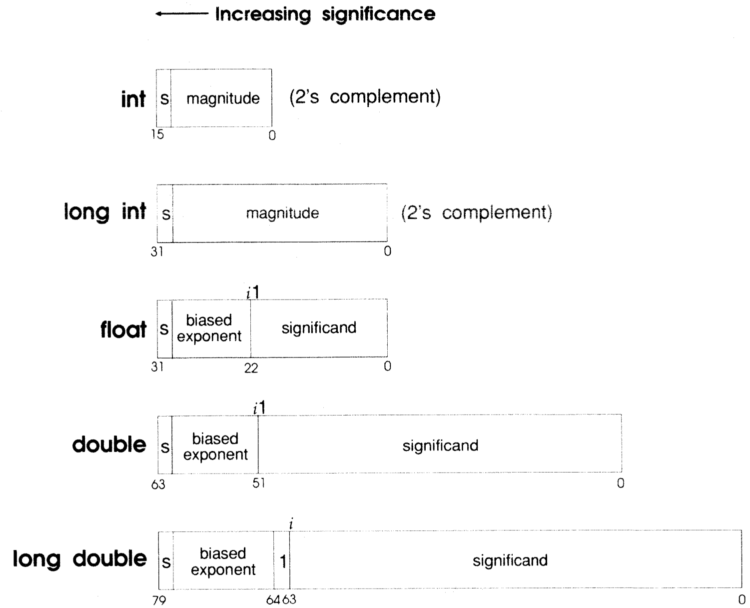
Constants and internal representation

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 Turbo C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of inner representations for the various data types. The next table lists the sizes and resulting ranges of the data types for Turbo C++; see page 39 for more information on these data types. Figure 1.1 shows how these types are represented internally.

Table 1.9: 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

Figure 1.1
Internal representations of data 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 Turbo 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 args */
```

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 on page 79.

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 **;** 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, Turbo C++ may 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;
...
switch (a) {
   case 1: puts("One");
           break;
   case 2: puts("Two");
           break;
...
default:   puts("None of the above!");
           break;
}
```

Labels are covered on page 98.

Ellipsis

Ellipsis (...) are three successive periods with no whitespace intervening. Ellipsis 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 a pointer to an int */
double ***double_ptr;     /* a pointer to a pointer to a pointer
                             to 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:

```
a = b + c;
ptr = farmalloc(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 153 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.

C H A P T E R

2

Language structure

This chapter provides a formal definition of Turbo C++'s language structure. It details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units. By contrast, lexical elements (described in Chapter 1) are concerned with the different categories of word-like units, known as tokens, recognized by a language.

Declarations

Scope is discussed starting on page 27; visibility on page 29; duration on page 29; and linkage on page 31.

This section briefly reviews concepts related to declarations: objects, types, storage classes, 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 not to be confused with 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

Turbo 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 Turbo C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Turbo 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 nonnull 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).

Types and storage classes

Associating identifiers with objects requires that each identifier has at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Turbo 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, as explained earlier, 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 special compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type; see page 87 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 scope

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 scope

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 scope

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 scope

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 scope (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 111 to 124.

Scope and name spaces

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

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

1. **goto** label names. These must be unique within the function in which they are declared.
2. 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.
3. 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.

4. 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.

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: Special C++ operators allow hidden identifiers to be accessed under certain conditions (see page 112).

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 duration

Objects with *static* duration are allocated memory 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.

Static duration must not be confused with file or global scope. An object can have static duration and local scope.

Local duration

An object with local duration also has local scope, since it does not exist outside of its enclosing block. The converse is not true: A local scope object can have static duration.

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 always must have local or function scope. The storage class specifier **auto** may be used when declaring local duration variables, but is usually redundant, since **auto** is the default for variables declared within a block.

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. Turbo 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 duration

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 36.

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 only within one file. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules are as follows:

1. 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.
2. 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.
3. If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
4. 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:

1. any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
2. function parameters
3. 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, simply 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 class, type, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known simply 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 simply 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 the following tables. The recursive nature of the declarator syntax allows complex declarators. We encourage the use of **typedefs** to improve legibility.

Table 2.1
Turbo C++ declaration syntax

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

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

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

storage-class-specifier:
    auto
    register
    static
    extern

fct-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*

For the following table, note that there are 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 47.

Table 2.2: Turbo C++ declarator syntax

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

init-declarator:
  declarator <*initializer*>

declarator:
  dname
  modifier-list
  ptr-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

ptr-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)
  ***typedef**-name*

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

abstract-declarator:
  ptr-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)

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

fct-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 "Translation units," page 31). For each identifier in a translation unit declared with internal linkage, there can be no more than one external definition.

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**), there must be exactly one external definition of that identifier somewhere in the entire program.

Turbo C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. For 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
```

The following table covers class declaration syntax. Page 105 covers C++ reference types (closely related to pointer types) in detail.

Table 2.3: Turbo C++ class declarations (C++ only)

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

class-head:
    class-key <identifier> <base-spec>
    class-key class-name <base-spec>

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-spec:
    : *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 <ptr-operator>

ctor-initializer:
    : *mem-initializer-list*

mem-initializer-list:
    mem-initializer
    mem-initializer , *mem-initializer-list*

mem-initializer:
    class name (*<argument-list>*)
    identifier (*<argument-list>*)

operator-function-name:
    operator *operator*

operator: one of
    new delete sizeof

+ − * / % ^
& | ~ ! = < >
+= −= *= /= %= ^=
&= |= << >> >>= <<=
== != <= >= && ||
++ −− , −>* −> ( )
[ ] .*

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++ there are some situations where a missing type specifier leads to syntactic ambiguity, so C++ practice uses the explicit entry of all **int** type specifiers.

Type taxonomy

There are four basic type categories: *void*, *scalar*, *function*, and *aggregate*. The scalar and aggregate types can be further divided as follows:

- Scalar: arithmetic, enumeration, pointer, and reference types (C++)
- Aggregate: array, structure, union, and class types (C++)

Types can also be divided into *fundamental* and *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

Note that `type& var`, `type &var`, *and* `type & var` *are all equivalent.*

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

And here's how you could declare derived types in a class:

```
class ct {              // class ct holds ptr to type plus a function
                        // taking a type parameter (C++)
   type *ptr;
   public:
   void func(type*);
}
```

Type void

C++ handles ***func()*** *in a special manner. See "Declarations and prototypes" on page 61 and code examples on page 62.*

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

- 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

signed *and* ***unsigned*** *are modifiers that can be applied to the integral types.*

The fundamental type specifiers are built from the following keywords:

char	**int**	**signed**
double	**long**	**unsigned**
float	**short**	

From these keywords, you can build the integral and floating-point types, which are together known as the *arithmetic* 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. The integral type specifiers are as follows, with synonyms listed on the same line:

Table 2.5
Integral types

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	

At most, one of **signed** and **unsigned** can be used with **char**, **short**, **int**, or **long**. If you use the keywords **signed** and **unsigned** on their own, they mean **signed int** and **unsigned int,** respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Turbo 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**.

At most, one of **long** and **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 insist that **short**, **int**, and **long** form a non-decreasing 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 Turbo C++, the types **int** and **short** are equivalent, both being 16 bits. **long** is a 32-bit object. The signed varieties are all stored in 2's complement format using the most significant bit (MSB) as a

sign bit: 0 for positive, 1 for negative (which explains the ranges shown in Table 1.9 on page 19). 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. Turbo C++ uses the IEEE floating-point formats.

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.

Table 1.9 on page 19 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, Turbo 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 Turbo C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Table 2.6. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers, **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
unsigned short	unsigned int	Same value
enum	int	Same value

Special char, int, and enum conversions

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

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**.

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 it has automatic storage duration, its value is indeterminate.

- to zero if it is of 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} <,>}

Rules governing initializers are

1. The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.
2. The item to be initialized must be an object type or an array of unknown size.
3. For C (not required for C++), all expressions must be constants if they appear in one of these places:
    a. in an initializer for an object that has static duration
    b. in an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed)
4. 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.
5. If there are fewer initializers in a brace-enclosed list than there are 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*, intended to count how many times each day of the week appears in a month (and assuming that each day will appear at least once), as follows:

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

Use these rules to initialize character arrays and wide character arrays:

1. 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).

2. 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. You can eliminate the braces, but you must follow certain rules, and it isn't recommended practice.

Simple declarations

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.

Storage class specifiers

A *storage class specifier*, or a type specifier, must be present in a declaration. The storage class specifiers can be one of the following:

auto	**register**	**typedef**
extern	**static**	

Use of storage class specifier 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.

Use of storage class specifier 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 32 for information on using **extern** to prevent name mangling when combining C and C++ code.

Use of storage class specifier register

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 that the variable should be allocated 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 Turbo C++ does a good job of placing variables in registers, it is rarely necessary to use the **register** keyword.

Turbo C++ lets you select register variable options from the Options | Compiler | Optimizations Options dialog box. If you check Automatic, Turbo C++ will try to allocate registers even if you have not used the **register** storage class specifiers.

Use of storage class specifier 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.

Use of storage class specifier 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. For 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 */
```

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 Turbo C++ are summarized in Table 2.7.

The const modifier

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:

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

```
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, since 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). For example,

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

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

Table 2.7
Turbo C++ modifiers

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

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

Table 2.7: Turbo C++ modifiers (continued)

Turbo 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.
pascal	Functions	Forces Pascal argument-passing convention. Affects Linker and link-time names.
pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
interrupt	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt handlers.
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,	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.
_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.

The interrupt function modifier

The **interrupt** modifier is specific to Turbo C++. **interrupt** functions are designed to be used with the 8086/8088 interrupt vectors. Turbo 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** definition:

```
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 volatile modifier

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

The **volatile** modifier indicates that the object may 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, since the value could (in theory) change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile  int  ticks;
interrupt   timer()
{
   ticks++;
}

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 doesn't change the value of *ticks*.

The cdecl and pascal modifiers

*Page 31 tells how to use **extern**, which allows C names to be referenced from a C++ program.*

Turbo 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.

In Turbo C++, all global identifiers are saved in their original case (lower, upper, or mixed) with an underscore (_) prepended to the front of the identifier, unless you have selected the Generate Underbars box in the Options | Compiler | Advanced Code Generation dialog box.

pascal

In Pascal, global identifiers are not saved in their original case, nor are underscores prepended to them. Turbo 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 "Function type modifiers," page 52, for more details.)

*The Calling Convention Pascal in the Options | Compiler | Entry | Exit Code dialog box causes all functions (and pointers to those functions) to be treated as if they were of type **pascal**.*

The **pascal** modifier is specific to Turbo 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, so 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 Windows API functions are **pascal** functions.

cdecl

Once you have compiled with Pascal calling convention turned on (IDE Options | Compiler | Entry/Exit Code), you may want to ensure that certain identifiers have their case preserved and keep

the underscore on the front, especially if they're C identifiers from another file. You can do so by declaring those identifiers to be **cdecl**. (This also has an effect on parameter passing for functions).

Like **pascal**, the **cdecl** modifier is specific to Turbo C++. It is used with functions and pointers to functions. It overrides the IDE Options | Compiler | Entry/Exit Code compiler directive 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();
void putnums(int i, int j, int k);

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 IDE Options | Compiler | Entry/Exit Code, all functions used from the run-time library will need to have **cdecl** declarations. If you look at the header files (such as stdio.h), you'll see that every function is explicitly defined as **cdecl** in anticipation of this.

The pointer modifiers

Turbo C++ has eight modifiers that affect the pointer declarator (*); that is, they modify pointers to data. These are **near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_seg**, and **_ss**.

C lets you compile 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, medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact, large), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes, when 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.

Function type 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 **_export**, **_loadds**, and **_saveregs** modifiers to modify functions.

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 Turbo C++.

A non-**interrupt** function 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. It's used in an executable (if you don't use smart callbacks) or in a DLL; see page 222 of Chapter 8 for details. The **_export** modifier has no significance for DOS programs.

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.

Complex declarations and declarators

See Table 2.1 on page 35 for the declarator syntax. The definition covers both identifier and function declarators.

Simple declarations have a list of comma-delimited identifiers following the optional storage class specifiers, type specifiers, and other modifiers.

A complex declaration uses a comma-delimited list of declarators following the various specifiers and modifiers. Within each declarator, there exists just one identifier, namely the identifier being declared. Each of the declarators in the list is associated with the leading storage class and type specifier.

The format of the declarator indicates how the declared *dname* 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 *dname* 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 declarations

```
int n, nao[], naf[3], *pn, *apn[], (*pan)[], &nr=n;
int f(void), *fnp(void), (*pfn)(void);
```

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 2.8: Complex declarations

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*)(), since the precedence of both the array declarator **[ ]** and the function declarator **( )** is higher than the pointer declarator *. The parentheses in *(*name*[]) are optional.

Pointers

See page 85 for a discussion 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 Turbo 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 51).

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.

Under C++, which has stronger type checking, a pointer to a function has type "pointer to function taking argument types ***type*** and returning ***type***." In fact, under C, a function defined with argument types will also have this narrower type. For example,

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

In C, this is a pointer to a function returning nothing. In C++, it's a pointer to a function taking no arguments and returning nothing. In this example,

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

func* is a pointer to a function taking an **int argument and returning nothing.

Pointer declarations

See page 39 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. Turbo 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;    /* Danger--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
```

Pointers and constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be assigned to. 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.

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.

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

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***."

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 105, "Referencing," 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. Thus, a two-dimensional array of five rows and seven columns called *alpha* is declared as

```
type alpha [5] [7];
```

In certain contexts, the first array declarator of a series may have no expression inside the brackets. Such an array is of indeter-

minate size. The contexts where this is legitimate are ones in which 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, nor does an array function parameter. As a special extension to ANSI C, Turbo 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. Turbo 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 31).

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 Turbo 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 original 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 : "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 only used 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 contains 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

The general syntax for external function definitions is given in the following table:

Table 2.9
External function definitions

file
    external-definition
    file external-definition

external-definition:
    function-definition
    declaration
    asm-statement

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, interrupt, near, far, huge, _export, _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 similar syntax to that of the declarators found in normal identifier declarations. Here are a few examples:

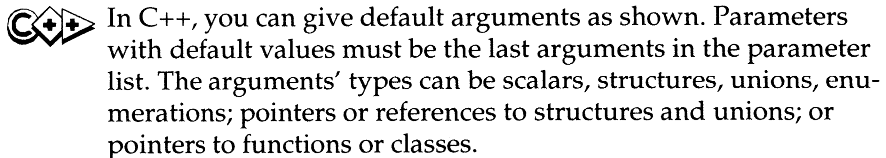
```
int func(void) {                   // no args

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, 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 enjoy 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 argument declarators.

Function calls and argument conversions

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

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

1. The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
2. A function may 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, Turbo C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in the section "Standard conversions," starting on page 41. When a function prototype is in scope, Turbo C++ converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Turbo 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 only be compatible 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, Turbo 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 42.

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 Turbo 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];
```

You don't usually 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 36.

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

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

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

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

   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.

2. 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 selection operators **.** and **->**. 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; 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 not 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 *a*1 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;
```

the object *s* occupies sufficient memory to hold a 2-byte integer, a 21-byte string, and an 8-byte double. The format of this object in memory is determined by the Turbo C++ word alignment option. With this option off (the default), *s* will be allocated 31 contiguous bytes.

If you turn on word alignment with the Options | Compiler | Code Generation dialog box, Turbo C++ pads the structure with bytes to ensure the structure is aligned as follows:

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.

With word alignment on, the structure would therefore have a byte added before the **double**, making a 32-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:
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, since 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:

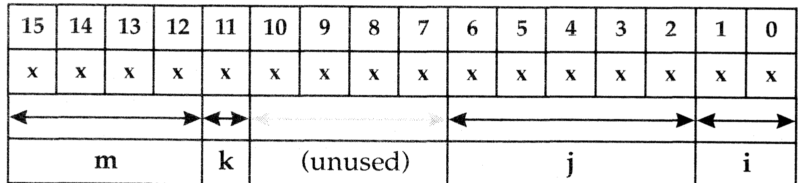
type-specifier <bitfield-id> : width;

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:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
m				k	(unused)				j					i	

Integer fields are stored in 2'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, since 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, since 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:

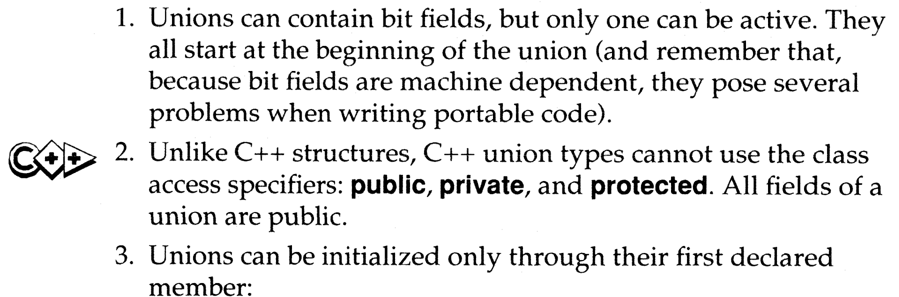
```
union { member-list };
```

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 may not have external linkage.

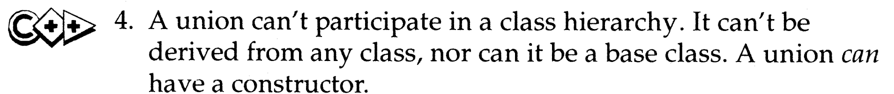
Union declarations

The general declaration syntax for unions is pretty much the same as that for structures. Differences are



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

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



4. 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.

Turbo C++ is free to store enumerators in a single byte when Treat enums as ints is unchecked (O | C | Code Generation). 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 17 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.

Expressions

Table 2.11 shows how identifiers and operators are combined to form grammatically legal "phrases."

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Table 2.11, 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.)

The standard conversions are detailed in Table 2.6 on page 42.

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules which 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 Turbo C++ (see "Evaluation order" on page 78).

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

We've summarized the precedence and associativity of the operators in Table 2.10. The grammar in Table 2.11 on page 77

completely defines the precedence and associativity of the operators.

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

Where there are duplicates of operators 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 2.10
Associativity and precedence of Turbo C++ operators

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

Table 2.11: Turbo 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
    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 --

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 136. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the relational operator **==** might be defined in the 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 rules for operators and conversions discussed in this section may not apply to expressions in C++.

Evaluation order

The order in which Turbo 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. 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.

Turbo 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 online Help.

We've summarized the precedence and associativity of the operators in Table 2.10. During the evaluation of an expression, Turbo 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 Turbo 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 operator) and **?:** (conditional operator) (and you also can't overload the C++ operators **::** and **.***).

If an operator is overloaded, the discussion may not be true for it anymore. Table 2.11 on page 77 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. Turbo 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 covered starting on page 135.

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, we confine our discussion to the standard operators of Turbo C++.

After defining the standard operators, we discuss data types and declarations, and explain how these affect the actions of each operator. From there we can proceed with the syntax for building expressions from operators, punctuators, and objects.

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

operator: one of

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

The operators # and ## are used only by the preprocessor (see page 153).

And the following operators 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++)

Unary operators

&	Address operator
*	Indirection operator
+	Unary plus
–	Unary minus
~	Bitwise complement (1's complement)
!	Logical negation
++	Prefix: preincrement; Postfix: postincrement
– –	Prefix: predecrement; Postfix: postdecrement

Binary operators

Additive operators

+	Binary plus (addition)
–	Binary minus (subtraction)

Multiplicative operators

*	Multiply
/	Divide
%	Remainder

Shift operators

<<	Shift left
>>	Shift right

Bitwise operators

&	Bitwise AND
^	Bitwise XOR (exclusive OR)
|	Bitwise inclusive OR

Logical operators

&&	Logical AND
||	Logical OR

Assignment operators

=	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 operators	**<**	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
Equality operators	**==**	Equal to
	!=	Not equal to
Component selection operators	**.**	Direct component selector
	–>	Indirect component selector
Class-member operators	**::**	Scope access/resolution
	.*	Dereference pointer to class member
	–>*	Dereference pointer to class member
Conditional operator	**a ? x : y**	"if *a* then *x*; else *y*"
Comma operator	**,**	Evaluate; e.g., `a, b, c;` from left to right

The operator functions, as well as their syntax, precedences, and associativities, are covered starting on page 75.

Postfix and prefix operators

The six postfix operators **[ ] ( ) . –> ++** and **– –** are used to build postfix expressions as shown in the expressions syntax table (Table 2.11). The increment and decrement operators (**++** and **– –**) are also prefix and unary operators; they are discussed starting on page 84.

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 "Function calls and argument conversions," page 64, for more on function calls.

Structure/union member operator . (dot)

In the expression

postfix-expression **.** *identifier*

lvalues are defined on page 26.

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 **.** and **->** for structures are given on page 67.

Structure/union pointer operator ->

In the expression

postfix-expression **->** *identifier*

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 and only if the postfix expression is an lvalue.

Postfix 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 modifiable lvalue (see page 26 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.

Postfix 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.

Increment and decrement operators

The first two unary operators are **++** and **– –**. These are also postfix and prefix operators, so they are discussed here. The remaining six unary operators are covered following this discussion.

Prefix increment operator

In the expression

++ *unary-expression*

the unary expression is the operand; it must be of scalar type and must be a modifiable lvalue. The prefix increment operator is 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.

Prefix decrement operator

The prefix decrement, also known as the *predecrement*, operator has the following syntax:

– – *unary-expression*

It follows the same rules as the prefix increment operator, except that the operand is decremented by 1 before the whole expression is evaluated.

Unary operators

The six unary operators (aside from **++** and **– –**) are **&** ***** **+** **–** **~** and **!**. The syntax is

unary-operator cast-expression

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

Address operator &

*The symbol **&** is also used in C++ to specify reference types; see page 105.*

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 ***type***, the result is of type pointer to ***type***.

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:

```
type t1 = 1, t2 = 2;
type *ptr = &t1;     // initialized pointer
*ptr = t2;           // same effect as t1 = t2
```

type `*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 ***type***," where ***type*** is any type. The result of the indirection is of type ***type***. 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:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Unary 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.

Unary 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 op-

erand is nonzero; 1 if the operand is zero. The expression *!E* is equivalent to `(0 == E)`.

The sizeof operator

There are two distinct uses of the **sizeof** operator:

> **sizeof** *unary-expression*
> **sizeof** (*type-name*)

How much space is set aside 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 Turbo 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 size).

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 41).

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.

1. 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*.
2. 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*.

Additive operators

There are two additive operators: **+** and **–**. The syntax is

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

The addition operator +

The legal operand types for *op1* **+** *op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. *op1* is of integral type, and *op2* is of pointer to object type.
3. *op2* is of integral type, and *op1* is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In cases 2 and 3, the rules of pointer arithmetic apply. (Pointer arithmetic is covered on page 58.)

The subtraction operator –

The legal operand types for *op1* – *op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. 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***.

3. *op1* is of pointer to object type, and *op2* is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

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*

Bitwise shift operators (<< 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, since the sign bit may 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*

The less-than operator <

In the expression *E1* **<** *E2*, the operands must conform to one of the following sets of conditions:

Qualified names are defined on page 117.

1. Both *E1* and *E2* are of arithmetic type.
2. Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible object types.
3. Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible incomplete types.

In case 1, 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 cases 2 and 3, where *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 under "Pointer arithmetic" on page 58. 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.

The greater-than operator >

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 defined for the less-than operator. The same operand rules and restrictions also apply.

The less-than or equal-to operator <=

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 defined for the less-than operator. The same operand rules and restrictions also apply.

The greater-than or equal-to operator >=

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 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.
➡ However, **==** and **!=** have a lower precedence than the relational operators **< >**, **<=**, 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*

The equal-to operator ==

In the expression *E1* == *E2*, the operands must conform to one of the following sets of conditions:

1. Both *E1* and *E2* are of arithmetic type.
2. Both *E1* and *E2* are pointers to qualified or unqualified versions of compatible types.

3. 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**.
4. 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 case 1, 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 case 2, *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 case 4, 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**).

The inequality operator !=

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.

Bitwise AND operator &

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 2.12.

Table 2.12
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

Bitwise exclusive OR operator ^

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 2.12.

Bitwise inclusive OR operator |

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 2.12.

Logical AND operator &&

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**, 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.

Logical OR operator ||

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 operator ? :

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 sets of rules:

1. Both of arithmetic type
2. Both of compatible structure or union types
3. Both of void type
4. Both of type pointer to qualified or unqualified versions of compatible types
5. One operand of pointer type, the other a null pointer constant
6. One operand of type pointer to an object or incomplete type, the other 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 case 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 case 2, the type of the result is the structure or union type of *E2* and *E3*.

In case 3, the result is of type **void**.

In cases 4 and 5, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.

In case 6, the type of the result is that of the nonpointer-to-void operand.

Assignment operators

There are eleven assignment operators. The = operator is the simple assignment operator; the other ten are known as compound assignment operators.

The syntax is

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

assignment-operator: one of

```
=    *=   /=   %=   +=   -=
<<=  >>=  &=   ^=   |=
```

The simple assignment operator =

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 sets of rules:

1. *E1* is of qualified or unqualified arithmetic type and *E2* is of arithmetic type.

2. *E1* has a qualified or unqualified version of a structure or union type compatible with the type of *E2*.
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.
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.
5. *E1* is a pointer and *E2* is a null pointer constant.

The compound assignment operators

The compound assignments *op***=**, where *op* can be any one of the ten 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

ambiguity with the commas used in function argument and 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

See page 108 for information on the scope access operator (::).

The operators specific to C++ are **::** **.*** **–>***. 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 (no, that isn't a typo; 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,

```
(ptr2object->*ptr2memberfunc) (10);
```

calls the member function denoted by *ptr2memberfunc* for the object pointed to be *ptr2object*.

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. Table 2.13 on page 98 lays out the syntax for statements:

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.

Table 2.13: Turbo C++ statements

```
statement:
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
    asm-statement
    declaration (C++ specific)

asm-statement:
    asm tokens newline
    asm tokens;
    asm { tokens; <tokens;>=
          <tokens;>
          }

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> ;

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> ;
```

Labeled statements

A statement can be labeled in the following ways:

1. *label-identifier* **:** *statement*

   The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and enjoy function scope. In C++ you can label both declaration and non-declaration statements.

2. **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>;

Turbo 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.

A special case is the *null statement*, consisting of a single semicolon (;). The null statement does nothing. It is nevertheless useful in situations where the Turbo 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:

The parentheses around cond-expression are essential.

if (*cond-expression*) *t-st* <**else** *f-st*>

The *cond-expression* must be of scalar type. The expression is evaluated. If the value is zero (or null for pointer types), we say that the *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, say, Pascal, Turbo 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**.

Iteration statements

Iteration statements let you loop a set of statements. There are three forms of iteration in Turbo C++: **while**, **do**, and **for** loops.

while statements

The general format for this statement is

The parentheses are essential.

while (*cond-exp*) *t-st*

The loop statement, *t-st*, will be 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 99). 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 statements

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. (But don't try this at home. Such stunts are performed by trained professionals.)
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 < j; ++i)
{
   if (i ...) ...              // ok to refer to i here
}
if (i...)                      // illegal; i 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. Since iteration and **switch** statements can be intermixed and nested to any depth, take care to 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 "Labeled statements,"page 98), 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 "falls through," 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.

C H A P T E R

3

C++ specifics

C++ is basically a superset of C. This means that, 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 peculiar to C++. Some situations need special care. For example, the same function **func** declared twice in C with different argument types will invoke a duplicated name error. Under C++, however, **func** will be interpreted as an overloaded function—whether this is legal or not will depend 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. We first review these aspects of C++ that can be used independently of classes, then get into the specifics of classes and class mechanisms.

Referencing

Pointer referencing and dereferencing is discussed on page 85.

While in C you pass arguments only by value, in C++ you can pass arguments by value or by reference. C++ reference types, which are 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 that 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 possibly dangerous side effects) to pass `&x`, especially if *x* is a large data structure. Passing *x* directly by value involves the wasteful copying of the data structure.

Compare the three implementations of the function **treble**:

Implementation 1

```
int treble_1(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 that `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, Turbo 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 access operator

The scope access (or resolution) operator :: (two semicolons) lets you access a global (or file duration) name even if it is hidden by a local redeclaration of that name (see page 27 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 in the **malloc** and **free** families (see online Help).

A simplified syntax is

> *pointer-to-name* = **new** *name* *<name-initializer>*;
> **delete** *pointer-to-name*;

name can be of any type except "function returning..." (however, pointers to functions are allowed).

new tries to create an object of type *name* by allocating (if possible) **sizeof**(*name*) bytes in *free store* (also called the heap). The storage duration of the new object is from the point of creation until the operator **delete** kills it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the new object. A null pointer indicates a failure (such as insufficient or fragmented heap memory). As with **malloc**, you need to test for null before trying to access the new object (unless you use a new-handler; see the following section for details). However, unlike **malloc, new** calculates the size of *name* without the need for an explicit **sizeof**

operator. Further, the pointer returned is of the correct type, "pointer to *name*," without the need for explicit casting.

***new**, being a keyword, doesn't need a prototype.*

```
name *nameptr;       // name is any nonfunction type
   ...
if (!(nameptr = new name)) {
   errmsg("Insufficient memory for name");
   exit (1);
}
// use *nameptr to initialize new name object
...
delete nameptr;   // destroy name and deallocate sizeof(name) bytes
```

Handling errors

You can define a function that will be called if the **new** operator fails (returns 0). To tell the **new** operator about the new-handler function, call **set_new_handler** and supply a pointer to the new-handler. The prototype for **set_new_handler** is as follows (from new.h):

```
void (*set_new_handler( void (*)() ))();
```

set_new_handler returns the old new-handler, and changes the function **_new_handler** so that it, in turn, points to the new-handler that you define.

The operator new with arrays

If *name* is an array, the pointer returned by **new** points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the left-most 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 may be a variable, all following dimensions must be constants.

The operator delete with arrays

You must use the syntax "delete [] expr" when deleting an array. In 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. In order to allow 2.0 code to compile, Turbo C++ issues a warning and simply 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()
```

With Turbo C++, the [] is actually only required when the array element is a class with a destructor. But it is good programming practice to always tell the compiler that an array is being deleted.

The ::operator new

When used with non-class objects, **new** works by calling a standard library routine, the global **::operator new**. With class objects of type *name*, a specific operator called *name*::**operator new** can be defined. **new** applied to class *name* objects invokes the appropriate *name*::**operator new** if present; otherwise, the standard **::operator new** is used.

Initializers with the new operator

The optional initializer is another advantage **new** has over **malloc** (although **calloc** does clear its allocations to zero). 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 126). The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

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 class-name <: *base-list*> { <*member-list*> }

class-key is one of **class**, **struct**, or **union**.

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 120, "Base and derived class access"). 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 118, "Member access control").

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that may affect which functions can access which members.

Class names

class-name is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted (see "Untagged structures and typedefs," page 66.)

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, with some tricks peculiar to classes. 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 only be referred to 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 "Structure member declarations," page 66). 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 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 a mechanism whereby 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) and 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 that they differ in argument type or number of arguments.

The keyword this

Nonstatic member functions operate on the class type object with which they are called. 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 *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.

The keyword **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.

Turbo 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 is a request (or hint) to the compiler that you would welcome an inline expansion. As with the **register** storage class specifier, the compiler may or may not take the hint!

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:

```
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 the section on member scope starting on page 116.

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 object in its 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)
}
```

Since a static member function can be called with no particular object in mind, it has no **this** pointer. A consequence of this is that a static member function 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 obey the usual class member access rules, except they can be initialized.

```
class X {
   ...
   static int x;
   ...
};

int X::x = 1;
```

The main 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 on page 114 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 it 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 also "The keyword **this**," page 113). 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. A key point is 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

In C++ 2.1, even tag or typedef names declared inside a class lexically belong to the scope of that class. Such names can in general be accessed only 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 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. 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();
};

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
```

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 "Friends of classes," page 122).

public The member can be used by any function.

private The member can be used only by member functions and friends of the class in which it is declared.

protected Same as for **private**, but 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,

```
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;
};
```

The access specifiers can be listed and grouped in any convenient sequence. You can save a little typing effort by declaring all the private members together, and so on.

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** may 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.

In both cases, note carefully that **private** members of a base class are, and remain, inaccessible to member functions of the derived class *unless* **friend** declarations are explicitly declared in the base class granting access. For example,

```
class X : A {                // default for class is private A
...
}
/* class X is derived from class A */

class Y : B, public C {      // override default for C
...
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */

struct S : D {               // default for struct is public D
...                          /* struct S is derived from D */
}
struct T : private D, E {    // override default for D
                             // E is public by default
...
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations in 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 that, although not a member function of **X**, has 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 "Inline functions," page 114). 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**. However, friendship *is* 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.

1. They do not have return value declarations (not even **void**).
2. They cannot be inherited, though a derived class can call the base class's constructors and destructors.
3. Constructors, like most C++ functions, can have default arguments or use member initialization lists.
4. Destructors can be **virtual**, but constructors cannot.
5. You can't take their addresses.

   ```
   int main(void)
   {
      ...
      void *ptr = base::base;    // illegal
     ...
   }
   ```

6. Constructors and destructors can be generated by Turbo 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.

7. 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
...
}
```

8. The compiler automatically calls constructors and destructors when defining and destroying objects.
9. Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
10. An object with a constructor or destructor cannot be used as a member of a union.

If a 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 of which it is a member. 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*. We discuss the default constructor next; the description of the copy constructor starts on page 127.

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, Turbo C++ generates a default constructor. On a declaration such as `X x`, the default constructor creates the object *x*.

Important! 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.

Take care to avoid ambiguity in calling constructors. In the following case, the two default constructors could become ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};

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`: `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 copying a class object, typically when you declare with initialization by another class object: `X x = y`. Turbo C++ generates a copy constructor for class **X** if one is needed and none is defined in class **X**.

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; }
};

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
```

```
...
   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 non-virtual 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 non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first so that the virtual base class may be properly constructed. The code

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // non-virtual 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 only constructed once
base2();       // non-virtual base of virtual base level2
               // must be called to construct level2
level2();      // virtual base class
base2();       // non-virtual base of level1
level1();      // other non-virtual base
toplevel();
```

In the event that a class hierarchy contains multiple instances of a virtual base class, that base class is only constructed once. If, however, there exist both virtual and non-virtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each non-virtual 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 of the type of 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&);
};

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 }
};
```

Or it can use an initializer list prior to the function body:

```
class X
{
   int a, b;
public:
   X(int i, int j) : a(i), b(j) {}
};
```

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. The values of the derived class can't be changed and then have an affect on the base class's creation.

```
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.

```
class X
{
public:
   ~X();  // destructor for class X
};
```

If a destructor is not explicitly defined for a class, the compiler will generate one.

When destructors are invoked

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 128).

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 Turbo C++ program in these regards 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 only necessary for objects allocated a specific address through calls to **new**.

```
class X {
...
   ~X();
...
};

void* operator new(size_t size, void *ptr)
{
   return ptr;
}

char buffer[sizeof(X)];

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 the following declarations

```
color *palette[3];

palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;
```

will 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, in the event that no destructors were 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**.

Overloaded operators

C++ lets you redefine the action 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.

All the operators on page 79 can be overloaded except for

. .* :: ?:

The preprocessing symbols # and ## also cannot be overloaded.

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.

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 (Table 2.10 on page 76) applying to normal operator use. Consider the class *complex*:

*This class was invented for illustrative purposes only. It isn't the same as the class **complex** in the run-time library.*

```
class complex {
   double real, imag;                        // private by default
public:
   ...
   complex() { real = imag = 0; }            // inline constructor
   complex(double r, double i = 0) {         // another one
        real = r; imag = i;
   }
   ...
}
```

We could easily devise a function for adding complex numbers, say,

```
complex AddComplex(complex c1, complex c2);
```

but it would be more natural to be able to write:

```
complex c1(0,1), c2(1,0), c3
c3 = c1 + c2;
```

than

```
c3 = AddComplex(c1, c2);
```

The operator **+** is easily overloaded by including the following declaration in the class *complex*:

```
friend complex operator +(complex c1, complex c2);
```

and defining it (possibly inline) as:

```
complex operator +(complex c1, complex c2)
{
   return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

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 the next sections), 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 "Overloading the assignment operator =" on page 139), 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** may possibly be further overloaded for **Y**.

Overloading **new** and **delete**

*The type **size_t** is defined in stdlib.h*

The operators **new** and **delete** can be overloaded to provide alternative free storage (heap) memory-management routines. A user-defined operator **new** must return a **void*** and must have a **size_t** as its first argument. A 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. 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 */ }
   ...
};
```

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**) will invoke the matching user-defined **X::operator new()** and **X::operator delete()**, respectively.

The **X::operator new** 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** and **delete** operators can still be used within the scope of **X**, either explicitly with the global scope operator (**::operator new** 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**. The size of a derived class object may well differ from that of the base class.

Overloading unary operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a non-member 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.

Under C++ 2.0, an overloaded operator**++** or — is used for both prefix and postfix uses of the operator. For example:

```
struct foo
{
   operator::();
   operator--();
}
   x;

void func()
{
   x++;   // calls x.operator++()
   ++x;   // calls x.operator++()

   x--;   // calls x.operator--()
   --x;   // calls x.operator--()
}
```

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. For example add the following lines to the previous code:

```
operator++(int);
operator--(int);
```

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. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in 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()
```

Overloading binary operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member 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.

Overloading the 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
}
```

Overloading the 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*).

Overloading the 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*).

Overloading the 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.

Virtual functions

Virtual functions can only be member 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, then redefine it in any derived class, even if the number and type of arguments are the same. The 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. The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available.

With virtual functions, you cannot change just the function type. It is illegal, therefore, to redefine a virtual function so that it differs only in the return type. If two functions with the same name have different arguments, C++ considers them different, and the virtual function mechanism is ignored.

If a base class **B** contains a virtual function **vf**, and class **D**, derived from **B**, contains a function **vf** of the same type, then if **vf** is called for an object *d* or **D**, the call made is `D::vf`, even if the access is via a pointer or reference to **B**. For example,

```
struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
};
class D : public B {
   virtual void vf1(); // virtual specifier is legal but redundant
   void vf2(int);      // not virtual, since it's using a different
                       // arg list
   char vf3();         // Illegal: return-type-only change!
   void f();
};

void extf()
{
   D d;        // declare a D object
   B* bp = &d; // standard conversion from D* to B*
   bp->vf1();  // calls D::vf1
   bp->vf2();  // call B::vf2 since D's vf2 has different args
   bp->f();    // calls B::f (not virtual)
}
```

The overriding function **vf1** in **D** is automatically virtual. The **virtual** specifier *can* be used with an overriding function declaration in the derived class, but its use is redundant.

The interpretation of a virtual function call depends on the type of the object for which it is called; with non-virtual function calls, the

interpretation depends only on the type of the pointer or reference denoting the object for which it is called.

Virtual functions must be members of some class, but they cannot be static members. A virtual function can be a **friend** of another class.

A virtual function in a base class, like all member functions of a base class, must be defined or, if not defined, declared *pure*:

```
class B {
virtual void vf(int) = 0;      // = 0 means 'pure'
```

In a class derived from such a base class, each pure function must be defined or redeclared as pure (see the next section, "Abstract classes").

If a virtual function is defined in the base it need not necessarily be redefined in the derived class. Calls will simply call the base function.

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 using the pure-specifier.

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: attempted creation of 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 may 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 only be used 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:

1. The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
2. If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.

3. 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.
4. 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.
5. Any name not covered so far that is used in a static member function must be declared in the block in which it occurs 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.
6. Any name not covered so far that is used in a nonstatic member function of class **X** must be declared in the block in which it occurs 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.
7. 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 non-defining 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.
8. A constructor initializer (see *ctor-initializer* in the class declarator syntax, Table 2.3 on page 37) 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*, allow you to construct a family of related functions or classes.

In this section, we'll introduce the basic concept then some specific points.

Syntax:

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,

```
int max(int x, int y)
{
   return (x > y) ? x : y;
}

long max(long x, long y)
{
   return (x > y) ? x : y;
}
   ⋮
```

followed by other versions of **max**.

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*)** 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 Foo
{
 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,* if you like). A specific instantiation of a function template is called a *template function.* You can override the generation of a template function for a specific type with a non-template 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 agruments. If it doesn't there is no way of deducing the actual values for the unused template arguments when the function is called.

Implicit and explicit template functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ingores 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, this function, on the other hand, will participate fully in overload resolution. For example:

```
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)
}
```

Class templates

A class template (also called a *generic class* or *class generator*) allows you to 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 *type* 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;
};

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 may be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol **Vector** must be 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 \BORLANDC\CLASSLIB\INCLUDE subdirectory.

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 { ... };
```

Non-type template arguments such as *size* can have default arguments. The value supplied for a non-type 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

Take care 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.

Type-safe generic lists

In general, when you need to write lots of nearly identical things, think templates. The problems with the following class definition, a generic list class,

```
class GList
{
 public:
   void insert( void * );
   void *peek();
    // ...
};
```

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 you'll get back out. 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 only hold 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**. Now to do the same thing 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, once again, templates to the rescue:

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. And there's no code generated by the type conversions from each wrapper class so 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 a big benefit 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, since templates allow them to be inlined.

C H A P T E R

4

The preprocessor

Although Turbo C++ uses an integrated single-pass compiler , it is useful to retain the terminology associated with earlier multipass compilers.

With a multipass compiler, a first pass of the source text would pull in any include files, test for any conditional-compilation directives, expand any macros, and produce an intermediate file for further compiler passes.

The following discussion on preprocessor directives, their syntax and semantics, therefore, applies to the preprocessor functionality built into the Turbo C++ compiler.

The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them.

Turbo C++ 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 Turbo C++'s preprocessor directives is given in the next table.

Table 4.1: Turbo 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
    #         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 situ* 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 may 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* turns out to be 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 will result in a warning unless the new definition is *exactly* the same, token-by-token definition as the existing one. The preferred strategy where definitions may 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 is not 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 Define option

Identifiers can be defined, but not explicitly undefined, from the Defines input box in the Code Generation | Options dialog box (under O | C | Code Generation) (see Chapter 1, "IDE basics," in the *User's Guide*).

Keywords and protected words

It is legal but ill-advised to use Turbo C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers may *not* 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 that 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 *place holder*.

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 and potential pitfalls (see page 160).

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:

1. **Nested parentheses and commas:** The *actual_arg_list* may 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))); */
```

2. **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)
   ```

   then the call `VAR(x,6)` would expand to (*x*6). This replaces the older (nonportable) method of using `(i/**/j)`.

3. **Converting to strings with #:** The # symbol can be placed in front of a formal macro argument in order 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);
   ```

4. **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 */
   ```

5. **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 forms:

The angle brackets are real tokens, not metasymbols that imply that header_name is optional.

#include *<header_name>*
#include *"header_name"*
#include *macro_identifier*

The third variant assumes that neither **<** nor " appears as the first non-whitespace character following **#include**; further, it assumes that a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the *<header_name>* or *"header_name"* formats.

The first and second variant 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 DOS file name with an extension (traditionally .h for header) and optional path name and path delimiters.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the com-

piler "sees" the enlarged text. The placement of the **#include** may 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>* variant 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 is not located in any of the default directories, an error message is issued.

Header file search with "header_name"

The *"header_name"* variant 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 is not 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:\TURBOC\INCLUDE\MYSTUFF.H"
/* Note: Single backslashes OK here; within a C statement you would
   need "C:\\TURBOC\\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:\TURBOC\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

Turbo 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 Turbo 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 interlude) 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 whether combinations of identifiers are defined or not. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) or **defined** identifier (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). So the directive

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use **defined** repeatedly in a complex expression following the **#if** directive, such as

```
#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 is

#line *integer_constant* <*"filename"*>

indicating 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 then compile and run TEMP.C, you'll get the output shown here:

```
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 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 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 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, Turbo C++ can define whatever directives it desires 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.

Turbo C++ supports the following **#pragma** directives:

- #pragma argsused
- #pragma exit
- #pragma hdrfile
- #pragma hdrstop
- #pragma saveregs
- #pragma startup

#pragma argsused

The **argsused** pragma is only allowed 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 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 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**; in other words, it should be declared as

```
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,

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

*Note that the function name used in **pragma startup** or **exit** must be defined (or declared) before the pragma line is reached.*

```
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)
   _InitWinCrt();
#endif

   printf("This is main.\n");
}
```

#pragma hdrfile

This directive sets the name of the file in which to store precompiled headers. The default file name is TCDEF.SYM. The syntax is

#pragma hdrfile "*filename*.SYM"

See Appendix D, "Precompiled headers" in the User's Guide for more details.

If you aren't using precompiled headers, this directive has no effect. You can use Precompiled Header (O | C | Code Generation) to change the name of the file used to store precompiled headers.

#pragma hdrstop

This directive terminates the list of header files that are eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers. (See Appendix D in the *User's Guide* for more on precompiled headers.)

#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.

Predefined macros

Turbo C++ predefines certain global identifiers, each of which is discussed in this section. Except for __cplusplus and _Windows, each of these starts and ends with two underscore characters (__). These macros are also known as *manifest constants*.

__CDECL__

This macro is specific to Borland's C and C++ family of compilers. It signals that the Options | Compiler | Entry/Exit | Calling is not set to Pascal. Set to the integer constant 1 if calling was not used; otherwise, undefined.

The following four symbols are defined based on the memory model chosen at compile time.

__COMPACT__	__MEDIUM__
__LARGE__	__SMALL__

Only one 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.

__cplusplus

This macro is defined as 1 if in C++ mode; it's undefined otherwise. This allows you to 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 to be 1 if you compile a module using the Windows DLL All Functions Exportable radio button (O | C | C | Entry/Exit Code) 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 to be 1, through the **#line** directive can affect this. See page 164 for information on the **#line** directive.

_ _MSDOS_ _

This macro is specific to Borland's C/C++ family of compilers. It provides the integer constant 1 for all compilations.

_ _PASCAL_ _

This macro is specific to Borland's C and C++ family of compilers. It signals that the Pascal calling convention (O | C | C | Exit/Entry)

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 with ANSI radio button (Source Options); otherwise, the macro is undefined.

__TCPLUSPLUS__

This macro is specific to Borland's C and C++ family of compilers. It is only defined for C++ compilation. If you've selected C++ compilation, it is defined as 0x0300, a hexadecimal constant. This numeric value will increase in later releases.

__TEMPLATES__

This macro is specific to Borland's C and C++ family of compilers. It is defined as 1 for C++ files (meaning that Turbo C++ supports templates); it's undefined otherwise.

__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).

__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.

_Windows

Indicates that Windows-specific code is being generated. This macro is defined by default for Turbo C++.

C H A P T E R

5

Using C++ streams

This chapter is divided into two sections: a brief, practical overview of using C++ stream I/O, and a reference section to the C++ stream class library.

Stream input/output in C++ (commonly referred to as *iostreams*, or merely *streams*) provide all the functionality to the **stdio** library in C. iostreams are used to convert typed objects into readable text, and vice versa. Streams may also read and write binary data. The C++ language allows you to 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 memory buffers (iostream.h), files (fstream.h), and strings (strstrea.h) as sources or sinks (or both).

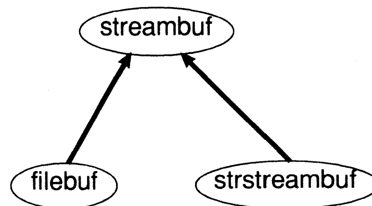
The iostream library

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 physical devices. **streambuf** provides general methods for buffering and handling streams when little or no formatting is required. **streambuf** is a useful base class employed by other parts of the iostream library, though you can also derive classes from it for your own functions and libraries. The classes **filebuf** and **strstreambuf** are derived from **streambuf**.

Figure 5.1
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 5.2. 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

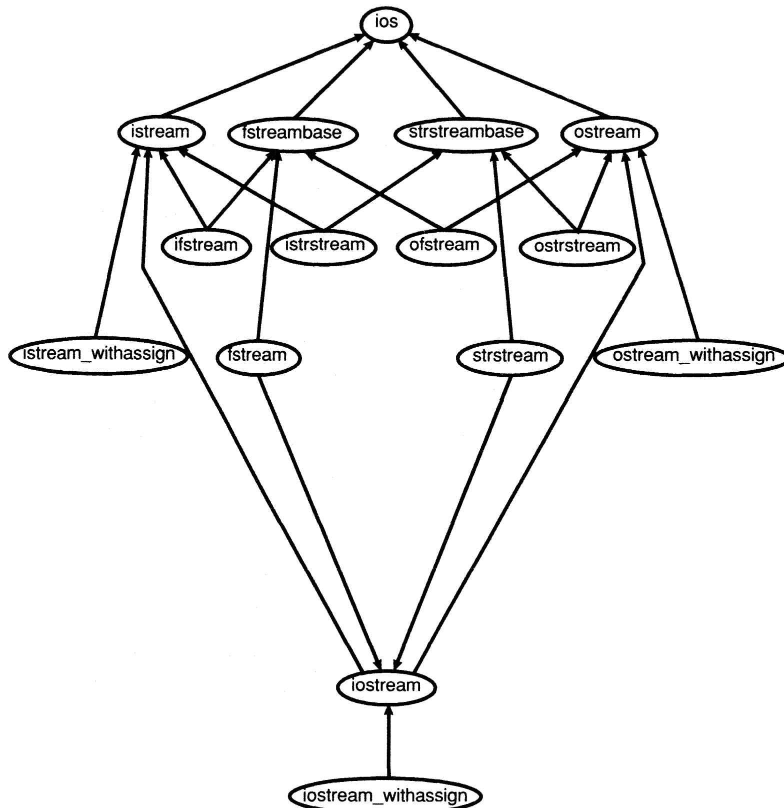
classes in the **ios**-based tree use a **streambuf** (or a **filebuf** or **strstreambuf**, which are special cases of a **streambuf**) as its source and/or sink.

C++ programs start with four predefined open streams, declared as objects of **withassign** classes as follows:

```
extern istream_withassign cin;    // Corresponds to stdin
extern ostream_withassign cout;   // Corresponds to stdout
extern ostream_withassign cerr;   // Corresponds to stderr
extern ostream_withassign clog;   // A buffered cerr
```

Figure 5.2
Class **ios** and its derived classes

*By accepted practice, the arrows point **from** the derived class **to** the base class.*



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 for which it is invoked. 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 the **ostream** class (page 192) for other output functions.

Output formatting

Formatting for both input and output is determined by various *format state* flags contained in the class **ios**. The format flags are as follows:

```
public:
enum {
   skipws,      // skip whitespace on input
   left,        // left-adjust output
   right,       // right-adjust output
   internal,    // pad after sign or base indicator
   dec,         // decimal conversion
   oct,         // octal conversion
   hex,         // hexadecimal conversion
   showbase,    // show base indicator on output
   showpoint,   // show decimal point (floating-point output)
   uppercase,   // uppercase hex output
   showpos,     // show '+' with positive integers
   scientific,  // suffix floating-point numbers with exponential (E)
                   notation on output
   fixed,       // use fixed decimal point for floating-point numbers
   unitbuf,     // flush all streams after insertion
   stdio,       // flush stdout, stderr after insertion
};
```

These flags are read and set with the **flags**, **setf**, and **unsetf** member functions (see class **ios** starting on page 185).

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 5.1
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.
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 the 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-justification (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 on page 186 for a list of masks used by **setf**.

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**, beginning on page 189).

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 allow you to 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, chosen from the following table:

Table 5.2
File modes

Mode bit	Action
ios::app	Append data—always write at end of file.
ios::ate	Seek to end of file upon original open.
ios::in	Open for input (default for **ifstreams**).
ios::out	Open for output (default for **ofstreams**).
ios::binary	Open file in binary mode.
ios::trunc	Discard contents if file exists (default if **ios::out** is specified and neither **ios::ate** nor **ios::app** is specified).
ios::nocreate	If file does not exist, **open** fails.
ios::noreplace	If file exists, **open** for output fails unless **ate** or **app** is set.

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** functions are available for the class **istrstream** (*i*nput *str*ing *stream*); likewise 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];
   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.

Stream class reference

The stream class library in C++ consists of several classes. This reference presents some of the most useful details of these classes, in alphabetical organization. The following cross-reference lists tell which classes belong to which header files.

iostream.h: **ios**, **iostream**, **iostream_withassign**, **istream**, **istream_withassign**, **ostream**, **ostream_withassign**, **streambuf**.

fstream.h: **filebuf**, **fstream**, **fstreambase**, **ifstream**, **ofstream**.

strstrea.h: **istrstream**, **ostrstream**, **strstream**, **strstreambase**, **strstreambuf**.

filebuf <fstream.h>

Specializes **streambuf** to handle files.

constructor `filebuf();`

Makes a **filebuf** that isn't attached to a file.

constructor `filebuf(int fd);`

Makes a **filebuf** attached to a file as specified by file descriptor *fd*.

constructor `filebuf(int fd, char *, int n);`

Makes a **filebuf** attached to a file and uses a specified *n*-character buffer.

Member functions

attach `filebuf* attach(int)`

Attaches this closed **filebuf** to opened file descriptor.

close `filebuf* close()`

Flushes and closes the file. Returns 0 on error.

fd Returns the file descriptor or EOF.

is_open `int is_open();`

Returns nonzero if the file is open.

open
```
filebuf* open(const char*, int mode, int prot = filebuf::openprot);
```
Opens the given file and connects to it.

overflow
```
virtual int overflow(int = EOF);
```
Flushes a buffer to its destination. Every derived class should define the actions to be taken.

seekoff
```
virtual streampos seekoff(streamoff, ios::seek_dir, int);
```
Moves the file pointer relative to the current position.

setbuf
```
virtual streambuf* setbuf(char*, int);
```
Specifies a buffer for this **filebuf**.

sync
```
virtual int sync();
```
Establishes consistency between internal data structures and the external stream representation.

underflow
```
virtual int underflow();
```
Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

fstream <fstream.h>

This stream class, derived from **fstreambase** and **iostream**, provides for simultaneous input and output on a **filebuf**.

constructor
```
fstream();
```
Makes an **fstream** that isn't attached to a file.

constructor
```
fstream(const char*, int, int = filebuf::openprot);
```
Makes an **fstream**, opens a file, and connects to it.

constructor
```
fstream(int);
```
Makes an **fstream**, connects to an open file descriptor.

constructor
```
fstream(int, char*, int);
```
Makes an **fstream** connected to an open file and uses a specified buffer.

Member functions

open
```
void open(const char*, int, int = filebuf::openprot);
```
Opens a file for an **fstream**.

rdbuf
```
filebuf* rdbuf();
```
Returns the filebuf used.

fstreambase <fstream.h>

This stream class, derived from **ios**, provides operations common to file streams. It serves as a base for **fstream**, **ifstream**, and **ofstream**.

constructor
```
fstreambase();
```
Makes an **fstreambase** that isn't attached to a file.

constructor
```
fstreambase(const char*, int, int = filebuf::openprot);
```
Makes an **fstreambase**, opens a file, and connects to it.

constructor
```
fstreambase(int);
```
Makes an **fstreambase**, connects to an open file descriptor.

constructor
```
fstreambase(int, char*, int);
```
Makes an **fstreambase** connected to an open file and uses a specified buffer.

Member functions

attach
```
void attach(int);
```
Connects to an open file descriptor.

close
```
void close();
```
Closes the associated **filebuf** and file.

open
```
void open(const char*, int, int = filebuf::openprot);
```
Opens a file for an **fstreambase**.

rdbuf
```
filebuf* rdbuf();
```

Returns the filebuf used.

setbuf

```
void setbuf(char*, int);
```

Uses a specified buffer.

ifstream <fstream.h>

This stream class, derived from **fstreambase** and **istream**, provides input operations on a **filebuf**.

constructor

```
ifstream();
```

Makes an **ifstream** that isn't attached to a file.

constructor

```
ifstream(const char*, int = ios::in, int = filebuf::openprot);
```

Makes an **ifstream**, opens an input file in protected mode, and connects to it. The existing file contents are preserved; new writes are appended.

constructor

```
ifstream(int);
```

Makes an **ifstream**, connects to an open file descriptor.

constructor

```
ifstream(int fd, char *, int);
```

Makes an **ifstream** connected to an open file and uses a specified buffer.

Member functions

open

```
void open(const char*, int, int = filebuf::openprot);
```

Opens a file for an **ifstream**.

rdbuf

```
filebuf* rdbuf();
```

Returns the filebuf used.

ios <iostream.h>

Provides operations common to both, input and output. Its derived classes (**istream**, **ostream**, **iostream**) specialize I/O with high-level formatting operations. The **ios** class is a base for **istream**, **ostream**, **fstreambase**, and **strstreambase**.

constructor `ios();` **protected**

Constructs an **ios** object that has no corresponding **streambuf**.

constructor `ios(streambuf *);`

Associates a given **streambuf** with the stream.

Data members

`static const long`	`adjustfield;`	// left | right | internal	
`static const long`	`basefield;`	// dec | oct | hex	
`static const long`	`floatfield;`	// scientific | fixed	
`streambuf`	`*bp;`	// the associated streambuf	**protected**
`int`	`x_fill;`	// padding character on // output	**protected**
`long`	`x_flags;`	// formatting flag bits	**protected**
`int`	`x_precision;`	// floating-point precision on // output	**protected**
`int`	`state;`	// current state of the // streambuf	**protected**
`ostream`	`*x_tie;`	// the tied ostream, if any	**protected**
`int`	`x_width;`	// field width on output	**protected**

Member functions

bad `int bad();`

Nonzero if error occurred.

bitalloc `static long bitalloc();`

Acquires a new flag bit set. The return value may be used to set, clear, and test the flag. This is for user-defined formatting flags.

clear `void clear(int = 0);`

Sets the stream state to the given value.

eof `int eof();`

Nonzero on end of file.

fail `int fail();`

Nonzero if an operation failed.

fill `char fill()`

Returns the current fill character.

fill `char fill(char);`

Resets the fill character; returns the previous one.

flags `long flags();`

Returns the current format flags.

flags `long flags(long);`

Sets the format flags to be identical to the given **long**; returns previous flags. Use **flags(0)** to set the default format.

good `int good();`

Nonzero if no state bits set (that is, no errors appeared).

init `void init(streambuf *);` **protected**

Provides the actual initialization.

precision `int precision();`

Returns the current floating-point precision.

precision `int precision(int);`

Sets the floating-point precision; returns previous setting.

rdbuf `streambuf* rdbuf();`

Returns a pointer to this stream's assigned streambuf.

rdstate `int rdstate();`

Returns the stream state.

setf `long setf(long);`

Sets the flags corresponding to those marked in the given **long**; returns previous settings.

setf `long setf(long _setbits, long _field);`

The bits corresponding to those marked in *_field* are cleared, and then reset to be those marked in *_setbits*.

setstate `protected:void setstate(int);`

Sets all status bits.

sync_with_stdio
```
static void sync_with_stdio();
```
Mixes stdio files and iostreams. This should not be used for new code.

tie
```
ostream* tie();
```
Returns the *tied stream*, or zero if none. Tied streams are those that are connected such that when one is used, the other is affected. For example, **cin** and **cout** are tied; when **cin** is used, it flushes **cout** first.

tie
```
ostream* tie(ostream*);
```
Ties another stream to this one and returns the previously tied stream, if any. When an input stream has characters to be consumed, or if an output stream needs more characters, the tied stream is first flushed automatically. By default, **cin**, **cerr** and **clog** are tied to **cout**.

unsetf
```
long unsetf(long);
```
Clears the bits corresponding to those marked in the given **long**; returns previous settings.

width
```
int width();
```
Returns the current width setting.

width
```
int width(int);
```
Sets the width as given; returns the previous width.

xalloc
```
static int xalloc();
```
Returns an array index of previously unused words that can be used as user-defined formatting flags.

iostream <iostream.h>

This class, derived from **istream** and **ostream**, is simply a mixture of its base classes, allowing both input and output on a stream. It is a base for **fstream** and **strstream**.

constructor
```
iostream(streambuf *);
```
Associates a given **streambuf** with the stream.

iostream_withassign <iostream.h>

This class is an **iostream** with an added assignment operator.

constructor

```
iostream_withassign();
```

Default constructor (calls **iostream**'s constructor).

Member functions

None (although the **=** operator is overloaded).

istream <iostream.h>

Provides formatted and unformatted input from a **streambuf**. The **>>** operator is overloaded for all fundamental types, as explained in the narrative at the beginning of the chapter. This **ios** class is a base for **ifstream, iostream, istrstream**, and **istream_withassign**.

constructor

```
istream(streambuf *);
```

Associates a given **streambuf** with the stream.

Member functions

gcount

```
int gcount();
```

Returns the number of characters last extracted.

get

```
int get();
```

Extracts the next character or EOF.

get

```
istream& get(signed char*, int len, char = '\n');
istream& get(unsigned char*, int len, char = '\n');
```

Extracts characters into the given **char *** until the delimiter (third parameter) or end-of-file is encountered, or until (*len* – 1) bytes have been read. A terminating null is always placed in the output string; the delimiter never is. Fails only if no characters were extracted.

get

```
istream& get(signed char&);
istream& get(unsigned char&);
```

Extracts a single character into the given character reference.

get

```
istream& get(streambuf&, char = '\n');
```

Extracts characters into the given **streambuf** until the delimiter is encountered.

getline

```
istream& getline(signed char *buffer, int, char = '\n');
istream& getline(unsigned char *buffer, int, char = '\n');
```

Same as **get**, except the delimiter is also extracted. The delimiter is not copied to *buffer*.

ignore

```
istream& ignore(int n = 1, int delim = EOF);
```

Causes up to *n* characters in the input stream to be skipped; stops if **delim** is encountered.

peek

```
int peek();
```

Returns next char without extraction.

putback

```
istream& putback(char);
```

Pushes back a character into the stream.

read

```
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Extracts a given number of characters into an array. Use **gcount()** for the number of characters actually extracted if an error occurred.

seekg

```
istream& seekg(long);
```

Moves to an absolute position (as returned from **tellg**).

seekg

```
istream& seekg(long, seek_dir);
```

Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**

tellg

```
long tellg();
```

Returns the current stream position.

istream_withassign <iostream.h>

This class is an **istream** with an added assignment operator.

constructor

```
istream_withassign();
```

Default constructor (calls **istream**'s constructor).

Member functions

None (although the = operator is overloaded).

istrstream <strstrea.h>

Provides input operations on a **strstreambuf**. This class is derived from **strstreambase** and **istream**.

constructor

```
istrstream(const char *);
```

Makes an **istrstream** with a specified string (a null character is never extracted).

constructor

```
istrstream(const char *, int n);
```

Makes an **istrstream** using up to *n* bytes of a specified string.

ofstream <fstream.h>

Provides input operations on a **filebuf**. This class is derived from **fstreambase** and **ostream**.

constructor

```
ofstream();
```

Makes an **ofstream** that isn't attached to a file.

constructor

```
ofstream(const char*, int = ios::out, int = filebuf::openprot);
```

Makes an **ofstream**, opens a file, and connects to it.

constructor

```
ofstream(int);
```

Makes an **ofstream**, connects to an open file descriptor.

constructor

```
ofstream(int fd, char*, int);
```

Makes an **ofstream** connected to an open file and uses a specified buffer.

Member functions

open

```
void open(const char*, int, int = filebuf::openprot);
```

Opens a file for an **ofstream**.

rdbuf

```
filebuf* rdbuf();
```

Returns the filebuf used.

ostream <iostream.h>

Provides formatted and unformatted output to a **streambuf**. The **<<** operator is overloaded for all fundamental types, as explained on page 174. This **ios**-based class is a base for **iostream**, **ofstream**, **ostrstream**, and **ostream_withassign**.

constructor

```
ostream(streambuf *);
```

Associates a given **streambuf** with the stream.

Member functions

flush

```
ostream& flush();
```

Flushes the stream.

put

```
ostream& put(char);
```

Inserts the character.

seekp

```
ostream& seekp(long);
```

Moves to an absolute position (as returned from **tellp**).

seekp

```
ostream& seekp(long, seek_dir);
```

Moves to a position relative to the current position, following the definition: **enum seek_dir {beg, cur, end};**

tellp

```
long tellp();
```

Returns the current stream position.

write

```
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
```

Inserts *n* characters (nulls included).

ostream_withassign <iostream.h>

This class is an **ostream** with an added assignment operator.

constructor

```
ostream_withassign();
```

Default constructor (calls **ostream**'s constructor).

Member functions

None (although the = operator is overloaded).

ostrstream <strstrea.h>

Provides output operations on a **strstreambuf**. This class is derived from **strstreambase** and **ostream**.

constructor

```
ostrstream();
```

Makes a dynamic **ostrstream**.

constructor

```
ostrstream(char*, int, int = ios::out);
```

Makes a **ostrstream** with a specified *n*-byte buffer. If **mode** is **ios::app** or **ios::ate**, the get/put pointer is positioned at the null character of the string.

Member functions

pcount

```
char *pcount();
```

Returns the number of bytes currently stored in the buffer.

str

```
char *str();
```

Returns and freezes the buffer. You must deallocate it if it was dynamic.

streambuf <iostream.h>

This is a buffer-handling class. Your applications gain access to buffers and buffering functions through a pointer to **streambuf** that is set by **ios**. **streambuf** is a base for **filebuf** and **strstreambuf**.

constructor

```
streambuf();
```

Creates an empty buffer object.

constructor

```
streambuf(char *, int);
```

Uses the given array and size as the buffer.

Member functions

allocate

```
int allocate();
```
protected

Sets up a buffer area.

base

```
char *base();
```
protected

Returns the start of the buffer area.

blen

```
int blen();
```
protected

Returns the length of buffer area.

eback

```
char *eback();
```
protected

Returns the base of putback section of get area.

ebuf

```
char *ebuf();
```
protected

Returns the end+1 of the buffer area.

egptr

```
char *egptr();
```
protected

Returns the end+1 of the get area.

epptr

```
char *epptr();
```
protected

Returns the end+1 of the put area.

gbump

```
void gbump(int);
```
protected

Advances the get pointer.

gptr

```
char *gptr();
```
protected

Returns the next location in get area.

in_avail

```
int in_avail();
```

Returns the number of characters remaining in the input buffer.

out_waiting

```
int out_waiting();
```

Returns the number of characters remaining in the output buffer.

pbase

```
char *pbase();
```
protected

Returns the start of put area.

pbump

```
void pbump(int);
```
protected

Advances the put pointer.

pptr `char *pptr();` **protected**

Returns the next location in put area.

sbumpc `int sbumpc();`

Returns the current character from the input buffer, then advances.

seekoff
```
virtual streampos seekoff(streamoff, ios::seek_dir,
                                int = (ios::in | ios::out);
```

Moves the get or put pointer (the third argument determines which one or both) relative to the current position.

seekpos `virtual streampos seekpos(streampos, int = (ios::in | ios::out));`

Moves the get or put pointer to an absolute position.

setb `void setb(char *, char *, int = 0 );` **protected**

Sets the buffer area.

setbuf
```
virtual streambuf* setbuf(signed char *, int);
streambuf* setbuf(unsigned char *, int);
```

Connects to a given buffer.

setg `void setg(char *, char *, char *);` **protected**

Initializes the get pointers.

setp `void setp(char *, char *);` **protected**

Initializes the put pointers.

sgetc `int sgetc();`

Peeks at the next character in the input buffer.

sgetn `int sgetn(char*, int n);`

Gets the next *n* characters from the input buffer.

snextc `int snextc();`

Advances to and returns the next character from the input buffer.

sputbackc `int sputbackc(char);`

Returns a character to input.

sputc `int sputc(int);`

Puts one character into the output buffer.

sputn `int sputn(const char*, int n);`

Puts *n* characters into the output buffer.

stossc `void stossc();`

Advances to the next character in the input buffer.

unbuffered `void unbuffered(int);` **protected**

Sets the buffering state.

unbuffered `int unbuffered();` **protected**

Returns non-zero if not buffered.

strstreambase <strstrea.h>

Specializes **ios** to string streams. This class is entirely protected except for the member function **strstreambase::rdbuf()**. This class is a base for **strstream, istrstream**, and **ostrstream**.

constructor `strstreambase();` **protected**

Makes an empty **strstreambase**.

constructor `strstreambase(char *, int, char *start);` **protected**

Makes an **strstreambase** with a specified buffer and starting position.

Member functions

rdbuf `strstreambuf * rdbuf();`

Returns a pointer to the strstreambuf associated with this object.

strstreambuf <strstrea.h>

Specializes **streambuf** for in-memory formatting.

constructor `strstreambuf();`

Makes a dynamic **strstreambuf**. Memory will be dynamically allocated as needed.

constructor

```
strstreambuf(void * (*)(long), void (*)(void *));
```

Makes a dynamic buffer with specified allocation and free functions.

constructor

```
strstreambuf(int n);
```

Makes a dynamic **strstreambuf**, initially allocating a buffer of at least *n* bytes.

constructor

```
strstreambuf(signed char *, int, signed char *end = 0);
strstreambuf(unsigned char *, int, unsigned char *end = 0);
```

Makes a static **strstreambuf** with a specified buffer. If *end* is not null, it delimits the buffer.

Member functions

doallocate

```
virtual int doallocate();
```

Performs low-level buffer allocation.

freeze

```
void freeze(int = 1);
```

If the input parameter is nonzero, disallows storing any characters in the buffer. Unfreeze by passing a zero.

overflow

```
virtual int overflow(int = EOF);
```

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

seekoff

```
virtual streampos seekoff(streamoff, ios::seek_dir, int);
```

Moves the pointer relative to the current position.

setbuf

```
virtual streambuf* setbuf(char*, int);
```

Specifies the buffer to use.

str

```
char *str();
```

Returns a pointer to the buffer and freezes it.

underflow

```
virtual int underflow();
```

Makes input available. This is called when a character is requested and the strstreambuf is empty. Every derived class should define the actions to be taken.

strstream <strstrea.h>

Provides for simultaneous input and output on a **strstreambuf**. This class is derived from **strstreambase** and **iostream**.

constructor

```
strstream();
```

Makes a dynamic **strstream**.

constructor

```
strstream(char*, int n, int mode);
```

Makes a **strstream** with a specified *n*-byte buffer. If *mode* is **ios::app** or **ios::ate**, the get/put pointer is positioned at the null character of the string.

Member function

str

```
char *str();
```

Returns and freezes the buffer. The user must deallocate it if it was dynamic.

C H A P T E R

6

Math

This chapter covers the floating-point options and explains how to use complex math.

Floating-point options

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor is set up to easily handle integer values, but it takes more time and effort to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387. We refer to this entire family of math coprocessors as the 80x87, or "the coprocessor."

If you have an 80486 processor, the numeric coprocessor is probably already built in.

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

Emulating the 80x87 chip

The default Turbo C++ code generation option is *emulation*. This option is for programs that may or may not have floating point, and for machines that may or may not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also generate information (FIXUPS) that Windows needs to run your program with no coprocessor. When the program runs, it will use the 80x87 if it is present; if no coprocessor is present at run time, it uses special software that *emulates* the 80x87.

Using 80x87 code

If your program is *only* going to run on machines with an 80x87 math coprocessor, you can save a small amount in your .EXE file size by omitting the 80x87 autodetection and emulation logic. Simply choose the 80x87 floating-point code generation option

No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing None for the floating-point code generation option. Then Turbo C++ will not link with MATH*x*.LIB.

Fast floating-point option

Turbo C++ has a fast floating-point option. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;
x = (float)(3.5*x);
```

To execute this correctly, *x* is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in *x*. Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

Registers and the 80x87

There are a couple of points concerning registers that you should be aware of when using floating point.

1. In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
2. If you are mixing floating point with inline assembly, you may need to take special care when using 80x87 registers. You might need to pop and save the 80x87 registers before calling functions that use the coprocessor, unless you are sure that enough free registers exist.

Disabling floating-point exceptions

By default, Turbo C++ programs abort if a floating-point overflow or divide by zero error occurs. You can mask these floating-point exceptions by a call to **_control87** in **main**, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
   _control87(MCW_EM,MCW_EM);
   ...
}
```

You can determine whether a floating-point exception occurred after the fact by calling **_status87** or **_clear87**. See the entries for these functions using *online Help* for details.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN will likely cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of **matherr** into your program.

```
#include <math.h>
int cdecl matherr(struct exception *e)
{
   return 1;            /* error has been handled */
}
```

Any other use of **matherr** to intercept math errors is not encouraged, as it is considered obsolete and may not be supported in future versions of Turbo C++.

Using complex math

Complex numbers are numbers of the form $x + yi$, where x and y are real numbers, and i is the square root of –1. Turbo C++ has always had a type

```
struct complex
{
   double  x, y;
};
```

defined in math.h. This type is convenient for holding complex numbers, as they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

See the description of class ***complex*** *in online Help for more information.*

To use complex numbers in C++, all you have to do is to include complex.h. In complex.h, all the following have been overloaded to handle complex numbers:

- all of the usual arithmetic operators
- the stream operators, **>>** and **<<**
- the usual math functions, such as **sqrt** and **log**

The complex library is invoked only if the argument is of type **complex**. Thus, to get the complex square root of –1, use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

As an example of the use of complex numbers, the following function computes a complex Fourier transform.

```
#include <complex.h>

// calculate the discrete Fourier transform of a[0], ..., a[n-1].
void Fourier(int n, complex a[], complex b[])
{
   int j, k;
   complex i(0,1);          // square root of -1
```

```
      for (j = 0; j < n; ++j)
      {
         b[j] = 0;
         for (k = 0; k < n; ++k)
            b[j] += a[k] * exp(2*M_PI*j*k*i/n);
         b[j] /= sqrt(n);
      }
   }
```

Using BCD math

Turbo C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This is sometimes confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.

Binary numbers are preferable for most applications, but in some situations the roundoff error involved in converting between base 2 and 10 is undesirable. The most common case is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```
#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n",x);
```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny roundoff error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Turbo C++ offers the C++ type **bcd**, which is declared in bcd.h. With **bcd**, the number 0.01 is represented exactly, and the **bcd** variable *x* will give an exact penny count.

```
#include <bcd.h>
int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
```

```
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";
```

Here are some facts to keep in mind about **bcd**.

- **bcd** does not eliminate all roundoff error: A computation like 1.0/3.0 will still have roundoff error.
- The usual math functions, such as **sqrt** and **log**, have been overloaded for **bcd** arguments.
- BCD numbers have about 17 decimal digits precision, and a range of about 1×10^{-125} to 1×10^{125}.

Converting BCD numbers

bcd is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is only performed when at least one operand is of the type **bcd**.

Important! The **bcd** member function **real** is available for converting a **bcd** number back to one the usual base 2 formats (**float**, **double**, or **long double**), though the conversion is not done automatically. **real** does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example,

```
bcd a = 12.1;
```

can be printed using any of the following four lines of code:

```
double x = a; printf("a = %g", x);

printf("a = %Lg", real(a));

printf("a = %g", (double)real(a));

cout << "a = " << a;
```

Note that since **printf** does not do argument checking, the format specifier must have the *L* if the **long double** value **real**(*a*) is passed.

Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a **bcd**. The number of places is an optional second argument to the constructor **bcd**. For example, to convert $1000.00/7 to a **bcd** variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

`1000.00/7`	=	142.85714…
`bcd(1000.00/7, 2)`	=	142.860
`bcd(1000.00/7, 1)`	=	142.900
`bcd(1000.00/7, 0)`	=	143.000
`bcd(1000.00/7, -1)`	=	140.000
`bcd(1000.00/7, -2)`	=	100.000

This method of rounding is specified by IEEE.

The number is rounded using banker's rounding, which means round to the nearest whole number, with ties being rounded to an even digit. For example,

`bcd(12.335, 2)`	=	12.34
`bcd(12.345, 2)`	=	12.34
`bcd(12.355, 2)`	=	12.36

C H A P T E R

7

BASM and inline assembly

This chapter tells you how to use the Turbo C++ built-in inline assembler (BASM) to include assembly language routines in your C and C++ programs without any need for a separate assembler. Such assembly language routines are called *inline assembly*, because they are compiled right along with your C routines, rather than being assembled separately, then linked together with modules produced by the C compiler.

Of course, Turbo C++ also supports traditional mixed-language programming in which your C program calls assembly language routines (or vice-versa) that are separately assembled by TASM (Turbo Assembler). In order to interface C and assembly language, you must know how to write 80x86 assembly language routines and how to define segments, data constants, and so on. You also need to be familiar with *calling conventions* (parameter passing sequences) in C and assembly language, including the **pascal** parameter passing sequence in C.

Inline assembly language

Turbo C++ lets you write assembly language code right inside your C and C++ programs. This is known as *inline assembly*.

BASM

If you don't invoke TASM, Turbo C++ can assemble your inline assembly instructions using the built-in assembler (BASM). This assembler can do everything TASM can do with the following restrictions:

- It cannot use assembler macros

- It cannot handle 80386 or 80486 instructions
- It does not permit Ideal mode syntax
- It allows only a limited set of assembler directives (see page 211)

Inline syntax

Of course, you also need to be familiar with the 80x86 instruction set and architecture. Even though you're not writing complete assembly language routines, you still need to know how the instructions you're using work, how to use them, and how not to use them.

Having done all that, you need only use the keyword **asm** to introduce an inline assembly language instruction. The format is

asm *opcode operands* ; or *newline*

where

- *opcode* is a valid 80x86 instruction (Table 7.1 lists all allowable *opcodes*).
- *operands* contains the operand(s) acceptable to the *opcode*, and can reference C constants, variables, and labels.
- ; or *newline* is a semicolon or a new line, either of which signals the end of the **asm** statement.

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 a number of **asm** statements, surround them with braces:

*The initial brace **must** appear on the same line as the **asm** keyword.*

```
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 is legal too */
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 Turbo C++ is generating from your C or C++ instructions. Any C

symbols are replaced with appropriate assembly language equivalents.

Because the inline assembly facility is not a complete assembler, it may not accept some assembly language constructs. If this happens, Turbo C++ will issue an error message. You then have two choices. You can simplify your inline assembly language code so that the assembler will accept it, or you can use an external assembler such as TASM. However, TASM might not identify the location of errors, since the original C source line number is lost.

Each **asm** statement counts as a C statement. For example,

```
myfunc()
{
   int  i;
   int x;

   if  (i > 0)
      asm  mov  x,4
   else
      i = 7;
}
```

This construct is a valid C **if** statement. Note that no semicolon was needed after the `mov x,4` instruction. **asm** statements are the only statements in C that depend on the occurrence of a new line. This is not in keeping with the rest of the C language, but this is the convention adopted by several UNIX-based compilers.

An assembly statement can be used as an executable statement inside a function, or as an external declaration outside of a function. Assembly statements located outside any function are placed in the data segment, and assembly statements located inside functions are placed in the code segment.

Opcodes

You can include any of the 80x86 instruction opcodes as inline assembly statements. There are four classes of instructions allowed by the Turbo 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

Note that 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 7.1
Opcode mnemonics

This table lists the opcode mnemonics that can be used in inline assembler.

aaa	fdivrp	fpatan	lsl
aad	feni	fprem	mov
aam	ffree**	fptan	mul
aas	fiadd	frndint	neg
adc	ficom	frstor	nop
add	ficomp	fsave	not
and	fidiv	fscale	or
bound	fidivr	fsqrt	out
call	fild	fst	pop
cbw	fimul	fstcw	popa
clc	fincstp**	fstenv	popf
cld	finit	fstp	push
cli	fist	fstsw	pusha
cmc	fistp	fsub	pushf
cmp	fisub	fsubp	rcl
cwd	fisubr	fsubr	rcr
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

*If you are using inline assembly in routines that use floating-point emulation, the opcodes marked with ** are not supported.*

String instructions

In addition to the listed opcodes, the string instructions given in the following table can be used alone or with repeat prefixes.

Table 7.2
String instructions

cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	outs	scasb	stosw
ins	lodsw	outsb	scasw	
insb	movs			

Prefixes

The following prefixes can be used:

```
lock    rep    repe    repne    repnz    repz
```

Jump instructions

Jump instructions are treated specially. Since a label cannot be included on the instruction itself, jumps must go to C labels (discussed in "Using jump instructions and labels" on page 213). The allowed jump instructions are given in the next table.

Table 7.3
Jump instructions

ja	jge	jnc	jns	loop
jae	jl	jne	jnz	loope
jb	jle	jng	jo	loopne
jbe	jmp	jnge	jp	loopnz
jc	jna	jnl	jpe	loopz
jcxz	jnae	jnle	jpo	
je	jnb	jno	js	
jg	jnbe	jnp	jz	

Assembly directives

The following assembly directives are allowed in Turbo C++ inline assembly statements:

```
db    dd    dw    extrn
```

Inline assembly references to data and functions

You can use C symbols in your **asm** statements; Turbo C++ automatically converts them to appropriate assembly language operands and appends underscores onto identifier names. You can use any symbol, including automatic (local) variables, register variables, and function parameters.

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. Simply using the name will include the correct offsets.

However, it may be necessary 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 fashion (that is, *variable.member*). In such a case, you are dealing with a variable, and you can store or retrieve values. 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, myA.a_b
         mov  bx, [di].a_c
        }
   ...
}
```

We've declared a structure type named *myStruct* with three members, *a_a*, *a_b*, and *a_c*; we've also declared 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]
```

Why would you even want to do this? 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 actually 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.

However, there is one restriction. If two structures that you are 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. They are only valid 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 be automatically converted to a long-distance jump. 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, you can use a register name as the operand of the jump instruction.

Interrupt functions

The 80x86 reserves the first 1024 bytes of memory for a set of 256 far pointers—known as interrupt vectors—to special system routines known as *interrupt handlers*. These routines are called by executing the 80x86 instruction

int *int#*

where *int#* goes from 0h to FFh. When this happens, the computer saves the code segment (CS), instruction pointer (IP), and status flags, disables the interrupts, then does a far jump to the location pointed to by the corresponding interrupt vector. For example, one interrupt call you're likely to see is

```
int  21h
```

which calls most DOS routines. But many of the interrupt vectors are unused, which means, of course, that you can write your own interrupt handler and put a **far** pointer to it into one of the unused interrupt vectors.

To write an interrupt handler in Turbo C++, you must define the function to be of type **interrupt**; more specifically, it should look like this:

```
void  interrupt myhandler(bp, di, si, ds, es, dx,
                          cx, bx, ax, ip, cs, flags, ... );
```

As you can see, all the registers are passed as parameters, so you can use and modify them in your code without using the pseudovariables discussed earlier in this chapter. You can also pass additional parameters (*flags*, ...) to the handler; those should be defined appropriately.

A function of type **interrupt** will automatically save (in addition to SI, DI, and BP) the registers AX through DX, ES, and DS. These same registers are restored on exit from the interrupt handler.

Interrupt handlers may use floating-point arithmetic in all memory models. Any interrupt handler code that uses an 80x87 must

save the state of the chip on entry and restore it on exit from the handler.

An interrupt function can modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This may be useful when you are using an interrupt handler to act as a user service, much like the DOS INT 21 services. Also, note that an interrupt function exits with an IRET (return from interrupt) instruction.

So, why would you want to write your own interrupt handler? For one thing, that's how most memory-resident routines work. They install themselves as interrupt handlers. That way, whenever some special or periodic action takes place (clock tick, keyboard press, and so on), these routines can intercept the call to the routine handling the interrupt and see what action needs to take place. Having done that, they can then pass control on to the routine that was there.

Using low-level practices

You've already seen a few examples of how to use these different low-level practices in your code; now it's time to look at a few more. Let's start with an interrupt handler that does something harmless but tangible (or, in this case, audible): It beeps whenever it's called.

First, write the function itself. Here's what it might look like:

```
#include        <dos.h>

void  interrupt  mybeep(unsigned bp, unsigned di, unsigned si,
                        unsigned ds, unsigned es, unsigned dx,
                        unsigned cx, unsigned bx, unsigned ax)
{
   int     i, j;
   char    originalbits, bits;
   unsigned char   bcount = ax >> 8;

   /* Get the current control port setting */
   bits = originalbits = inportb(0x61);

   for (i = 0; i <= bcount; i++){

      /* Turn off the speaker for awhile */
      outportb(0x61, bits & 0xfc);
      for (j = 0; j <= 100; j++)
         ;   /* empty statement */
```

```
      /* Now turn it on for some more time */
      outportb(0x61, bits | 2);
      for (j = 0; j <= 100; j++)
         ;   /* another empty statement */
      }

   /* Restore the control port setting */
   outportb(0x61, originalbits);
}
```

Next, write a function to install your interrupt handler. Pass it the address of the function and its interrupt number (0 to 255 or 0x00 to 0xFF).

```
void  install(void interrupt (*faddr)(), int inum)
{
   setvect(inum, faddr);
}
```

Finally, call your beep routine to test it out. Here's a function to do just that:

```
void  testbeep(unsigned char bcount, int inum)
{
   _AH = bcount;
   geninterrupt(inum);
}
```

Your **main** function might look like this:

```
main()
{
   char  ch;

   install(mybeep,10);
   testbeep(3,10);
   ch = getch();
}
```

You might also want to preserve the original interrupt vector and restore it when your main program is finished. Use the **getvect** and **setvect** functions to do this.

C H A P T E R

8

Building a Windows application

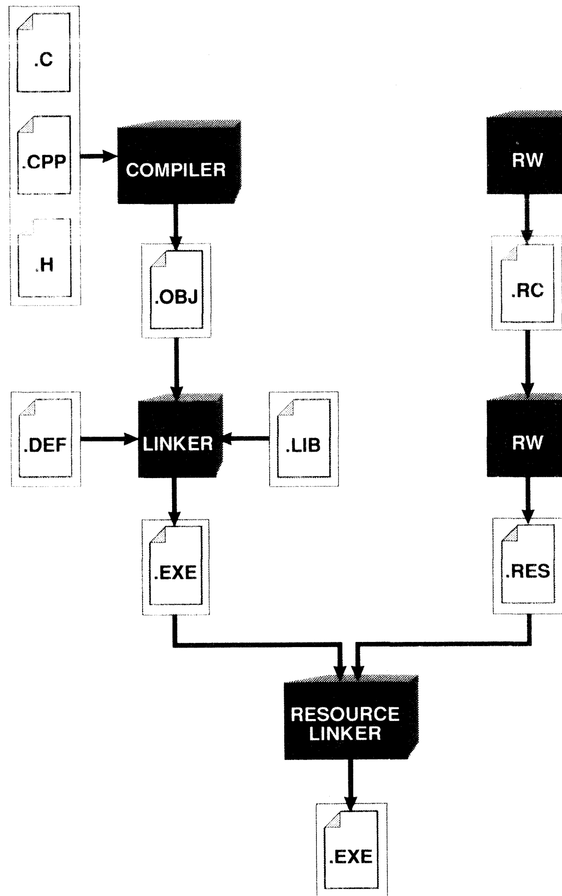
We don't explain the intricacies of designing Windows applications, nor teach you how to program under Windows—these topics go beyond the scope of this chapter or book.

This chapter explains how to use Turbo C++ to build Windows applications or dynamic link libraries (DLLs). Compiling and linking a module for Windows is basically the same as it is for DOS. The compiler first generates an object file which differs from a DOS compilation primarily in the special Windows prolog and epilog code that wraps each function. The prolog and epilog code varies depending on which Windows compilation options are used; these options are described later.

To create a Windows module for the memory model you are compiling under, the linker links the object files with the appropriate Turbo C++ startup code, various libraries, and the module definition file.

Finally, the IDE's resource linker binds the resources to the module. Figure 8.1 illustrates the entire process.

Figure 8.1
Compiling and linking
a Windows program



The next section, "Compiling and linking with the IDE," gives you a quick example of how to compile, link, and run a Windows program in the Turbo C++ IDE.

Compiling and linking within the IDE

You can find complete descriptions of the various IDE commands and options in Chapter 1 of the User's Guide.

By way of example, you'll be producing a simple Windows application called WHELLO, which creates a window and writes a text message to that window. WHELLO.EXE is produced by compiling and linking the following three files:

- WHELLO.CPP, the C++ source file
- WHELLO.RC, the resource file
- WHELLO.DEF, the module definition file

Understanding resource files

Windows applications typically use *resources*, which can be icons, menus, dialog boxes, fonts, cursors, bitmaps, or user-defined resources. These resources are defined in a file called a resource file. For this application, the resource file is WHELLO.RC.

.RC resource files are source files, also called resource script files. Before an .RC file can be added to an executable, the .RC file must first be compiled by the Resource Workshop into a binary format; compilation creates a .RES file. For instance, you can use the Resource Workshop to compile WHELLO.RC, creating WHELLO.RES. The resource linker in the IDE will bind WHELLO.RES to WHELLO.EXE.

Understanding module definition files

The module definition file WHELLO.DEF provides information to the linker about the contents and system requirements of a Windows application. Because the linker has other ways of finding out the information contained in the module definition, module definition files are not required for Turbo C++'s linker to create a Windows application, although one is included here for the sake of example.

Compiling and linking WHELLO

You can also open the WHELLO.PRJ project file, and skip the process of adding files to the project.

Here's how you turn these three files into a Windows application:

1. Choose Project | Open Project. In the Project Name box, type WHELLO.PRJ. Press *Enter* or click OK to open a new project with the name WHELLO.
2. Choose Project | Add item and type `whello.*` in the Name box, so that you'll get a list of all the WHELLO files. Press *Enter* or click OK.
3. Add the three files WHELLO.CPP, WHELLO.RES, and WHELLO.DEF for the application. Close the dialog box after you've added the three files.

4. Choose Options | Application to open the Application Options dialog box and select Windows App if it's not already selected. The information pane at the top of the dialog box changes. Each of the buttons at the bottom of the dialog box sets several other options in the IDE.
5. Choose Run | Run to build the project and then run the completed WHELLO.EXE application.

That's all there is to building and running a Windows application with Turbo C++. You can generalize this process into the following checklist:

1. Create a project.
2. Use the Resource Workshop to visually create your resources or to compile .RC resource scripts into .RES resource files.
3. Add the source files, resource files, import libraries (if necessary), and the module definition file (if necessary) to the project.
4. Set up the compilation and link environment with the Application Options dialog box, or with a combination of other settings and options.
5. Build the project.
6. Run the application.

Using the project manager

Specifying a .RES file causes the Project Manager to use the built-in resource linker to bind the resources to the executable file. If you specify an .RC file, the Project Manager will look for the corresponding .RES file. You can use the Resource Workshop to create .RES files from .RC files.

For example, if you enter HELLO.CPP, HELLO.RC, and HELLO.DEF into a project, the Turbo C++ Project Manager will

You will need to use the Resource Workshop to compile the .RC file into a .RES file

- create HELLO.OBJ by compiling HELLO.CPP with the C++ compiler
- create HELLO.EXE by linking HELLO.OBJ with its appropriate libraries, using information contained in HELLO.DEF
- create the final HELLO.EXE by using the resource linker to bind the resources contained in HELLO.RES to HELLO.EXE

Setting compile and link options

The bulk of the setup in this example is accomplished by the Application Options dialog box. The command buttons in this dialog box check or set various other options in other dialog boxes. Turbo C++ makes it easy for you to change the settings that control compilation and linking of your programs, so you'll want to familiarize yourself with the following dialog boxes (all described in full in Chapter 4 in the *User's Guide*):

- The Code Generation Options dialog box sets such things as the memory model, tells the compiler to use precompiled headers, and more. Choose Options | Compiler | Code Generation to see this dialog box.
- The Entry/Exit Code Generation dialog box sets Turbo C++ compiler options for prolog and epilog code generation, and export options. Choose Options | Compiler | Entry/Exit Code and browse through the contents of this dialog box.
- The Linker Settings dialog box (Options | Linker | Settings) sets options for the type of output you want from the linker—such as a a Windows .EXE or a Windows DLL—as well as a number of other linker options.

WinMain

You must supply the **WinMain** function as the main entry point for a Windows application;

The following parameters are passed to **WinMain**:

The HANDLE and LPSTR types are defined in windows.h.

```
int PASCAL WinMain(HANDLE hInstance, HANDLE 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.
- *lpCmdLine* is a far pointer 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.

The return value from **WinMain** is not currently used by Windows. However, it can be useful during debugging since Turbo Debugger for Windows can display this value when your program terminates.

Prologs and epilogs

The need for prologs and epilogs is not new to Windows; they must be generated for code intended for DOS as well. However, if the program is intended for Windows, the compiler generates a different prolog and epilog than it would for DOS.

When you compile a module for Windows, the compiler needs to know which kind of prolog and epilog to create for each of a module's functions. Settings in the IDE and options for the command-line compiler 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 near and far 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 options dictate the exact instructions contained in the code.

The following list describes the effects of the different prolog/epilog code generation options and their corresponding command-line compiler options. To set these options in the IDE, choose Options | Compiler | Entry/Exit Code.

Windows All Functions Exportable

This option creates a Windows application object module with all far functions exportable.

This is the most general kind of Windows application module, although not necessarily the most efficient. The compiler generates a prolog and epilog for every far function that makes the function exportable. This does not mean that all far functions actually will be exported, it only means that the function can be exported. In order 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.

*See page 52 for description and usage of the **_export** keyword.*

Windows Explicit Functions Exported

This option creates an object module with only those functions marked as **_export** exportable.

Since, in any given application module, many of the functions won't be exported, it is not 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.

Note that the Windows Explicit Functions Exported option *only* works in conjunction with the **_export** keyword. This option does not 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

This option creates an object module with functions using smart callbacks.

This form of prolog and epilog assumes that DS == SS; in other words, that 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 here don't need the **_export** keyword or 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 since DLLs assume DS != SS.

Because of the assumption that DS == SS, you can only use this option for applications, not DLLs. Furthermore, you must not explicitly change DS in your program (a very unsafe practice under Windows in any circumstance).

Windows DLL All Functions Exportable

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.

Windows DLL Explicit Functions Exported

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.

The _export keyword

*Note that 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 a function declaration, **_export** immediately precedes the function name; for example,

```
LONG FAR PASCAL _export MainWindowProc( HWND hWnd, unsigned iMessage,
                                        WORD wParam, LONG lParam )
```

You can also use **_export** with a C++ class definition; see page 233.

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; if so, the function is called exportable. Second, the linker must create an entry for every export function in the header section of the executable. All of this occurs so that the correct data segment can be bound to the function at run time.

If a function is flagged with the **_export** keyword and any of the Windows compiler options are used, it will be compiled as exportable and linked as an export.

If a function is *not* flagged with the **_export** keyword, then Turbo C++ will take one of the following actions:

- If you compile with either of the All functions exportable options, the function will be compiled as exportable.

  If the function is listed in the EXPORTS section of the module definition file, then 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 either of the Explicit functions exported options, the function will *not* be compiled as exportable. Including this function in the EXPORTS section of the module definition will cause it be exported, but, because the prolog is incorrect, the program will run incorrectly. You may get the Windows error message, "Unrecoverable Application Error."

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** and **far**?	Yes	Yes	Yes	Yes	No	No	No	No
Function listed in EXPORTS?	Yes	Yes	No	No	Yes	Yes	No	No
And the Entry/Exit code option is:*	All	Explicit	All	Explicit	All	Explicit	All	Explicit
Will function be exportable?	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Will function be exported?	Yes	Yes	Yes	Yes	Yes	Yes**	No***	No

* 'All' means either 'Windows all functions exportable' or 'Windows DLL all functions exportable' and 'Explicit' means either 'Windows explicit functions exported' or 'Windows DLL explicit functions exported.' You can select these options in the Options | Compiler | Entry/Exit Code dialog box.

** 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.

*** This combination also makes little sense. It's inefficient to compile all functions as exportable if you don't actually export some of them.

Memory models

You can use the small, medium, compact, or large memory models with any kind of Windows executable, including DLLs.

Module definition files

The module definition file is not strictly necessary to produce a Windows executable under Turbo C++. If no module definition file is specified, the following defaults are assumed.

CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE (for applications) or PRELOAD MOVEABLE SINGLE (for DLLs)
HEAPSIZE	4096
STACKSIZE	5120

To replace the EXETYPE statement, the Turbo C++ linker can discover what kind of executable you want to produce by checking settings in the IDE or options on the command line. You can include an import library to substitute for the IMPORTS section of the module definition.

You can use the **_export** keyword in the definitions of export functions in your C and C++ source code to remove the need for an EXPORTS section. Note, however, that if **_export** is used to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need to have a module definition file.

A quick example

Here's the module definition from the WHELLO example:

```
NAME          WHELLO
DESCRIPTION   'C++ Windows Hello World'
EXETYPE       WINDOWS
CODE          MOVEABLE
DATA          MOVEABLE MULTIPLE
HEAPSIZE      1024
STACKSIZE     5120
EXPORTS       MainWindowProc
```

Let's take this file apart, statement by statement:

- NAME specifies a name for an application. If you want to build a DLL instead of an application, you would use the LIBRARY statement instead. 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.
- DESCRIPTION lets you specify a string that describes your application or library.
- EXETYPE can be either WINDOWS or OS2. Only WINDOWS is supported in this version of Turbo C++.
- CODE defines the default attributes of code segments. The MOVEABLE option means that the code segment can be moved in memory at run-time.
- DATA defines the default attributes of data segments. MOVEABLE means that it can be moved in memory at run-time. Windows lets you run more than one instance of an application at the same time. In support of that, the MULTIPLE options 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.
- EXPORTS lists those functions in the WHELLO application that will be called by other applications or by Windows. Functions that are intended to be called by other modules are called callbacks, callback functions, or export functions.
- To help you avoid the necessity of creating and maintaining long EXPORTS sections, Turbo C++ provides the **_export** keyword. Functions flagged with **_export** will be identified by the linker and entered into an export table for the module. If the Smart Callbacks option is used at compile time (Options | Compiler | Entry/Exit Code), then callback functions do *not* need to be listed either in the EXPORTS statement or flagged with the **_export** keyword. Turbo C++ compiles them in such a way so that they can be callback functions.

This application doesn't have an IMPORTS statement, because the only functions it calls from other modules are those from the Windows API; those functions are imported via the automatic inclusion of the IMPORT.LIB import library. When an application

needs to call other external functions, these functions must be listed in the IMPORTS statement, or included via an import library (see page 229 for a discussion of import libraries).

This application doesn't include a STUB statement. Turbo C++ uses a built-in stub for Windows applications. The built-in stub simply checks to see if the application was loaded under Windows, and, if not, terminates the application with a message that Windows is required. If you want to write and include a custom stub, specify the name of that stub with the STUB statement.

Linking for Windows

In general, Turbo C++ needs to take object files compiled with the correct Windows options and then link them with the proper Windows initialization code, run-time and math libraries, and a module definition file. Settings in the Linker Settings dialog box in the IDE do this for you automatically.

Linking in the IDE

With the Linker Settings dialog box in the IDE, you can set link options for a Windows application or DLL. Options in the IDE override settings in the module definition file. This means if you check the Windows EXE box instead of the Windows DLL box, and the module definition file has a LIBRARY statement instead of a NAME statement, the file will be linked as a Windows application, not a DLL.

The linker uses the C0W*x*.OBJ initialization file for applications and the C0D*x*.OBJ initialization file for DLLs, where *x* depends on the memory model set in the Code Generation dialog box. For both Windows options, the linker uses the current project object files and libraries, IMPORT.LIB, MATHW*x*.LIB, and CW*x*.LIB. Turbo C++ allows you to override the default setting for a memory model.

Dynamic link libraries

A dynamic link library (DLL) is a library of functions that a Windows module can call to accomplish a task. If you've written a Windows application, then you've already used DLLs. The files

KERNEL.EXE, USER.EXE, and GDI.EXE are actually DLLs, not applications (as the .EXE extension implies). The references to the API functions that you call from these modules are resolved at run time (dynamic linking), instead of at link time (static linking).

Compiling and linking a DLL within the IDE

To compile and link a DLL from within the IDE, follow these steps:

*The **_export** keyword should immediately precede the function name.*

1. Create the DLL source files. Optionally, create the resource file and the module definition file.
2. Choose Project | Open Project to start a new project.
3. Choose Project | Add Item, and add the source and resource files for the DLL.
4. If you have created a module definition file for the DLL, add it to the project. (Note that Turbo C++ can link without one. To link without a module definition file for the DLL, you must have flagged every function to be exported in the DLL with the keyword **_export**. In addition, choose Options | Compiler | Entry/Exit Code | Windows DLL Explicit Functions Exportable.)
5. Choose Options | Application | Windows DLL.
6. Choose Compile | Build all.

Import libraries

If a Windows application module or another DLL uses functions from a DLL, you have two ways to tell the linker about them:

- You can add an IMPORTS section to the module definition file and list every function from DLLs that the module will use.
- Or you can include the import library for the DLLs when you link the module. See the following section for information on creating your own import libraries.

Creating an import library

The IMPLIBW import librarian for Windows creates an import library that can be substituted for part or all of the IMPORTS section of a module definition file for a Windows application.

An import library lists some or all of the exported functions for one or more DLLs. IMPLIBW creates an import library directly

from DLLs or from module definition files for DLLs (or a combination of the two).

Select an import library

To create an import library with IMPLIBW, double-click on the IMPLIBW icon. An empty window will appear. Selecting File | Create will bring up a dialog box listing all DLLs in the current directory. You can navigate between directories as normal by double-clicking on directory names in the list box.

From a DLL

When you select a DLL and click on Create (or simply double-click on a DLL filename), IMPLIBW will create an import library with the same base name as the selected DLL, and an extension of .LIB.

From a module definition file

If you have a module definition (.DEF) file for a DLL, you can use it instead of the DLL itself to create an import library.

Instead of selecting a DLL, type the name of a .DEF file into the edit control and click on Create. IMPLIBW will create an import library with the same base name as the selected .DEF file, and an extension of .LIB.

Creating the import library

If there were no errors as IMPLIBW examined the DLL or module definition file and created the import library, the window will report "No Warnings." That's all there is to it. You now have a new import library that you can include in a project.

Creating DLLs

The following sections provide information on the specifics of writing a DLL.

LibMain and WEP

You must supply the **LibMain** function as the main entry point for a Windows DLL.

Windows calls **LibMain** once, when the library is first loaded. **LibMain** performs initialization for the DLL. This initialization depends almost entirely on the function of the particular DLL, but might include the following tasks:

- Unlocking the data segment with **UnlockData**, if it has been declared as MOVEABLE
- Setting up global variables for the DLL, if it uses any

The DLL startup code C0Dx.OBJ initializes the local heap automatically; you do not need to include code in **LibMain** to do this.

The following parameters are passed to **LibMain**:

HANDLE, WORD, and LPSTR are defined in windows.h.

```
int FAR PASCAL LibMain (HANDLE 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 since 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 inititalization). If 0, Windows will unload the DLL from memory.

The exit point of a DLL is the function **WEP** (which stands for Windows Exit Procedure). This function is not necessary in a DLL (since the Turbo C++ run-time libraries provide a default) but can be supplied by the writer of a DLL to perform any cleanup of the DLL before it is unloaded from memory. Windows will call **WEP** just prior to unloading the DLL.

Under Turbo C++, **WEP** does not need to be exported. Turbo C++ defines its own **WEP** that calls your **WEP**, 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. The former means that all of Windows is shutting down and the latter indicates that just this DLL is being unloaded.

WEP should return 1 to indicate success. Windows currently doesn't do anything with this return value.

Pointers and memory

Functions in a DLL are not linked directly into a Windows application; 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 need to be far as well.

Let's say 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, the function prototype would take this form:

```
extern int _export FAR f(char FAR *dstring);
```

In the DLL, 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 and list **f1** in the EXPORTS section of the module definition file, or you can flag the function with the **_export** keyword, like so:

```
int FAR _export f1(char far *dstring)
{
:
}
```

*Before an application could use **f1**, it would have 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.*

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 this form

```
extern int FAR f(char FAR *dstring);
```

because the prototype would need to be correct for a module compiled with a smaller memory model. But in the DLL, the function could be defined like this:

```
int _export f1(char *dstring)
{
:
}
```

Static data in DLLs

Through a DLL's functions, all applications using the DLL have access to that DLL's global data. A particular function will use the same data, regardless of the application that called it. If you want a DLL's global data to be protected for use by a single application, you would 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 will need to dynamically allocate the data and manage the access to that data manually. Static data in a DLL is global to all callers of a DLL.

C++ classes and pointers

A C++ class used only inside a DLL doesn't need to be declared **far**. The class requires special handling if it will be used from another DLL or a Windows application.

All the members of a shared class must be far. Do this by declaring the class members as **far** or compiling the DLL under the large memory model. The classes also must be exported, which can be accomplished two ways:

- Include the names of all the class members in the EXPORTS section of the module definition file, then compile the DLL with the Options | Compiler | Entry/Exit code | Windows DLL All Functions Exportable option.
- Mark the entire class with the **_export** keyword and compile the DLL with the Options | Compiler | Entry/Exit code | Windows DLL Explicit Functions Exported option.

C++ classes use virtual table pointers and include a hidden **this** pointer. Both pointers must be far pointers as well. There are two basic ways to accomplish this.

One way is to simply compile the DLL modules and the application using the DLL with the Far Virtual Tables option (Options | Compiler | C++ Options in the IDE). This causes all virtual table pointers and **this** parameters to be full 32-bit pointers. The advantage of this approach is that it does not require any source code changes; however, all classes, shared or not, suffer the overhead of 32-bit pointers.

Note that a huge class can only inherit from other huge classes.

A more efficient approach is to declare the shared classes **huge** instead of **far** which tells the compiler to use full 32-bit pointers for those classes only. Here is an example of a huge class declaration:

```
class huge DLLclass
{
:
};
```

For a class that is defined in a DLL to be usable from a Windows application, its non-inline member functions and static data members must be made available by making them exported names. You can do this by adding their public (mangled) names to the EXPORTS section of the DLL module definition file, but this can be rather tedious.

There's an easier alternative: Declare the classes to be exported as **_export**. Whenever a class is declared as **_export**, Turbo C++ treats it as huge (with 32-bit pointers), and automatically exports all its non-inline member functions and static data members. If you declare a class as **_export**, you can't also declare it as **far** or **huge** (**_export** implies **huge**, which implies **far**).

If you declare the class in an include file that is included both by the DLL source files and by the source files of the application using the DLL, such a class should be declared **_export** when compiling the DLL, and merely **huge** when compiling the application. To do this, you can use the __DLL__ macro, which is defined by the compiler when it's building a DLL. The following code could be a part of an include file that defines a shared class:

Note that the compiler encodes (in the mangled name) the information that a given class member is a member of a huge class. This ensures that any mismatches are caught by the linker when a program is using huge and non-huge classes.

```
#ifdef  __DLL__
#  define  EXPORT   _export
#else
#  define  EXPORT   huge
#endif

class EXPORT DLLclass
{
:
};
```

C H A P T E R

9

Error messages

Turbo C++ error messages include: compile-time, Help, run-time, Librarian, and Linker. We explain them here; user-interface error messages are explained in online Help.

The *type* of message (such as *Compile-time* or *Help*) is noted in the column to the left. Most explanations provide a probable cause and remedy for the error or warning message.

Finding a message in this chapter

The messages are listed in ASCII alphabetic order; messages beginning with symbols normally come first, followed by numbers and letters of the alphabet.

Since messages that begin with a variable cannot be alphabetized by what you will actually see when you receive such a message, all such messages are alphabetized by the word following the variable.

For example, if you have a C++ function **goforit**, you might receive the following actual message:

```
goforit must be declared with no arguments
```

In order to look this error message up, you would need to find

***function* must be declared with no arguments**

alphabetized starting with the word "must".

If the variable occurs later in the text of the error message (for example, "Address of overloaded function *function* doesn't match

Type"), you can find the message in correct alphabetical order; in this case, under the letter *A*.

Types of messages

The kinds of messages you get are different, depending on where they come from. This section lists each category with a table of variables that it may contain.

Compile-time messages

The Turbo C++ compiler diagnostic messages fall into three classes: fatal errors, errors, and warnings.

Fatal errors are rare. Some of them indicate an internal compiler error. When a fatal error occurs, compilation stops immediately. You must take appropriate action and then restart compilation.

Errors indicate program syntax errors, and disk or memory access errors. The compiler completes the current phase of the compilation and then stop. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing and code-generating).

Warnings do not prevent the compilation from finishing. They indicate conditions that are suspicious, but are usually legitimate as part of the language. The compiler also produces warnings if you use some machine-dependent constructs in your source files.

The compiler prints messages with the message class first, then the source file name and line number where the compiler detected the condition, and finally the text of the message itself.

Line numbers are not exact

You should be aware of one detail about line numbers in error messages: the compiler only generates messages as they are detected. Because C and C++ do not force any restrictions on placing statements on a line of text, the true cause of the error may be one or more lines before or after the line number mentioned.

The following variable names and values are some of those that appear in the compiler messages listed in this chapter (most are self-explanatory). When you get an error message, the appropriate name or value is substituted.

Table 9.1 Compile-time message variables

What you'll see in the manual	What you'll see on your screen
argument	An argument
class	A class name
filename	A file name (with or without extension)
function	A function name
group	A group name
identifier	An identifier (variable name or other)
language	The name of a programming language
member	The name of a data member or member function
message	A message string
module	A module name
number	An actual number
option	An option
parameter	A parameter name
segment	A segment name
specifier	A type specifier
symbol	A symbol name
type	A type name
XXXXh	A 4-digit hexadecimal number, followed by *h*

Help compiler messages

The Help Compiler displays messages when it encounters errors or warnings in building the Help resource file. Messages during processing of the project file are numbered beginning with the letter *P* and appear as in the following examples:

```
Error P1025: line...7 of filename.HPJ : Section heading sectionname
unrecognized.

Warning P1039: line...38 of filename.HPJ : [BUILDTAGS] section
missing.
```

Messages that occur during processing of the RTF topic file(s) are numbered beginning with the letter *R* and appear as in the following examples:

```
Error R2025: File environment error.

Warning R2501: Using old key-phrase table.
```

Topic numbers

Whenever possible, the compiler will display the topic number or file name that contains the error. If you numbered your topics, the topic number given with an error message refers to that topic's sequential position in your RTF file (first, second, and so on). These numbers may be identical to the page number shown by your word processor, depending on the number of lines you have assigned to the hypothetical printed page. Remember that topics

are separated by hard page breaks, even though there is no such thing as a "page" in online Help.

Messages beginning with the word "Error" (on your screen) are fatal errors. Errors are always reported, and no usable Help resource file will result from the build. Messages beginning with the word "Warning" (on your screen) are less serious in nature. A build with warnings will produce a valid Help resource file that will load under Windows, but the file may contain operational errors. You can specify the amount of warning information to be reported by the compiler.

During processing of the project file, the compiler ignores lines that contain errors and attempts to continue with the build. This means that errors encountered early in a build may result in many more errors being reported as the build continues. Similarly, errors during processing of the RTF topic files will be reported and if not serious, the compiler will continue with the build. A single error condition in the topic file may result in more than one error message being reported by the compiler. For instance, a misidentified topic will cause an error to be reported every time jump terms refer to the correct topic identifier. Such a mistake is easily rectified by simply correcting the footnote containing the wrong context string.

Table 9.2
Help message variables

What you'll see in the manual	**What you'll see on your screen**
contextname	A context string alias
context-string	A context string
filename	A file name (with or without extension)
fontname	The name of a font
optionname	An option name
sectionname	A section heading
tagname	A build tag
topicnumber	A topic number

Run-time error messages

Turbo C++ has a small number of run-time error messages. These errors occur after the program has successfully compiled and while it is running.

Librarian messages

Librarian has error and warning messages. The following generic names and values appear in Librarian messages. When you get a message, the variable is substituted.

Table 9.3
Librarian message variables

What you'll see in the manual	What you'll see on your screen
filename	A file name (with or without extension)
function	A function name
len	An actual number
module	A module name
num	An actual number
path	A path name
reason	Reason given in warning message
size	An actual number
type	A type name

Linker messages

The linker has three types of messages: fatal errors, errors, and warnings.

- A fatal error causes the linker to stop immediately; the .EXE file is deleted.
- An error (also called a nonfatal error) does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file. Errors are treated as fatal errors in the IDE.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

Table 9.4
Linker error message variables

What you'll see in the manual	What you'll see on your screen
errorcode	Error code number for internal errors
filename	A file name (with or without extension)
group	A group name
linenum	The line number within a file
module	A module name
segment	A segment name
symbol	A symbol name
XXXXh	A 4-digit hexadecimal number, followed by *h*

Message explanations

Compile-time error

(expected

A left parenthesis was expected before a parameter list.

Compile-time error

) expected

A right parenthesis was expected at the end of a parameter list.

Compile-time error

, expected

A comma was expected in a list of declarations, initializations, or parameters.

Compile-time error

: expected after private/protected/public

When used to begin a **private/protected/public** section of a C++ class, these reserved words must be followed by a colon.

Compile-time error

< expected

The keyword **template** was not followed by a left angle bracker (<). Every template declaration must include the template formal parameters enclosed within angle brackets (< >), immediately following the **template** keyword.

Compile-time error

{ expected

A left brace ({) was expected at the start of a block or initialization.

Compile-time error

} expected

A right brace (}) was expected at the end of a block or initialization.

Run-time error

Abnormal program termination

The program called **abort** because Windows fails to execute. Can happen through memory overwrites.

Compile-time error

Access can only be changed to public or protected

A C++ derived class may modify the access rights of a base class member, but only to **public** or **protected**. A base class member cannot be made **private**.

Librarian warning

added file *filename* does not begin correctly, ignored

The librarian has decided that in no way, shape, or form is the file being added an object module, so it will not try to add it to the library. The library is created anyway.

Compile-time error

Address of overloaded function *function* doesn't match *type*

A variable or parameter is assigned/initialized with the address of an overloaded function, and the type of the

variable/parameter doesn't match any of the overloaded functions with the specified name.

Compile-time error

Ambiguity between *function1* and *function2*

Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.

Compile-time error

Ambiguous member name *name*

A structure member name used in inline assembly must be unique. If it is defined in more than one structure all of the definitions must agree in type and offset within the structures. The member name in this case is ambiguous. Use the syntax `(struct xxx).yyy` instead.

Compile-time warning

Ambiguous operators need parentheses

This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears unparenthesized with a shift operator will produce this warning. Programmers frequently confuse the precedence of these operators.

Compile-time error

Array allocated using new may not have an initializer

When initializing a vector (array) of classes, you must use the constructor that has no arguments. This is called the *default constructor*, which means that you may not supply constructor arguments when initializing such a vector.

Compile-time error

Array bounds missing]

Your source file declared an array in which the array bounds were not terminated by a right bracket.

Compile-time error

Array must have at least one element

ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed). An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with **malloc**. You can still use this trick, but you must declare the array element to have (at least) one element if you are compiling in strict ANSI mode. Declarations (as opposed to definitions) of arrays of unknown size are still allowed, of course.

For example,

```
char ray[];         /* definition of unknown size -- illegal */
char ray[0];        /* definition of 0 size -- illegal */
extern char ray[];  /* declaration of unknown size -- ok */
```

Compile-time error

Array of references is not allowed

It is illegal to have an array of references, since pointers to references are not allowed and array names are coerced into pointers.

Compile-time warning

Array size for 'delete' ignored

With the latest specification of C++, it is no longer necessary to specify the array size when deleting an array; to allow older code to compile, Turbo C++ ignores this construct, and issues this warning.

Compile-time error

Array size too large

The declared array is larger than 64K.

Compile-time warning

Array variable *identifier* is near

Whenever you use the IDE Options | Compiler | Advanced Code Generation... | Far Data Threshold selection to set threshold limit, global variables larger than the threshold size are automatically made far by the compiler. However, when the variable is an initialized array with an unspecified size, its total size is not known when the decision whether to make it near or far has to be made by the compiler, and so it is made near. If the number of initializers given for the array causes the total variable size to exceed the data size threshold, the compiler issues this warning. If the fact that the variable is made near by the compiler causes problems (for example, the linker reports a group overflow due to too much global data), you must make the offending variable explicitly far by inserting the keyword **far** immediately to the left of the variable name in its definition.

Compile-time error

Assembler statement too long

Inline assembly statements may not be longer than 480 bytes.

Compile-time warning

Assigning *type* to *enumeration*

Assigning an integer value to an **enum** type. This is an error in C++, but is reduced to a warning to give existing programs a chance to work.

Compile-time error

Assignment to this not allowed, use X::operator new instead

In early versions of C++, the only way to control allocation of class of objects was by assigning to the **this** parameter inside a constructor. This practice is no longer allowed, since a better, safer, and more general technique is to define a member function **operator new** instead.

Linker warning

Attempt to export non-public symbol *symbol*

A symbol name was listed in the EXPORTS section of the module definition file, but no symbol of this name was found as public in the modules linked. This either implies a mistake in spelling or case, or that a procedure of this name was not defined.

Compile-time error

Attempt to grant or reduce access to *identifier*

A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class. It cannot add or reduce access rights.

Compile-time error

Attempting to return a reference to a local object

In a function returning a reference type, you attempted to return a reference to a temporary object (perhaps the result of a constructor or a function call). Since this object will disappear when the function returns, the reference will then be illegal.

Compile-time error

Attempting to return a reference to local variable *identifier*

This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal, since the variable referred to disappears when the function exits. You may return a reference to any static or global variable, or you may change the function to return a value instead.

Linker error

Automatic data segment exceeds 64K

The sum of the DGROUP physical segment, local heap, and stack exceeded 64K. Either specify smaller values for the HEAPSIZE and STACKSIZE statements in the module definition file, or decrease the size of your near data in DGROUP. The map file will show the sizes of the component segments in DGROUP.

Compile-time fatal error

Bad call of intrinsic function

You have used an intrinsic function without supplying a prototype, or you supplied a prototype for an intrinsic function that was not what the compiler expected.

Linker fatal error

Bad character in parameters

One of the following characters was encountered in the command line or in a response file:

" * < = > ? [ ] |

or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.

Compile-time error

Bad define directive syntax

A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.

Compile-time error

Bad file name format in include directive

Include file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.

Compile-time error

Bad file name format in line directive

Line directive file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.

Librarian warning

bad GCD type in GRPDEF, extended dictionary aborted
bad GRPDEF type encountered, extended dictionary aborted

The librarian has encountered an invalid entry in a group definition (GRPDEF) record in an object module while creating an extended dictionary. The only type of GRPDEF record that the librarian supports are segment index type. If any other type of GRPDEF is encountered, the librarian won't be able to create an extended dictionary. It's possible that an object module created by products other than Borland tools may create GRPDEF records of other types. It's also possible for a corrupt object module to generate this warning.

Compile-time error

Bad ifdef directive syntax

An #**ifdef** directive must contain a single identifier (and nothing else) as the body of the directive.

Linker fatal error

Bad object file record in library file *filename* in module *module* near module file offset 0x*xxxxxxxx*
Bad object file record in module *filename* near module file offset 0x*xxxxxxxx*

An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was pressed.

Librarian error

bad OMF record type *type* encountered in module *module*

The librarian encountered a bad Object Module Format (OMF) record while reading through the object module. The librarian has already read and verified the header records on the *module*, so this usually indicates that the object module has become corrupt in some way and should be recreated.

Compile-time error

Bad syntax for pure function definition

Pure virtual functions are specified by appending "= 0" to the declaration. You wrote something similar, but not quite the same.

Compile-time error

Bad undef directive syntax

An **#undef** directive must contain a single identifier (and nothing else) as the body of the directive.

Linker fatal error

Bad version number in parameter block

This error indicates an internal inconsistency in the IDE. If it occurs, exit and restart the IDE. This error will not occur in the standalone version.

Compile-time error

Base class *class* contains dynamically dispatchable functions

Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance. This error occurs because a class which contains DDVT function attempted to inherit DDVT functions from multiple parent classes.

Compile-time warning

Base class *class* is inaccessible because also in *class*

It is not legal to use a class as both a direct and indirect base class, since the members are automatically ambiguous. Try making the base class virtual in both locations.

Compile-time error

Base class *class* is included more than once

A C++ class may be derived from any number of base classes, but may be directly derived from a given class only once.

Compile-time error

Base class *class* is initialized more than once

In a C++ class constructor, the list of initializations following the constructor header includes base class *class* more than once.

Compile-time error

Base initialization without a class name is now obsolete

Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list. It is now recommended to include the base class name.

This makes the code much clearer, and is required when there are multiple base classes.

Old way:

```
derived::derived(int i) : (i, 10) { ... }
```

New way:

```
derived::derived(int i) : base(i, 10) { ... }
```

Compile-time error

Bit field cannot be static

Only ordinary C++ class data members can be declared **static**, not bit fields.

Compile-time error

Bit field too large

This error occurs when you supply a bit field with more than 16 bits.

Compile-time error

Bit fields must be signed or unsigned int

In ANSI C, bit fields may only be signed or unsigned **int** (not **char** or **long**, for example).

Compile-time warning

Bit fields must be signed or unsigned int

In ANSI C, bit fields may not be of type signed char or unsigned char; when not compiling in strict ANSI mode, though, the compiler will allow such constructs, but flag them with this warning.

Compile-time error

Bit fields must contain at least one bit

You cannot declare a named bit field to have 0 (or less than 0) bits. You can declare an unnamed bit field to have 0 bits, a convention used to force alignment of the following bit field to a byte boundary (or word boundary, if you select IDE Options | Compiler | Code Generation | Word Alignment). In C++, bit fields must have an integral type; this includes enumerations.

Compile-time error

Bit fields must have integral type

In C++, bit fields must have an integral type; this includes enumerations.

Compile-time error

Body has already been defined for function *function*

A function with this name and type was previously supplied a function body. A function body can only be supplied once.

Compile-time warning

Both return and return with a value used

The current function has **return** statements with and without values. This is legal in C, but almost always an error. Possibly a **return** statement was omitted from the end of the function.

Compile-time error

Call of nonfunction

The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or misspelling the function name.

Compile-time warning

Call to function *function* with no prototype

The "Prototypes required" warning was enabled and you called function *function* without first giving a prototype for that function.

Compile-time error

Cannot add or subtract relocatable symbols

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that *Var* is a variable and *Const* is a constant, then the instructions

```
MOV AX,Const+Const
```

and

```
MOV AX,Var+Const
```

are valid, but `MOV AX,Var+Var` is not.

Compile-time error

Cannot allocate a reference

An attempt to create a reference using the **new** operator has been made; this is illegal, as references are not objects and cannot be created through **new**.

Compile-time error

***identifier* cannot be declared in an anonymous union**

The compiler found a declaration for a member function or static member in an anonymous union. Such unions can only contain data members.

Compile-time error

function1* cannot be distinguished from *function2

The parameter type lists in the declarations of these two functions do not differ enough to tell them apart. Try changing the order of parameters or the type of a parameter in one declaration.

Compile-time error

Cannot call near class member function with a pointer of type *type*

Member functions of near classes (remember that classes are near by default in the tiny, small, and medium memory models) cannot be called using far or huge member pointers. (Note that this also applies to calls using pointers to members.) Either change the pointer to be near, or declare the class as far.

Compile-time error

Cannot cast from *type1* to *type2*

A cast from type *type1* to type *type2* is not allowed. In C, a pointer may be cast to an integral type or to another pointer. An integral type may be cast to any integral, floating, or pointer type. A floating type may be cast to an integral or floating type. Structures and arrays may not be cast to or from. You cannot cast from a **void** type.

C++ checks for user-defined conversions and constructors, and if one cannot be found, then the preceding rules apply (except for pointers to class members). Among integral types, only a constant zero may be cast to a member pointer. A member pointer may be cast to an integral type or to a similar member pointer. A similar member pointer points to a data member if the original does, or to a function member if the original does; the qualifying class of the type being cast to must be the same as or a base class of the original.

Compile-time error

Cannot convert *type1* to *type2*

An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal.

Compile-time error

Cannot create instance of abstract class *class*

Abstract classes—those with pure virtual functions—cannot be used directly, only derived from.

Compile-time error

Cannot define a pointer or reference to a reference

It is illegal to have a pointer to a reference or a reference to a reference.

Compile-time error

Cannot find *class*::*class* (*class* &) to copy a vector

When a C++ class *class1* contains a vector (array) of class *class2*, and you want to construct an object of type *class1* from another object of type *class1*, there must be a constructor **class2::class2(class2&)** so that the elements of the vector can be constructed. This constructor takes just one parameter (which is a reference to its class) and is called a *copy constructor*.

Usually the compiler supplies a copy constructor automatically. However, if you have defined a constructor for class *class2* that has a parameter of type *class2&* and has additional parameters with default values, the copy constructor cannot be created by the compiler. (This is because `class2::class2(class2&)` and `class2::class2(class2&, int = 1)` cannot be distinguished.) You must redefine this constructor so

that not all parameters have default values. You can then define a copy constructor or let the compiler create one.

Compile-time error

Cannot find *class*::operator=(*class*&) to copy a vector

When a C++ class *class1* contains a vector (array) of class *class2*, and you wish to copy a class of type *class1*, there must be an assignment operator **class2::operator=(class2&)** so that the elements of the vector can be copied. Usually the compiler supplies such an operator automatically. However, if you have defined an **operator=** for class *class2*, but not one that takes a parameter of type *class2&*, the compiler will not supply it automatically—you must supply one.

Compile-time error

Cannot find default constructor to initialize array element of type *class*

When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor (it will be used to initialize the array elements that don't have explicit initializers). The compiler will define a default constructor for a class unless you have defined any constructors for the class.

Compile-time error

Cannot find default constructor to initialize base class *class*

Whenever a C++ derived class *class2* is constructed, each base class *class1* must first be constructed. If the constructor for *class2* does not specify a constructor for *class1* (as part of *class2*'s header), there must be a constructor **class1::class1()** for the base class. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class1*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

Compile-time error

Cannot find default *constructor* to initialize member *identifier*

When a C++ class *class1* contains a member of class *class2*, and you wish to construct an object of type *class1* but not from another object of type *class1*, there must be a constructor **class2::class2()** so that the member can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class2*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

Compile-time error

Cannot generate *function* from template function *template*

A call to a template function was found, but a matching template function cannot be generated from the function template.

Compile-time error

Cannot have a near class member in a far class

All members of a C++ **far** class must be far. This member is in a class that was declared (or defaults to) **near**.

Compile-time error

Cannot have a non-inline function in a local class
Cannot have a static data member in a local class

All members of classes declared local to a function must be entirely defined in the class definition. This means that such local classes may not contain any static data members, and all of their member functions must have bodies defined within the class definition.

Compile-time error

Cannot initialize a class member here

Individual members of **struct**s, **union**s, and C++ **class**es may not have initializers. A **struct** or **union** may be initialized as a whole using initializers inside braces. A C++ **class** may only be initialized by the use of a constructor.

Compile-time error

Cannot initialize *type1* with *type2*

You are attempting to initialize an object of type *type1* with a value of type *type2*, which is not allowed. The rules for initialization are essentially the same as for assignment.

Compile-time error

Cannot modify a const object

This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.

Compile-time error

Cannot overload 'main'

main is the only function which cannot be overloaded.

Compile-time error

***function* cannot return a value**

A function with a return type **void** contains a **return** statement that returns a value; for example, an **int**.

Compile-time error

***identifier* cannot start an argument declaration**

Undefined *identifier* found at the start of an argument in a function declarator. Often the type name is misspelled or the type declaration is missing (usually caused by not including the appropriate header file).

Librarian error

cannot write GRPDEF list, extended dictionary aborted

The librarian cannot write the extended dictionary to the tail end of the library file. This usually indicates lack of space on the disk.

Compile-time error

Case bypasses initialization of a local variable

In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a case label which can transfer control past this local variable.

Compile-time error

Case outside of switch

The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched braces.

Compile-time error

Case statement missing :

A **case** statement must have a constant expression followed by a colon. The expression in the **case** statement either is missing a colon or has an extra symbol before the colon.

Compile-time error

Character constant must be one or two characters long

Character constants can be only one or two characters long.

Compile-time error

Class *class* may not contain pure functions

The class being declared cannot be abstract, and therefore it may not contain any pure functions.

Compile-time error

Class member *member* declared outside its class

C++ class member functions can be declared only inside the class declaration. Unlike nonmember functions, they cannot be declared multiple times or at other locations.

Compile-time warning

Code has no effect

The compiler encountered a statement with operators that have no effect. For example the statement

```
a + b;
```

has no effect on either variable. The operation is unnecessary and probably indicates a bug in your file.

Linker error

Common segment exceeds 64K

The program had more than 64K of near uninitialized data. Try declaring some uninitialized data as far.

Compile-time error

Compiler could not generate copy constructor for class *class*

The compiler cannot generate a needed copy constructor due to language rules.

Compile-time error

Compiler could not generate default constructor for class *class*

The compiler cannot generate a needed default constructor due to language rules.

Compile-time error

Compiler could not generate operator= for class *class*

The compiler cannot generate a needed assignment operator due to language rules.

Compile-time fatal error

Compiler table limit exceeded

One of the compiler's internal tables overflowed. This usually means that the module being compiled contains too many function bodies. Making more memory available to the compiler will not help with such a limitation; simplifying the file being compiled is usually the only remedy.

Compile-time error

Compound statement missing }

The compiler reached the end of the source file and found no closing brace. This is often caused by mismatched braces.

Compile-time warning

Condition is always false
Condition is always true

The compiler encountered a comparison of values where the result is always true or false. For example:

```
void proc(unsigned x)
{
   if (x >= 0)        /* always 'true' */
   {
   :
   }
}
```

Compile-time error

Conflicting type modifiers

This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) may be given for a function.

Linker warning

symbol* conflicts with module *module* in module *module

This indicates an inconsistency in the definition of *symbol*; the linker found one virtual function and one common definition with the same name.

Compile-time error

Constant expression required

Arrays must be declared with constant size. This error is commonly caused by misspelling a **#define** constant.

Compile-time warning

Constant is long

The compiler encountered either a decimal constant greater than 32767 or an octal (or hexadecimal) constant greater than 65535 without a letter *l* or *L* following it. The constant is treated as a **long**.

Compile-time error

Constant member *member* in class without constructors

A class that contains constant members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.

Compile-time warning

Constant member *member* is not initialized

This C++ class contains a constant member *member*, which does not have an initialization. Note that constant members may be initialized only, not assigned to.

Compile-time warning

Constant out of range in comparison

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an **unsigned** quantity to –1 makes no sense. To get an **unsigned** constant greater than 32767 (in decimal), you should either cast the constant to **unsigned** (for example, (**unsigned**)65535) or append a letter *u* or *U* to the constant (for example, 65535u).

Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a **char** expression to 4000, the code will still perform the test.

Compile-time error

Constant variable *variable* must be initialized

This C++ object is declared **const**, but is not initialized. Since no value may be assigned to it, it must be initialized at the point of declaration.

Compile-time error

constructor cannot be declared const or volatile

A constructor has been declared as **const** and/or **volatile**, and this is not allowed.

Compile-time error

constructor cannot have a return type specification

C++ constructors have an implicit return type used by the compiler, but you cannot declare a return type or return a value.

Compile-time warning

Conversion may lose significant digits

For an assignment operator or some other circumstance, your source file requires a conversion from **long** or **unsigned long** to

int or **unsigned int** type. Since **int** type and **long** type variables don't have the same size, this kind of conversion may alter the behavior of a program.

Compile-time error

Conversion operator cannot have a return type specification

This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function **operator** may not specify any return type.

Compile-time error

Conversion to *type* will fail for members of virtual base *class*

This warning is issued in some cases when a member pointer is cast to another member pointer type, if the class of the member pointer contains virtual bases, and only if the IDE Options | Compiler | Advanced Compiler | Deep Virtual Bases has been used. It means that if the member pointer being cast happens to point (at the time of the cast) to a member of ***class***, the conversion cannot be completed, and the result of the cast will be a NULL member pointer. (See the *User's Guide* for details).

Librarian error

could not allocate memory for per module data

The librarian has run out of memory.

Librarian error

could not create list file *filename*

The librarian could not create a list file for the library. This could be due to lack of disk space.

Compile-time error

Could not find a match for *argument(s)*

No C++ function could be found with parameters matching the supplied arguments.

Compile-time error

Could not find file *filename*

The compiler is unable to find the file supplied on the command line.

Librarian error

Could not write output.

The librarian could not write the output file.

Librarian error

couldn't alloc memory for per module data

The librarian has run out of memory.

Librarian warning

***filename* couldn't be created, original won't be changed**

An attempt has been made to extract an object ('*' action) but the librarian cannot create the object file to extract the module into. Either the object already exists and is read only, or the disk is full.

Linker warning

Debug information in module *module* will be ignored

Object files compiled with debug information now have a version record. The major version of this record is higher than

what the linker currently supports and the linker did not generate debug information for the module in question.

Compile-time error

Declaration does not specify a tag or an identifier

This declaration doesn't declare anything. This may be a **struct** or **union** without a tag or a variable in the declaration. C++ requires that something be declared.

Compile-time error

Declaration is not allowed here

Declarations cannot be used as the control statement for **while**, **for**, **do**, **if**, or **switch** statements.

Compile-time error

Declaration missing ;

Your source file contained a declaration that was not followed by a semicolon.

Compile-time error

Declaration syntax error

Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.

Compile-time error

Declaration terminated incorrectly

A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body. A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.

Compile-time error

Declaration was expected

A declaration was expected here but not found. This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.

Compile-time error

Declare operator delete (void*) or (void*, size_t)

Declare the operator **delete** with a single **void*** parameter, or with a second parameter of type **size_t**. If you use the second version, it will be used in preference to the first version. The global operator **delete** can only be declared using the single-parameter form.

Compile-time warning

Declare type *type* prior to use in prototype

When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype. For example,

```
int func(struct s *ps);
struct s { /* ... */ };
```

Since there is no struct *s* in scope at the prototype for **func**, the type of parameter *ps* is pointer to undefined struct *s*, and is not

the same as the struct *s* which is later declared. This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message. To fix the problem, you can move the declaration for struct *s* ahead of any prototype which references it, or add the incomplete type declaration `struct s;` ahead of any prototype which references struct *s*. If the function parameter is a **struct**, rather than a pointer to **struct**, the incomplete declaration is not sufficient; you must then place the struct declaration ahead of the prototype.

Compile-time warning

identifier is declared but never used

Your source file declared the named variable as part of the block just ending, but the variable was never used. The warning is indicated when the compiler encounters the closing brace of the compound statement or function. The declaration of the variable occurs at the beginning of the compound statement or function.

Compile-time error

Default argument value redeclared for parameter _parameter_

When a parameter of a C++ function is declared to have a default value, this value cannot be changed, redeclared, or omitted in any other declaration for the same function.

Compile-time error

Default expression may not use local variables

A default argument expression is not allowed to use any local variables or other parameters.

Compile-time error

Default outside of switch

The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched braces.

Compile-time error

Default value missing

When a C++ function declares a parameter with a default value, all of the following parameters must also have default values. In this declaration, a parameter with a default value was followed by a parameter without a default value.

Compile-time error

Default value missing following parameter _parameter_

All parameters following the first parameter with a default value must also have defaults specified.

Compile-time error

Define directive needs an identifier

The first non-whitespace character after a #**define** must be an identifier. The compiler found some other character.

Linker error or warning

symbol* defined in module *module* is duplicated in module *module

There is a conflict between two symbols (either public or communal). This usually means that a symbol is defined in two modules. An error occurs if both are encountered in the .OBJ file(s), because the linker doesn't know which is valid. A warning results if the linker finds one of the duplicated symbols in a library and finds the other in an .OBJ file; in this case, the linker uses the one in the .OBJ file.

D

Compile-time error

Delete array size missing]

The array specifier in an operator date is missing a right bracket.

Compile-time error

Destructor cannot be declared const or volatile

A destructor has been declared as **const** and/or **volatile**, and this is not allowed.

Compile-time error

Destructor cannot have a return type specification

It is illegal to specify the return type for a destructor.

Compile-time error

Destructor for *class* is not accessible

The destructor for this C++ class is **protected** or **private**, and cannot be accessed here to destroy the class. If a class destructor is **private**, the class cannot be destroyed, and thus can never be used. This is probably an error. A **protected** destructor can be accessed only from derived classes. This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.

Compile-time error

Destructor for *class* required in conditional expression

If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor, since the variable may or may not have been initialized. The temporary variable can be explicitly created, as with classname (val, val), or implicitly created by some other code. Recast your code to eliminate this temporary value.

Compile-time error

Destructor name must match the class name

In a C++ class, the tilde (~) introduces a declaration for the class destructor. The name of the destructor must be the same as the class name. In your source file, the tilde (~) preceded some other name.

Run-time error

Divide error

You've tried to divide an integer by zero. For example,

```
int n = 0;
n = 2 / n;
```

You can trap this error with the signal function. Otherwise, Turbo C++ calls **abort** and your program terminates.

Compile-time error

Division by zero

Your source file contained a division or remainder operator in a constant expression with a zero divisor.

Compile-time warning

Division by zero

A division or remainder operator expression had a literal zero as a divisor.

Compile-time error

do statement must have while

Your source file contained a **do** statement that was missing the closing **while** keyword.

Linker fatal error

DOS error, ax = *number*

This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a Linker internal error or a DOS error. The only DOS calls the linker makes where this error could occur are read, write, seek, and close.

Compile-time error

do-while statement missing (

In a **do** statement, the compiler found no left parenthesis after the **while** keyword.

Compile-time error

do-while statement missing)

In a **do** statement, the compiler found no right parenthesis after the test expression.

Compile-time error

do-while statement missing ;

In a **do** statement test expression, the compiler found no semicolon after the right parenthesis.

Compile-time error

Duplicate case

Each **case** of a **switch** statement must have a unique constant expression value.

Linker warning

***filename* (*linenum*): Duplicate external name in exports**

Two export functions listed in the EXPORTS section of a module definition file defined the same external name. For instance,

```
EXPORTS
   AnyProc=MyProc1
   AnyProc=MyProc2
```

Linker warning

filename (_linenum_): Duplicate internal name in exports

Two export functions listed in the EXPORTS section of the module definition file defined the same internal name. For example,

```
EXPORTS
   AnyProc1=MyProc
   AnyProc2=MyProc
```

Linker warning

filename (_linenum_): Duplicate internal name in imports

Two import functions listed in the IMPORTS section of the module definition file defined the same internal name. For instance,

```
IMPORTS
   AnyProc=MyMod1.MyProc1
   AnyProc=MyMod2.MyProc2
```

or

```
IMPORTS
   MyMod1.MyProc
   MyMod2.MyProc]
```

Linker warning

Duplicate ordinal _number_ in exports

This warning occurs when the linker encounters two exports with the same ordinal value. First check the module definition file to ensure that there are no duplicate ordinal values specified in the EXPORTS section. If not, then you are linking with modules which specify exports by ordinals and one of two things happened: either two export records specify the same ordinal, or the exports section in the module definition file duplicates an ordinal in an export record.

Export records (EXPDEF) are comment records found in object files and libraries which specify that particular variables are to be exported. Optionally, these records can specify ordinal values when exporting by ordinal (rather than by name).

Compile-time error

Enum syntax error

An **enum** declaration did not contain a properly formed list of identifiers.

Compile-time fatal error

Error directive: _message_

The text of the #**error** directive being processed in the source file is displayed.

Compile-time fatal error

Error writing output file

A DOS error that prevents Turbo C++ from writing an .OBJ, .EXE, or temporary file. Check the Options | Directories | Output directory and make sure that this is a valid directory. Also check that there is enough free disk space.

Compile-time error

Expression expected

An expression was expected here, but the current symbol cannot begin an expression. This message may occur where the controlling expression of an **if** or **while** clause is expected or where a variable is being initialized. It is often due to an accidentally inserted or deleted symbol in the source code.

Compile-time error

Expression of scalar type expected

The not (!), increment (++), and decrement (—) operators require an expression of scalar type—only types **char**, **short**, **int**, **long**, **enum**, **float**, **double**, **long double**, and pointer types are allowed.

Compile-time error

Expression syntax

This is a catchall error message when the compiler parses an expression and encounters some serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

Librarian warning

reason **– extended dictionary not created**

The librarian could not produce the extended dictionary because of the *reason* given in the warning message.

Compile-time error

extern variable cannot be initialized

The storage class **extern** applied to a variable means that the variable is being declared but not defined here—no storage is being allocated for it. Therefore, you can't initialize the variable as part of the declaration.

Compile-time error

Extra argument in template class name ***template***

A template class name specified too many actual values for its formal parameters.

Compile-time error

Extra parameter in call

A call to a function, via a pointer defined with a prototype, had too many arguments given.

Compile-time error

Extra parameter in call to ***function***

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

Compile-time error

File must contain at least one external declaration

This compilation unit was logically empty, containing no external declarations. ANSI C and C++ require that something be declared in the compilation unit.

Compile-time error

File name too long

The file name given in an **#include** directive was too long for the compiler to process. Path names in DOS must be no more than 79 characters long.

Librarian error

***filename* file not found**

The IDE creates the library by first removing the existing library and then rebuilding. If any objects do not exist, the library is considered incomplete and thus an error. If the IDE reports that an object does not exist, either the source module has not been compiled or there were errors during compilation. Performing either a Compile | Make or Compile | Build should resolve the problem or indicate where the errors have occurred.

Linker fatal error

***filename* (*linenum*): File read error**

A DOS error occurred while the linker read the module definition file. This usually means that a premature end of file occurred.

Linker error

Fixup overflow at *segment*:*xxxx*h, target = *segment*:*xxx*h in module *module*

Fixup overflow at *segment*:*xxxx*h, target = *symbol* in module *module*

Either of these messages indicate an incorrect data or code reference in an object file that the linker must fix up at link time.

The cause is often a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named module, so look in the source file of that module for the offending reference.

In an assembly language program, a fixup overflow frequently occurs if you have declared an external variable within a segment definition, but this variable actually exists in a different segment.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or a high-level language besides Turbo C++, there may be other possible causes for this message. Even in Turbo C++, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Run-time error

Floating point error: Divide by 0.
Floating point error: Domain.
Floating point error: Overflow.

These fatal errors result from a floating-point operation for which the result is not finite.

- "Divide by 0" means the result is +INF or –INF exactly, such as 1.0/0.0.
- "Domain" means the result is NAN (not a number), like 0.0 /0.0.
- "Overflow" means the result is +INF (infinity) or –INF with complete loss of precision, such as assigning 1e200*1e200 to a **double**.

Run-time error

Floating point error: Partial loss of precision.
Floating point error: Underflow.

These exceptions are masked by default, and the error messages do not occur. Underflows are converted to zero and losses of precision are ignored. They can be unmasked by calling **_control87**.

Run-time error

Floating point error: Stack fault.

The floating-point stack has been overrun. This error does not normally occur and may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

These floating-point errors can be avoided by masking the exception so that it doesn't occur, or by catching the exception with **signal**.

In each of the above cases, the program prints the error message and then calls **abort**, which prints

```
Abnormal program termination
```

and calls `_exit(3)`.

Compile-time error

For statement missing (

In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

Compile-time error

For statement missing)

In a **for** statement, the compiler found no right parenthesis after the control expressions.

Compile-time error

For statement missing ;

In a **for** statement, the compiler found no semicolon after one of the expressions.

Compile-time error

Friends must be functions or classes

A **friend** of a C++ class must be a function or another class.

Compile-time error

Function call missing)

The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

Compile-time error

Function defined inline after use as extern

Functions cannot become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely.

Compile-time error

Function definition cannot be a Typedef'ed declaration

In ANSI C a function body cannot be defined using a typedef with a function Type.

Compile-time error

Function *function* cannot be static

Only ordinary member functions and the operators **new** and **delete** can be declared static. Constructors, destructors and other operators must not be static.

Compile-time error

Function *function* should have a prototype

A function was called with no prototype in scope.

In C, `int foo();` is not a prototype, but `int foo(int);` is, and so is `int foo(void);`. In C++, `int foo();` is a prototype, and is the same as `int foo(void);`. In C, prototypes are *recommended* for all functions. In C++, prototypes are *required* for all functions. In all cases, a function definition (a function header with its body) serves as a prototype if it appears before any other mention of the function.

Compile-time warning

Function should return a value

This function was declared (maybe implicitly) to return a value. A **return** statement was found without a return value or the end of the function was reached without a **return** statement being found. Either return a value or declare the function as **void**.

Compile-time error

Function should return a value

Your source file declared the current function to return some type other than **void** in C++ (or **int** in C), but the compiler encountered a return with no value. This is usually some sort of error. In C **int** functions are exempt, since in old versions of C there was no **void** type to indicate functions which return nothing.

Compile-time error

Functions *function1* and *function2* both use the same dispatch number

Dynamically dispatched virtual table (DDVT) problem.

Compile-time warning

Functions containing local destructors are not expanded inline in function *function*

You've created an inline function for which Turbo C++ turns off inlining. You can ignore this warning if you like; the function will be generated out of line.

Compile-time warning

Functions containing *reserved word* are not expanded inline

Functions containing any of the reserved words **do**, **for**, **while**, **goto**, **switch**, **break**, **continue**, and **case** cannot be expanded inline, even when specified as **inline**. The function is still perfectly legal, but will be treated as an ordinary static (not global) function.

Compile-time error

Functions may not be part of a struct or union

This C **struct** or **union** field was declared to be of type function rather than pointer to function. Functions as fields are allowed only in C++.

Linker fatal error

General error
General error in library file *filename* in module *module* near module file offset 0x*yyyyyyyy*.
General error in module *module* near module file offset 0x*yyyyyyyy*

The linker gives as much information as possible about what processing was happening at the time of the unknown fatal error.

Compile-time error

Global anonymous union not static

In C++, a global anonymous union at the file level must be static.

Compile-time error

Goto bypasses initialization of a local variable

In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a **goto** which can transfer control past this local variable.

Compile-time error

Goto statement missing label

The **goto** keyword must be followed by an identifier.

Linker fatal error

Group *group* exceeds 64K

A group exceeded 64K bytes when the segments of the group were combined.

Compile-time error

Group overflowed maximum size: *group*

The total size of the segments in a group (for example, DGROUP) exceeded 64K.

Linker warning

Group *group1* overlaps group *group2*

This means that the linker has encountered nested groups.

Compile-time error

***specifier* has already been included**

This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.

Compile-time warning

Hexadecimal value contains more than 3 digits

Under older versions of C, a hexadecimal escape sequence could contain no more than three digits. The ANSI standard allows any number of digits to appear as long as the value fits in a byte. This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as "\x00045"). Older versions of C would interpret such a string differently.

Compile-time warning

function1* hides virtual function *function2

A virtual function in a base class is usually overridden by a declaration in a derived class. In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.

Compile-time error

Identifier expected

An identifier was expected here, but not found. In C, this is in a list of parameters in an old-style function header, after the reserved words **struct** or **union** when the braces are not present, and as the name of a member in a structure or union (except for bit fields of width 0). In C++, an identifier is also expected in a list of base classes from which another class is derived, following a double colon (::), and after the reserved word **operator** when no operator symbol is present.

Compile-time error

Identifier *identifier* cannot have a type qualifier

A C++ qualifier *class::identifier* may not be applied here. A qualifier is not allowed on **typedef** names, on function declarations (except definitions at the file level), on local variables or parameters of functions, or on a class member except to use its own class as a qualifier (redundant but legal).

Compile-time error

If statement missing (

In an **if** statement, the compiler found no left parenthesis after the **if** keyword.

Compile-time error

If statement missing)

In an **if** statement, the compiler found no right parenthesis after the test expression.

Compile-time error

Illegal character *character* (0x*value*)

The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed. This can also be caused by extra parameters passed to a function macro.

Linker fatal error

Illegal group definition: *group* in module *module*

This error results from a malformed GRPDEF record in an .OBJ file. This latter case could result from custom-built .OBJ files or a bug in the translator used to generate the .OBJ file. If this occurs in a file created by Turbo C++, recompile the file. If the error persists, contact Borland.

Compile-time error

Illegal initialization

In C, initializations must be either a constant expression, or else the address of a global **extern** or **static** variable plus or minus a constant.

Compile-time error

Illegal octal digit

An octal constant was found containing a digit of 8 or 9.

Compile-time error

Illegal parameter to __emit__

You supplied an argument to **emit** which is not a constant or an address.

Compile-time error

Illegal pointer subtraction

This is caused by attempting to subtract a pointer from a non-pointer.

Compile-time error

Illegal structure operation

In C or C++, structures may be used with dot (.), address-of (**&**), or assignment (**=**) operators, or be passed to or from

functions as parameters. In C or C++, structures can also be used with overloaded operators. The compiler encountered a structure being used with some other operator.

Compile-time error

Illegal to take address of bit field

It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.

Compile-time error

Illegal use of floating point

Floating-point operands are not allowed in shift, bitwise Boolean, indirection (*), or certain other operators. The compiler found a floating-point operand with one of these prohibited operators.

Compile-time error

Illegal use of member pointer

Pointers to class members can only be used with assignment, comparison, the **.***, **–>***, **?:**, **&&** and **||** operators, or passed as arguments to functions. The compiler has encountered a member pointer being used with a different operator.

Compile-time error

Illegal use of pointer

Pointers can only be used with addition, subtraction, assignment, comparison, indirection (*) or arrow (**->**). Your source file used a pointer with some other operator.

Compile-time warning

Ill-formed pragma

A pragma does not match one of the pragmas expected by the Turbo C++ compiler.

Compile-time error

Implicit conversion of *type1* to *type2* not allowed

When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to point to the appropriate base class. In this case, such an implicit conversion is illegal.

Linker error

Imported reference from a VIRDEF to *symbol*

The linker does not currently support references from VIRDEFs to symbols that are imported from dynamically-linked libraries. If you have an inline function in an executable which references a static data member of a class in a DLL, take the function out of the line.

Compile-time error

Improper use of typedef *identifier*

Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.

Linker fatal error

filename (_linenum_): Incompatible attribute

The linker encountered incompatible segment attributes in a CODE or DATA statement. For instance, both PRELOAD and LOADONCALL can't be attributes for the same segment.

Compile-time error

Incompatible type conversion

The cast requested can't be done. Check the types.

Compile-time error

Incorrect configuration file option: _option_

The compiler did not recognize the configuration file parameter as legal; check for a preceding hyphen (–).

Compile-time error

Incorrect number format

The compiler encountered a decimal point in a hexadecimal number.

Compile-time error

Incorrect use of default

The compiler found no colon after the **default** keyword.

Compile-time warning

Initializing enumeration with _type_

You're trying to initialize an **enum** variable to a different type. For example,

```
enum count { zero, one, two } x = 2;
```

will result in this warning, because 2 is of type **int**, not type **enum count**. It is better programming practice to use an **enum** identifier instead of a literal integer when assigning to or initializing **enum** types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

Compile-time error

Inline assembly not allowed in inline and template functions

The compiler cannot handle inline assembly statements in a C++ inline or template function. You could eliminate the inline assembly code or, in case of an inline function, make this a macro, or remove the **inline** storage class.

Linker fatal error

Internal linker error _errorcode_

An error occurred in the internal logic of the linker. This error shouldn't occur in practice, but is listed here for completeness in the event that a more specific error isn't generated. If this error persists, write down the *errorcode* number and contact Borland.

Compile-time error

Invalid combination of opcode and operands

The built-in assembler does not accept this combination of operands. Possible causes are:

- There are too many or too few operands for this assembler opcode; for example, **INC AX,BX**, or **MOV AX**.
- The number of operands is correct, but their types or order do not match the opcode; for example **DEC 1**, **MOV AX,CL**, or **MOV 1,AX**.

Linker warning

Invalid entry at *xxxxh*

This error indicates that a necessary entry was missing from the entry table of a Windows executable file. The application may not work in real mode unless you fix the code and data.

Linker error

Invalid entry point offset

This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.

Compile-time error

Invalid indirection

The indirection operator (*) requires a non-**void** pointer as the operand.

Linker fatal error

Invalid initial stack offset

This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.

Linker fatal error

Invalid limit specified for code segment packing

This error occurs if you used IDE Options | Linker | Settings | Pack code segments and specified a size limit that was out of range. To be valid, the size limit must be between 1 and 65536 bytes; the default is 8192.

Compile-time error

Invalid macro argument separator

In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.

Librarian warning

invalid page size value ignored

Invalid page size is given. The page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.

Compile-time error

Invalid pointer addition

Your source file attempted to add two pointers together.

Compile-time error

Invalid register combination (e.g. [BP+BX])

The built-in assembler detected an illegal combination of registers in an instruction. Valid index register combinations are **[BX]**, **[BP]**, **[SI]**, **[DI]**, **[BX+SI]**, **[BX+DI]**, **[BP+SI]**, and **[BP+DI]**.

Other index register combinations (such as **[AX]**, **[BP+BX]**, and **[SI+DX]**) are not allowed.

Local variables (variables declared in procedures and functions) are usually allocated on the stack and accessed via the BP register. The assembler automatically adds **[BP]** in references to such variables, so even though a construct like **Local[BX]** (where **Local** is a local variable) appears valid, it is not since the final operand would become **Local[BP+BX]**.

Linker fatal error

Invalid segment definition in module *module*

The compiler produced a flawed object file. If this occurs in a file created by Turbo C++, recompile the file. If the problem persists, contact Borland.

Linker error

Invalid size specified for segment alignment

This error occurs if an invalid value is specified for the Options | Linker | Settings | Segment alignment option. The value specified must be an integral multiple of 2 and less than 64K. Common values are 16 and 512. This error only occurs when linking for Windows.

Compile-time error

Invalid template argument list

In a template declaration, the keyword **template** must be followed by a list of formal arguments enclosed within the **<** and **>** delimiters; an illegal template argument list was found.

Compile-time error

Invalid template qualified name *template::name*

When defining a template class member, the actual arguments in the template class name that is used as the left operand for the :: operator must match the formal arguments of the template class. For example:

```
template <class T> class X
{
   void  f();
};

template <class T> void X<T>::f(){}
```

The following would be illegal:

```
template <class T> void X<int>::f(){}
```

Compile-time error

Invalid use of dot

An identifier must immediately follow a period operator (.).

Compile-time error

Invalid use of template *template*

Outside of a template definition, it is illegal to use a template class name without specifying its actual arguments. For example, you can use **vector<int>** but not **vector**.

Compile-time fatal error

Irreducible expression tree

This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. Whatever the offending expression is, it should be avoided. Notify Borland if the compiler ever encounters this error.

Compile-time error

base* is an indirect virtual base class of *class

A pointer to a C++ member of the given virtual base class cannot be created; an attempt has been made to create such a pointer (either directly, or through a cast).

Compile-time warning

***identifier* is assigned a value that is never used**

The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing brace.

Compile-time warning

***identifier* is declared as both external and static**

This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration. The identifier is taken as static. You should review all declarations for this identifier.

Linker error or warning

symbol* is duplicated in module *module

There is a conflict between two symbols (either public or communal) defined in the same module. An error occurs if both are encountered in an .OBJ file. A warning is issued if the linker finds the duplicates in a library; in this case, the linker uses the first definition.

Compile-time error

constructor* is not a base class of *class

A C++ class constructor *class* is trying to call a base class constructor *constructor*, or you are trying to change the access rights of *class::constructor*. *constructor* is not a base class of *class*. Check your declarations.

Compile-time error

identifier* is not a member of *struct

You are trying to reference *identifier* as a member of ***struct***, but it is not a member. Check your declarations.

Compile-time error

***identifier* is not a non-static data member and can't be initialized here**

Only data members can be initialized in the initializers of a constructor. This message means that the list includes a static member or function member.

Compile-time error

***identifier* is not a parameter**

In the parameter declaration section of an old-style function definition, *identifier* is declared but is not listed as a parameter. Either remove the declaration or add *identifier* as a parameter.

Compile-time error

identifier* is not a public base class of *classtype

The right operand of a **.***, **–>***, or **::operator** was not a pointer to a member of a class that is either identical to or an unambiguous accessible base class of the left operand's class type.

Compile-time error

***member* is not accessible**

You are trying to reference C++ class member *member*, but it is **private** or **protected** and cannot be referenced from this function. This sometimes happens when attempting to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function. The check for overload resolution is always made before checking for accessibility. If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

Compile-time error

Last parameter of *operator* must have type int

When a postfix **operator++** or **operator**– ± is declared, the last parameter must be declared with the type **int**.

Librarian warning

library contains COMDEF records – extended dictionary not created

An object record being added to a library contains a COMDEF record. This is not compatible with the extended dictionary option.

Librarian error

library too large, restart with library page size *size*

The library being created could not be built with the current library page size. In the IDE, the library page size can be set from the Options | Librarian dialog box.

Linker fatal error

Limit of 254 segments for new executable file exceeded

The new executable file format only allows for 254 segments. Examine the map file. Usually, one of two things cause the problem. If the application is large model, the code segment

packing size could be so small that there are too many code segments. Increasing the code segment packing size could help.

The other possibility is that you have a lot of far data segments with only a few bytes of data in them. The map file will tell you if this is happening. In this case, reduce the number of far data segments.

Compile-time error

Linkage specification not allowed

Linkage specifications such as **extern** "C" are only allowed at the file level. Move this function declaration out to the file level.

Linker fatal error

Linker stack overflow

The linker uses a recursive procedure for marking modules to be included in an executable image from libraries. This procedure can cause stack overflows in extreme circumstances. If you get this error message, remove some modules from libraries, include them with the object files in the link, and try again.

Compile-time error

Lvalue required

The left hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.

Compile-time error

Macro argument syntax error

An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.

Compile-time error

Macro expansion too long

A macro cannot expand to more than 4,096 characters.

Compile-time error

Matching base class function for *function* has different dispatch number.

If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.

Compile-time error

Matching base class function for *function* is not dynamic

If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.

Compile-time error

Member function must be called or its address taken

When a member function is used in an expression, either it must be called, or its address must be taken using the **&**

operator. In this case, a member function has been used in an illegal context.

Compile-time error

Member identifier expected

The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (**->**) operator must be the name of a member in the structure or class on the left of the operator.

Compile-time error

Member is ambiguous: *member1* and *member2*

You must qualify the member reference with the appropriate base class name. In C++ class *class*, member *member* can be found in more than one base class, and was not qualified to indicate which was meant. This happens only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path. The C++ language rules require that this test for ambiguity be made before checking for access rights (**private, protected, public**). It is therefore possible to get this message even though only one (or none) of the members can be accessed.

Compile-time error

Member *member* cannot be used without an object

This means that the user has written *class*::*member* where *member* is an ordinary (non-static) member, and there is no class to associate with that member. For example, it is legal to write *obj.class*::*member*, but not to write *class*::*member*.

Compile-time error

Member *member* has the same name as its class

A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class. Only a member function or a non-static member may have a name that is identical to its class.

Compile-time error

Member *member* is initialized more than once

In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

Compile-time error

Member pointer required on right side of .* or –>*

The right side of a C++ dot-star (**.***) or an arrow-star (**->***) operator must be declared as a pointer to a member of the class specified by the left side of the operator. In this case, the right side is not a member pointer.

Librarian warning

Memory full listing truncated!

The librarian has run out of memory creating a library listing file. A list file will be created but is not complete.

Compile-time error

Memory reference expected

The built-in assembler requires a memory reference. Most likely you have forgotten to put square brackets around an index register operand; for example, **MOV AX,BX+SI** instead of **MOV AX,[BX+SI]**.

Compile-time error

Misplaced break

The compiler encountered a **break** statement outside a **switch** or looping construct.

Compile-time error

Misplaced continue

The compiler encountered a **continue** statement outside a looping construct.

Compile-time error

Misplaced decimal point

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

Compile-time error

Misplaced elif directive

The compiler encountered an **#elif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

Compile-time error

Misplaced else

The compiler encountered an **else** statement without a matching **if** statement. An extra **else** statement could cause this message, but it could also be caused by an extra semicolon, missing braces, or some syntax error in a previous **if** statement.

Compile-time error

Misplaced else directive

The compiler encountered an **#else** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

Compile-time error

Misplaced endif directive

The compiler encountered an **#endif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

Linker fatal error

***filename* (*linenum*): Missing internal name**

In the IMPORTS section of the module definition file there was a reference to an entry specified via module name and ordinal number. When an entry is specified by ordinal number an internal name must be assigned to this import definition. It is this internal name that your program uses to refer to the imported definition. The syntax in the module definition file should be:

```
<internalname>=<modulename>.<ordinal>
```

Compile-time warning

Mixing pointers to signed and unsigned char

You converted a **signed char** pointer to an **unsigned char** pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)

Compile-time error

Multiple base classes require explicit class names

In a C++ class constructor, each base class constructor call in the constructor header must include the base class name when there is more than one immediate base class.

Compile-time error

Multiple declaration for *identifier*

This identifier was improperly declared more than once. This might be caused by conflicting declarations such as `int a; double a;`, a function declared two different ways, or a label repeated in the same function, or some declaration repeated other than an **extern** function or a simple variable (in C).

Compile-time error

***identifier* must be a member function**

Most C++ operator functions may be members of classes or ordinary nonmember functions, but certain ones are required to be members of classes. These are **operator =**, **operator –>**, **operator ()**, and type conversions. This operator function is not a member function but should be.

Compile-time error

***identifier* must be a member function or have a parameter of class type**

Most C++ operator functions must have an implicit or explicit parameter of class type. This operator function was declared outside a class and does not have an explicit parameter of class type.

Compile-time error

***identifier* must be a previously defined class or struct**

You are attempting to declare *identifier* to be a base class, but either it is not a class or it has not yet been fully defined. Correct the name or rearrange the declarations.

Compile-time error

***identifier* must be a previously defined enumeration tag**

This declaration is attempting to reference *identifier* as the tag of an **enum** type, but it has not been so declared. Correct the name, or rearrange the declarations.

Compile-time error

***function* must be declared with no parameters**

This C++ operator function was incorrectly declared with parameters.

Compile-time error

***function* must be declared with one parameter**

This C++ operator function was incorrectly declared with more than one parameter.

Compile-time error

***operator* must be declared with one or no parameters**

When **operator++** or **operator – –** is declared as a member function, it must be declared to take either no parameters (for the prefix version of the operator) or one parameter of type **int** (for the postfix version).

Compile-time error

***operator* must be declared with one or two parameters**

When **operator++** or **operator – –** is declared as a nonmember function, it must be declared to take either one parameter (for the prefix version of the operator) or two parameters (the postfix version).

Compile-time error

***function* must be declared with two parameters**

This C++ operator function was incorrectly declared with other than two parameters.

Compile-time error

Must take address of a memory location

Your source file used the address-of operator (**&**) with an expression which cannot be used that way; for example, a register variable (in C).

Compile-time error

Need an identifier to declare

In this context, an identifier was expected to complete the declaration. This might be a **typedef** with no name, or an extra semicolon at file level. In C++, it might be a class name improperly used as another kind of identifier.

Compile-time error

No : following the ?

The question mark (**?**) and colon (**:**) operators do not match in this expression. The colon may have been omitted, or parentheses may be improperly nested or missing.

Linker warning

No automatic data segment

No group named DGROUP was found. Because the Turbo C++ initialization files define DGROUP, you will only see this error if you don't link with an initialization file and your program doesn't define DGROUP. Windows uses DGROUP to find the local data segment. The DGROUP is required for Windows applications (but not DLLs) unless DATA NONE is specified in the module definition file.

Compile-time error

No base class to initialize

This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes. Check your declarations.

Compile-time warning

No declaration for function *function*

You called a function without first declaring that function. In C, you can declare a function without presenting a prototype, as in "int func();". In C++, every function declaration is also a prototype; this example is equivalent to "int func(void);". The declaration can be either classic or modern (prototype) style.

Compile-time error

No file name ending

The file name in an **#include** statement was missing the correct closing quote or angle bracket.

Linker warning

No module definition file specified: using defaults

The linker was invoked with one of the Windows options, but no module definition file was specified.

Linker warning

No program starting address defined

This warning means that no module defined the initial starting address of the program. This is almost certainly caused by forgetting to link in the initialization module C0*x*.OBJ. This warning should not occur when linking a Windows DLL.

Compile-time warning

Non-const function *function* called for const object

A non-**const** member function was called for a **const** object. This is an error, but was reduced to a warning to give existing programs a chance to work.

Compile-time warning

Nonportable pointer comparison

Your source file compared a pointer to a non-pointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.

Compile-time error

Nonportable pointer conversion

An implicit conversion between a pointer and an integral type is required, but the types are not the same size. This cannot be done without an explicit cast. This conversion may not make any sense, so be sure this is what you want to do.

Compile-time warning

Nonportable pointer conversion

A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same. Use an explicit cast if this is what you really meant to do.

Compile-time error

Nontype template argument must be of scalar type

A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.

Compile-time error

Non-virtual function *function* declared pure

Only virtual functions can be declared pure, since derived classes must be able to override them.

Compile-time warning

Non-volatile function *function* called for volatile object

In C++, a class member function was called for a volatile object of the class type, but the function was not declared with "volatile" following the function header. Only a volatile member function may be called for a volatile object.

Compile-time error

Not an allowed type

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

Linker fatal error

Not enough memory

There is not enough memory to run the linker. Try closing one or more applications, then run the linker again.

Librarian error

Not enough memory for command-line buffer

This error occurs when the librarian runs out of memory.

Compile-time error

Numeric constant too large

String and character escape sequences larger than hexadecimal \xFF or octal \377 cannot be generated. Two-byte character constants may be specified by using a second backslash. For example, \x0D\x0A represents a two-byte constant. A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0D" "12345");
```

This prints a carriage return followed by `12345`.

Librarian error

object module *filename* is invalid

The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.

Compile-time error

Objects of type *type* cannot be initialized with {}

Ordinary C structures can be initialized with a set of values inside braces. C++ classes can only be initialized with constructors if the class has constructors, private members, functions or base classes which are virtual.

Compile-time error

Only member functions may be 'const' or 'volatile'

Something other than a class member function has been declared **const** and/or **volatile**.

Compile-time error

Only one of a set of overloaded functions can be "C"

C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you wish to override the compiler's assigning a new name by declaring the function extern "C", you can do this for only one of a set of functions with the same name. (Otherwise the linker would find more than one global function with the same name.)

Compile-time error

Operand of delete must be non-const pointer

It is illegal to delete a constant pointer value using operator **delete**.

Compile-time error

Operator [ ] missing]

The C++ **operator[ ]** was declared as **operator [**. You must add the missing] or otherwise fix the declaration.

Compile-time error

operator –> must return a pointer or a class

The C++ **operator->** function must be declared to either return a class or a pointer to a **class** (or **struct** or **union**). In either case, it must be something to which the **–>** operator can be applied.

Compile-time error

operator delete must return void

This C++ overloaded operator **delete** was declared in some other way.

Compile-time error

Operator must be declared as function

An overloaded operator was declared with something other than function type.

Compile-time error

operator new must have an initial parameter of type size_t

Operator **new** can be declared with an arbitrary number of parameters, but it must always have at least one, which is the amount of space to allocate.

Compile-time error

operator new must return an object of type void *

The C++ overloaded operator **new** was declared another way.

Compile-time error

Operators may not have default argument values

It is illegal for overloaded operators to have default argument values.

Compile-time fatal error

Out of memory

The total working storage is exhausted. Compile the file on a machine with more memory. When running under Windows, close one or more applications to free up memory.

Librarian error

Out of memory

For any number of reasons, Turbo C++ ran out of memory while building the library. For many specific cases a more detailed message is reported, leaving "Out of memory" to be the basic catchall for general low memory situations. When running under Windows, close one or more applications to free up memory.

If this message occurs when public symbol tables grow too large, you must free up memory. In the IDE, some additional memory can be gained by closing editors. When running under Windows, close one or more applications to free up memory.

Linker fatal error

Out of memory

The linker has run out of dynamically allocated memory needed during the link process. This error is a catchall for running into a Linker limit on memory usage. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together. You can try reducing the size of RAM disks and/or disk caches that may be active. If running under Windows, close one or more applications to free up memory.

Librarian error

out of memory creating extended dictionary

The librarian has run out of memory creating an extended dictionary for a library. The library is created but will not have an extended dictionary.

Librarian error

out of memory reading LE/LIDATA record from object module

The librarian is attempting to read a record of data from the object module, but it cannot get a large enough block of memory. If the module that is being added has a large data segment or segments, it is possible that adding the module before any other modules might resolve the problem. By adding the module first, there will be memory available for holding public symbol and module lists later.

Librarian error

Out of space allocating per module debug struct

The librarian ran out of memory while allocating space for the debug information associated with a particular object module.

Removing debugging information from some modules being added to the library might resolve the problem.

Librarian error

Output device is full

The output device is full, usually no space left on the disk.

Compile-time warning

overload is now unnecessary and obsolete

Early versions of C++ required the reserved word **overload** to mark overloaded function names. C++ now uses a "type-safe linkage" scheme, whereby all functions are assumed overloaded unless marked otherwise. The use of **overload** should be discontinued.

Compile-time error

Overloadable operator expected

Almost all C++ operators can be overloaded. The only ones that can't be overloaded are the field-selection dot (**.**), dot-star (**.***), double colon (**::**), and conditional expression (**?:**). The preprocessor operators (# and ##) are not C or C++ language operators and thus cannot be overloaded. Other nonoperator punctuation, such as semicolon (**;**), of course, cannot be overloaded.

Compile-time error

Overloaded *function name* ambiguous in this context

The only time an overloaded function name can be used without actually calling the function is when a variable or parameter of an appropriate type is initialized or assigned. In this case an overloaded function name has been used in some other context.

Compile-time warning

Overloaded prefix 'operator *operator*' used as a postfix operator

With the latest specification of C++, it is now possible to overload both the prefix and postfix versions of the **++** and **– –** operators. To allow older code to compile, whenever only the prefix operator is overloaded, but is used in a postfix context, Turbo C++ uses the prefix operator and issues this warning.

Help project message

P1001 Unable to read file *filename*

The file specified in the project file is unreadable. This is a DOS file error.

Help project message

P1003 Invalid path specified in Root option

The path specified by the Root option cannot be found. The compiler uses the current working directory.

Help project message

P1005 Path and filename exceed limit of 79 characters

The absolute pathname, or the combined root and relative pathname, exceed the DOS limit of 79 characters. The file is skipped.

Help project message

P1007 Root path exceeds maximum limit of 66 characters

The specified root pathname exceeds the DOS limit of 66 characters. The pathname is ignored and the compiler uses the current working directory.

Help project message

P1009 [FILES] section missing

The [Files] section is required. The compilation is aborted.

Help project message

P1011 Option *optionname* previously defined

The specified option was defined previously. The compiler ignores the attempted redefinition.

Help project message

P1013 Project file extension cannot be .HLP

You cannot specify that the compiler use a project file with the .HLP extension. Normally, project files are given the .HPJ extension.

Help project message

P1015 Unexpected end-of-file

The compiler has unexpectedly come to the end of the project file. There might be an open comment in the project file or an included file.

Help project message

P1017 Parameter exceeds maximum length of 128 characters

An option, context name or number, build tag, or other parameter on the specified line exceeds the limit of 128 characters. The line is ignored.

Help project message

P1021 Context number already used in [MAP] section

The context number on the specified line in the project file was previously mapped to a different context string. The line is ignored.

Help project message

P1023 Include statements nested too deeply

The ***#include*** statement on the specified line has exceeded the maximum of five include levels.

Help project message

P1025 Section heading *sectionname* unrecognized

A section heading that is not supported by the compiler has been used. The line is skipped.

Help project message

P1027 Bracket missing from section heading sectionname

The right bracket (]) is missing from the specified section heading. Insert the bracket and compile again.

Help project message

P1029 Section heading missing

The section heading on the specified line is not complete. This error is also reported if the first entry in the project file is not a section heading. The compiler continues with the next line.

Help project message

P1030 Section *sectionname* previously defined

A duplicate section has been found in the project file. The lines under the duplicated section heading are ignored and the compiler continues from the next valid section heading.

Help project message

P1031 Maximum number of build tags exceeded

The maximum number of build tags that can be defined is 30. The excess tags are ignored.

Help project message

P1033 Duplicate build tag in [BUILDTAGS] section

A build tag in the [BUILDTAGS] section has been repeated unnecessarily.

Help project message

P1035 Build tag length exceeds maximum

The build tag on the specified line exceeds the maximum of 32 characters. The compiler skips this entry.

Help project message

P1037 Build tag *tagname* contains invalid characters

Build tags can contain only alphanumeric characters or the underscore (_) character. The line is skipped.

Help project message

P1039 [BUILDTAGS] section missing

The ***BUILD*** option declared a conditional build, but there is no [BuildTags] section in the project file. All topics are included in the build.

Help project message

P1043 Too many tags in Build expression

The Build expression on the specified line has used more than the maximum of 20 build tags. The compiler ignores the line.

Help project message

P1045 [ALIAS] section found after [MAP] section

When used, the [Alias] section must precede the [Map] section in the project file. The [Alias] section is skipped otherwise.

Help project message

P1047 Context string *contextname* already assigned an alias

You cannot do: a=b then a=c<_>(A context string can only have one alias.) The specified context string has previously been aliased in the [Alias] section. The attempted reassignment on this line is ignored.

Help project message

P1049 Alias string aliasname already assigned

You cannot do: a=b then b=c. An alias string cannot, in turn, be assigned another alias.

Help project message

P1051 Context string *contextname* cannot be used as alias string

You cannot do: a=b then c=a. A context string that has been assigned an alias cannot be used later as an alias for another context string.

Help project message **P1053 Maximum number of font ranges exceeded**

The maximum number of font ranges that can be specified is five. The rest are ignored.

Help project message **P1055 Current font range overlaps previously defined range**

A font size range overlaps a previously defined mapping. Adjust either font range to remove any overlaps. The second mapping is ignored.

Help project message **P1056 Unrecognized font name in Forcefont option**

A font name not supported by the compiler has been encountered. The font name is ignored and the compiler uses the default Helvetica font.

Help project message **P1057 Font name too long**

Font names cannot exceed 20 characters. The font is ignored.

Help project message **P1059 Invalid multiple-key syntax**

The syntax used with a ***MULTIKEY*** option is unrecognized. See "Building the Help files" for the proper syntax.

Help project message **P1061 Character already used**

The specified keyword-table identifier is already in use. Choose another character.

Help project message **P1063 Characters 'K' and 'k' cannot be used**

These characters are reserved for Help's normal keyword table. Choose another character.

Help project message **P1065 Maximum number of keyword tables exceeded**

The limit of five keyword tables has been exceeded. Reduce the number. The excess tables are ignored.

Help project message **P1067 Equal sign missing**

An option is missing its required equal sign on the specified line. Check the syntax for the option.

Help project message **P1069 Context string missing**

The line specified is missing a context string before an equal sign.

Help project message **P1071 Incomplete line in *sectionname* section**

The entry on the specified line is not complete. The line is skipped.

Help project message **P1073 Unrecognized option in [OPTIONS] section**

An option has been used that is not supported by the compiler. The line is skipped.

Help project message

P1075 Invalid build expression

The syntax used in the build expression on the specified line contains one or more logical or syntax errors.

Help project message

P1077 Warning level must be 1, 2, or 3

The ***WARNING*** reporting level can only be set to 1, 2, or 3. The compiler will default to full reporting (level 3).

Help project message

P1079 Invalid compression option

The ***COMPRESS*** option can only be set to TRUE or FALSE. The compilation continues without compression.

Help project message

P1081 Invalid title string

The ***TITLE*** option defines a string that is null or contains more than 32 characters. The title is truncated.

Help project message

P1083 Invalid context identification number

The context number on the specified line is null or contains invalid characters.

Help project message

P1085 Unrecognized text

The unrecognizable text that follows valid text in the specified line is ignored.

Help project message

P1086 Invalid font-range syntax

The font-range definition on the specified line contains invalid syntax. The compiler ignores the line. Check the syntax for the ***MAPFONTSIZE*** option.

Help project message

P1089 Unrecognized sort ordering

You have specified an ordering that is not supported by the compiler. Contact Borland Technical Support for clarification of the error.

Compile-time error

Parameter names are used only with a function body

When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype. A list of parameter names only is not allowed.

Example declarations include:

```
int func();              // declaration without prototype--OK
int func(int, int);      // declaration with prototype--OK
int func(int i, int j); // parameter names in prototype--OK
int func(i, j);          // parameter names only--illegal
```

Compile-time error

Parameter *number* missing name

In a function definition header, this parameter consisted only of a type specifier *number* with no parameter name. This is not

legal in C. (It is allowed in C++, but there's no way to refer to the parameter in the function.)

Compile-time warning

Parameter *parameter* is never used

The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

Librarian error

***path* – path is too long**

This error occurs when the length of any of the library file or module file's *path* is greater than 64.

Compile-time error

Pointer to structure required on left side of –> or –>*

Nothing but a pointer is allowed on the left side of the arrow (**->**) in C or C++. In C++ a **–>*** operator is allowed.

Compile-time warning

Possible use of *identifier* before definition

Your source file used the named variable in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the program uses it.

Compile-time warning

Possibly incorrect assignment

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (that is, part of an **if**, **while** or **do-while** statement). More often than not, this is a typographical error for the equality operator. If you wish to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. Thus,

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```

Librarian error

public *symbol* in module *filename* clashes with prior module

A public symbol may only appear once in a library file. A module which is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added.

Help RTF message

R2001 Unable to open bitmap file *filename*

The specified bitmap file is unreadable. This is a DOS file error.

Help RTF message

R2003 Unable to include bitmap file *filename*

The specified bitmap file could not be found or is unreadable. This is a DOS file error or an out-of-memory condition.

Help RTF message

R2005 Disk full

The Help resource file could not be written to disk. Create more space on the destination drive.

Help RTF message

R2009 Cannot use reserved DOS device name for file *filename*

A file has been referred to as COM1, LPT2, PRN, etc. Rename the file.

Help RTF message

R2013 Output file *filename* already exists as a directory

There is a subdirectory in the Help project root with the same name as the desired Help resource file. Move or rename the subdirectory.

Help RTF message

R2015 Output file *filename* already exists as read-only

The specified filename cannot be overwritten by the Help resource file because the file has a read-only attribute. Rename the project file or change the file's attribute.

Help RTF message

R2017 Path for file *filename* exceeds limit of 79 characters

The absolute pathname, or the combined root and relative pathname, to the specified file exceed the DOS limit of 79 characters. The file is ignored.

Help RTF message

R2019 Cannot open file *filename*

The specified file is unreadable. This is a DOS file error.

Help RTF message

R2021 Cannot find file *filename*

The specified file could not be found or is unreadable. This is a DOS file error or an out-of-memory condition.

Help RTF message

R2023 Not enough memory to build Help file

To free up memory, unload any unneeded applications, device drivers, and memory-resident programs.

Help RTF message

R2025 File environment error

The compiler has insufficient available file handles to continue. Increase the values for FILES= and BUFFERS= in your CONFIG.SYS file and reboot.

Help RTF message

R2027 Build tag *tagname* not defined in [BUILDTAGS] section of project file

The specified build tag has been assigned to a topic, but not declared in the project file. The tag is ignored for the topic.

Help RTF message

R2033 Context string in Map section not defined in any topic

There are one or more context strings defined in the project file that the compiler could not find topics for.

Help RTF message

R2035 Build expression missing from project file

The topics have build tags, but there is no Build= expression in the project file. The compiler includes all topics in the build.

Help RTF message

R2037 File *filename* cannot be created, due to previous error(s)

The Help resource file could not be created because the compiler has no topics remaining to be processed. Correct the errors that preceded this error and recompile.

Help RTF message

R2039 Unrecognized table formatting in topic *topicnumber* of file *filename*

The compiler ignores table formatting that is unsupported in Help. Reformat the entries as linear text if possible.

Help RTF message

R2041 Jump *context_string* unresolved in topic *topicnumber* of file *filename*

The specified topic contains a context string that identifies a nonexistent topic. Check spelling, and that the desired topic is included in the build.

Help RTF message

R2043 Hotspot text cannot spread over paragraphs

A jump term spans two paragraphs. Remove the formatting from the paragraph mark.

Help RTF message

R2045 Maximum number of tab stops reached in topic *topicnumber* of file *filename*

The limit of 32 tab stops has been exceeded in the specified topic. The default stops are used after the 32nd tab.

Help RTF message

R2047 File *filename* not created

There are no topics to compile, or the build expression is false for all topics. There is no Help resource file created.

Help RTF message

R2049 Context string text too long in topic *topicnumber* of file *filename*

Context string hidden text cannot exceed 64 characters. The string is ignored.

Help RTF message

R2051 File *filename* is not a valid RTF topic file

The specified file is not an RTF file. Check that you have saved the topic as RTF from your word processor.

Help RTF message

R2053 Font *fontname* in file *filename* not in RTF font table

A font not defined in the RTF header has been entered into the topic. The compiler uses the default system font.

Help RTF message

R2055 File *filename* is not a usable RTF topic file

The specified file contains a valid RTF header, but the content is not RTF or is corrupted.

Help RTF message

R2057 Unrecognized graphic format in topic *topicnumber* of file *filename*

The compiler supports only Windows bitmaps. Check that metafiles or Macintosh formats have not been used. The graphic is ignored.

Help RTF message

R2059 Context string identifier already defined in topic *topicnumber* of file *filename*

There is more than one context-string identifier footnote for the specified topic. The compiler uses the identifier defined in the first # footnote.

Help RTF message

R2061 Context string *contextname* already used in file *filename*

The specified context string was previously assigned to another topic. The compiler ignores the latter string and the topic has no identifier.

Help RTF message

R2063 Invalid context-string identifier for topic *topicnumber* of file *filename*

The context-string footnote contains nonalphanumeric characters or is null. The topic is not assigned an identifier.

Help RTF message

R2065 Context string defined for index topic is unresolved

The index topic defined in the project file could not be found. The compiler uses the first topic in the build as the index.

Help RTF message

R2067 Footnote text too long in topic *topicnumber* of file *filename*

Footnote text cannot exceed the limit of 1000 characters. The footnote is ignored.

Help RTF message

R2069 Build tag footnote not at beginning of topic *topicnumber* of file *filename*

The specified topic contains a build tag footnote that is not the first character in the topic. The topic is not assigned a build tag.

Help RTF message

R2071 Footnote text missing in topic *topicnumber* of file *filename*

The specified topic contains a footnote that has no characters.

Help RTF message

R2073 Keyword string is null in topic *topicnumber* of file *filename*

A keyword footnote exists for the specified topic, but contains no characters.

Help RTF message

R2075 Keyword string too long in topic *topicnumber* of file *filename*

The text in the keyword footnote in the specified topic exceeds the limit of 255 characters. The excess characters are ignored.

Help RTF message

R2077 Keyword(s) defined without title in topic *topicnumber* of file *filename*

Keyword(s) have been defined for the specified topic, but the topic has no title assigned. Search Topics Found displays Untitled Topic<< for the topic.

Help RTF message

R2079 Browse sequence string is null in topic *topicnumber* of file *filename*

The browse-sequence footnote for the specified topic contains no sequence characters.

Help RTF message

R2081 Browse sequence string too long in topic *topicnumber* of file *filename*

The browse-sequence footnote for the specified topic exceeds the limit of 128 characters. The sequence is ignored.

Help RTF message

R2083 Missing sequence number in topic *topicnumber* of file *filename*

A browse-sequence number ends in a colon (:) for the specified topic. Remove the colon, or enter a "minor" sequence number.

Help RTF message

R2085 Sequence number already defined in topic *topicnumber* of file *filename*

There is already a browse-sequence footnote for the specified topic. The latter sequence is ignored.

Help RTF message

R2087 Build tag too long

A build tag for the specified topic exceeds the maximum of 32 characters. The tag is ignored for the topic.

Help RTF message

R2089 Title string null in topic *topicnumber* of file *filename*

The title footnote for the specified topic contains no characters. The topic is not assigned a title.

Help RTF message

R2091 Title too long in topic *topicnumber* of file *filename*

The title for the specified topic exceeds the limit of 128 characters. The excess characters are ignored.

Help RTF message

R2093 Title titlename in topic *topicnumber* of file *filename* used previously

The specified title has previously been assigned to another topic.

Help RTF message

R2095 Title defined more than once in topic *topicnumber* of file *filename*

There is more than one title footnote in the specified topic. The compiler uses the first title string.

Help RTF message

R2501 Using old key-phrase table

Maximum compression can only result by deleting the .PH file before each recompilation of the Help topics.

Help RTF message

R2503 Out of memory during text compression

The compiler encountered a memory limitation during compression. Compilation continues with the Help resource file not compressed. Unload any unneeded applications, device drivers, and memory-resident programs.

Help RTF message

R2505 File environment error during text compression

The compiler has insufficient available file handles for compression. Compilation continues with the Help resource file not compressed. Increase the values for FILES= and BUFFERS= in your CONFIG.SYS file and reboot.

Help RTF message

R2507 DOS file error during text compression

The compiler encountered a problem accessing a disk file during compression. Compilation continues with the Help resource file not compressed.

Help RTF message

R2509 Error during text compression

One of the three compression errors—R2503, R2505, or R2507—has occurred. Compilation continues with the Help resource file not compressed.

Help RTF message

R2701 Internal error
R2703 Internal error
R2705 Internal error
R2707 Internal error
R2709 Internal error

Contact Borland Technical Support for clarification of the error.

Librarian error

record kind *num* found, expected theadr or lheadr in module *filename*

The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.

Librarian error

record length *len* exceeds available buffer in module *module*

This error occurs when the record length *len* exceeds the available buffer to load the buffer in module *module*. This occurs when the librarian runs out of dynamic memory.

Librarian error

record type *type* found, expected theadr or lheadr in *module*

The librarian encountered an unexpected type *type* instead of the expected THEADR or LHEADER record in module *module*.

Compile-time warning

Redefinition of *macro* is not identical

Your source file redefined the named macro using text that was not exactly the same as the first definition of the macro. The new text replaces the old.

Compile-time error

Reference initialized with *type1*, needs lvalue of type *type2*

A reference variable or parameter that is not declared constant must be initialized with an lvalue of the appropriate type. In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.

Compile-time error

Reference member *member* in class without constructors

A class that contains reference members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.

Compile-time error

Reference member *member* is not initialized

References must always be initialized. A class member of reference type must have an initializer provided in all constructors for that class. This means that you cannot depend on the compiler to generate constructors for such a class, since it has no way of knowing how to initialize the references.

Compile-time error

Reference member *member* needs a temporary for initialization

You provided an initial value for a reference type which was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Since there is no obvious place to store this temporary, the initialization is illegal.

Compile-time error

Reference variable *variable* must be initialized

This C++ object is declared as a reference but is not initialized. All references must be initialized at the point of their declaration.

Compile-time fatal error

Register allocation failure

This is a sign of some form of compiler error. Some expression in the indicated function was so complicated that the code generator could not generate code for it. Try to simplify the offending function. Notify Borland Technical Support if the compiler encounters this error.

Librarian warning

results are safe in file *filename*

The librarian has successfully built the library into a temporary file, but cannot rename the file to the desired library name. The temporary file will not be removed (so that the library can be preserved).

Compile-time error

Size of *identifier* is unknown or zero

This identifier was used in a context where its size was needed. A **struct** tag may only be declared (the **struct** not defined yet), or an **extern** array may be declared without a size. It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declaration so that the size of *identifier* is available.

Compile-time error

sizeof may not be applied to a bit field

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

Compile-time error

sizeof may not be applied to a function

sizeof may be applied only to data objects, not functions. You may request the size of a pointer to a function.

Compile-time error

Size of the type is unknown or zero

This type was used in a context where its size was needed. For example, a **struct** tag may only be declared (the **struct** not defined yet). It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declarations so that the size of this type is available.

Compile-time error

***identifier* specifies multiple or duplicate access**

A base class may be declared **public** or **private**, but not both. This access specifier may appear no more than once for a base class.

Run-time error

Stack overflow

The default stack size for Turbo C++ programs is 5120 bytes. This should be enough for most programs, but those which execute recursive functions or store a great deal of local data can overflow the stack. You will only get this message if you have stack checking enabled. If you do get this message, you can try increasing the stack size or decreasing your program's dependence on the stack. For Windows, to increase your stack size, use STACKSIZE in your module definition .DEF file.

To decrease the amount of local data used by a function, look at the example below. The variable *buffer* has been declared static and does not consume stack space like *list* does.

```
void anyfunction(void)
{
   static int buffer[2000];   /* resides in the data segment */
   int list[2000];            /* resides on the stack */
}
```

There are two disadvantages to declaring local variables as static.

1. It now takes permanent space away from global variables and the heap. (You have to rob Peter to pay Paul.) This is usually only a minor disadvantage.
2. The function may no longer be reentrant. What this means is that if the function is called recursively or asynchronously and it is important that each call to the function have its own unique copy of the variable, you cannot make it static. This is because every time the function is called, it will use the same exact memory space for the variable, rather than allocating new space for it on each call. You could have a sharing problem if the function is trying to execute from within itself (recursively) or at the same time as itself (asynchronously). For most DOS programs this is not a problem.

Linker warning

Stack size is less than 1400h. It has been reset to 1400h.

Windows 3.0 requires the stack size of an application to be at least 1400h. If the automatic data segment (ADS) is near 64K, but your stack is less than 1400h, this can cause the ADS to overflow at load time, but not at link time. To protect against this, the linker forces the stack size to be at least 1400h for a Windows application.

Compile-time error

Statement missing ;

The compiler encountered an expression statement without a semicolon following it.

Compile-time error

Storage class *storage class* is not allowed here

The given storage class is not allowed here. Probably two storage classes were specified, and only one may be given.

Compile-time warning

Structure passed by value

A structure was passed by value as an argument to a function without a prototype. It is a frequent programming mistake to leave an address-of operator (**&**) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.

Compile-time error

Structure required on left side of . or .*

The left side of a dot (.) operator (or C++ dot-star operator) must evaluate to a structure type. In this case it did not.

Compile-time error

Structure size too large

Your source file declared a structure larger than 64K.

Linker fatal error

Stub program exceeds 64K

This error occurs if a DOS stub program written for a Windows application exceeds 64K. Stub programs are specified via the STUB module definition file statement; the linker only supports stub programs up to 64K.

Compile-time warning

Style of function definition is now obsolete

In C++, this old C style of function definition is illegal:

```
int func(p1, p2)
int p1, p2;
{
:
}
```

This practice may not be allowed by other C++ compilers.

Compile-time error

Subscripting missing]

The compiler encountered a subscripting expression which was missing its closing bracket. This could be caused by a missing or extra operator, or mismatched parentheses.

Compile-time warning

Superfluous & with function

An address-of operator (**&**) is not needed with function name; any such operators are discarded.

Compile-time warning

Suspicious pointer conversion

The compiler encountered some conversion of a pointer which caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.

Compile-time error

Switch selection expression must be of integral type

The selection expression in parentheses in a **switch** statement must evaluate to an integral type (**char**, **short**, **int**, **long**, **enum**). You may be able to use an explicit cast to satisfy this requirement.

Compile-time error

Switch statement missing (

In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.

Compile-time error

Switch statement missing)

In a **switch** statement, the compiler found no right parenthesis after the test expression.

Linker fatal error

***filename* (*linenum*): Syntax error**

The linker found a syntax error in the module definition file. The filename and line number tell you where the syntax error occurred.

Linker fatal error

Table limit exceeded

One of linker's internal tables overflowed. This usually means that the programs being linked have exceeded the linker's capacity for public symbols, external symbols, or for logical segment definitions. Each instance of a distinct segment name in an object file counts as a logical segment; if two object files define this segment, then this results in two logical segments.

Compile-time error

Template argument must be a constant expression

A non-type actual template class argument must be a constant expression (of the appropriate type); this includes constant integral expressions, and addresses of objects or functions with external linkage or members.

Compile-time error

Template class nesting too deep: 'class'

The compiler imposes a certain limit on the level of template class nesting; this limit is usually only exceeded through a recursive template class dependency. When this nesting limit is exceeded, the compiler will issue this error message for all of the nested template classes, which usually makes it easy to spot the recursion. This is always followed by the fatal error **Out of memory**.

For example, consider the following set of template classes:

```
template<class T> class A
{
   friend  class   B<T*>;
};

template<class T> class B
{
   friend  class   A<T>;
};

A<int>  x;
```

This snippet will be flagged with the following errors:

```
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * *>'
Error: Template class nesting too deep: 'B<int * * * *>'
Error: Template class nesting too deep: 'A<int * * *>'
Error: Template class nesting too deep: 'B<int * * *>'
Error: Template class nesting too deep: 'A<int * *>'
Error: Template class nesting too deep: 'B<int * *>'
Error: Template class nesting too deep: 'A<int *>'
Error: Template class nesting too deep: 'B<int *>'
Error: Template class nesting too deep: 'A<int>'
Fatal: Out of memory
```

Compile-time error

Template function argument *argument* not used in argument types

The given argument was not used in the argument list of the function. The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.

Compile-time error

Template functions may only have type-arguments

A function template was declared with a non-type argument. This is not allowed with a template function, as there is no way to specify the value when calling it.

Compile-time error

Templates can only be declared at file level

Templates cannot be declared inside classes or functions, they are only allowed in the global scope (file level).

Compile-time error

Templates must be classes or functions

The declaration in a template declaration must specify either a class type or a function.

Compile-time warnings

Temporary used to initialize *identifier*
Temporary used for parameter *number* in call to *function*
Temporary used for parameter *parameter* in call to *function*
Temporary used for parameter *number*
Temporary used for parameter *parameter*

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

For example, here function **f** requires a reference to an **int**, and *c* is a **char**:

```
f(int&);
char c;
f(c);
```

Instead of calling **f** with the address of *c*, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

Linker fatal error

Terminated by user

You canceled the link.

Compile-time error

The constructor *constructor* is not allowed

Constructors of the form **X::(X)** are not allowed. The correct way to write a copy constructor is **X::(const X&)**.

Compile-time error

The value for *identifier* is not within the range of an int

All enumerators must have values which can be represented as an integer. You attempted to assign a value which is out of the range of an integer. In C++ if you need a constant of this value, use a **const** integer.

Compile-time error

'this' can only be used within a member function

In C++, **this** is a reserved word that can be used only within class member functions.

Compile-time warning

This initialization is only partly bracketed

Result of IDE Options | Compiler | Messages | ANSI violations selection. Initialization is only partially bracketed. When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces may be

used. This ensures that your idea and the compiler's idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

Compile-time error

Too few arguments in template class name *template*

A template class name was missing actual values for some of its formal parameters.

Compile-time error

Too few parameters in call

A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.

Compile-time error

Too few parameters in call to *function*

A call to the named function (declared using a prototype) had too few arguments.

Compile-time error

Too many decimal points

The compiler encountered a floating-point constant with more than one decimal point.

Compile-time error

Too many default cases

The compiler encountered more than one **default** statement in a single **switch**.

Compile-time error

Too many error or warning messages

There were more errors or warnings than the setting on Options | Compiler | Messages | Display.

Linker error

Too many error or warning messages

The number of messages reported by the compiler has exceeded its limit. This error indicates that the linker reached its limit.

Compile-time error

Too many exponents

The compiler encountered more than one exponent in a floating-point constant.

Compile-time error

Too many initializers

The compiler encountered more initializers than were allowed by the declaration being initialized.

Compile-time error

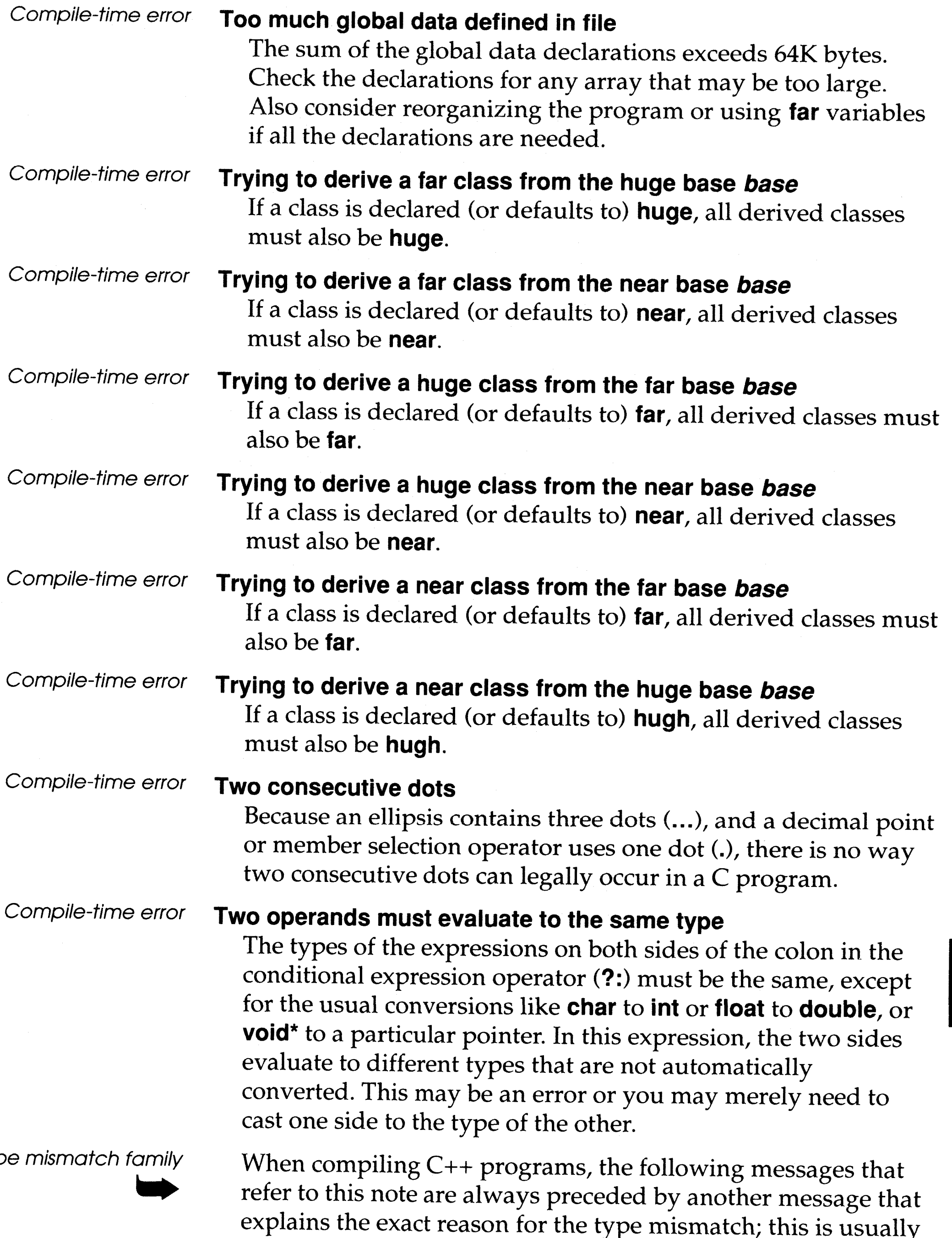
Too many storage classes in declaration

A declaration may never have more than one storage class.

Compile-time error

Too many types in declaration

A declaration may never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum**, or **typedef-***name*.



Compile-time error

Too much global data defined in file

The sum of the global data declarations exceeds 64K bytes. Check the declarations for any array that may be too large. Also consider reorganizing the program or using **far** variables if all the declarations are needed.

Compile-time error

Trying to derive a far class from the huge base *base*

If a class is declared (or defaults to) **huge**, all derived classes must also be **huge**.

Compile-time error

Trying to derive a far class from the near base *base*

If a class is declared (or defaults to) **near**, all derived classes must also be **near**.

Compile-time error

Trying to derive a huge class from the far base *base*

If a class is declared (or defaults to) **far**, all derived classes must also be **far**.

Compile-time error

Trying to derive a huge class from the near base *base*

If a class is declared (or defaults to) **near**, all derived classes must also be **near**.

Compile-time error

Trying to derive a near class from the far base *base*

If a class is declared (or defaults to) **far**, all derived classes must also be **far**.

Compile-time error

Trying to derive a near class from the huge base *base*

If a class is declared (or defaults to) **hugh**, all derived classes must also be **hugh**.

Compile-time error

Two consecutive dots

Because an ellipsis contains three dots (**...**), and a decimal point or member selection operator uses one dot (**.**), there is no way two consecutive dots can legally occur in a C program.

Compile-time error

Two operands must evaluate to the same type

The types of the expressions on both sides of the colon in the conditional expression operator (**?:**) must be the same, except for the usual conversions like **char** to **int** or **float** to **double**, or **void*** to a particular pointer. In this expression, the two sides evaluate to different types that are not automatically converted. This may be an error or you may merely need to cast one side to the type of the other.

Type mismatch family

When compiling C++ programs, the following messages that refer to this note are always preceded by another message that explains the exact reason for the type mismatch; this is usually

"Cannot convert 'type1' to 'type2'", but the mismatch may be due to many other reasons.

Compile-time error

Type mismatch in default argument value
Type mismatch in default value for parameter *parameter*

The default parameter value given could not be converted to the type of the parameter. The first message is used when the parameter was not given a name.

Compile-time error

Type mismatch in parameter *number*

The function called, via a function pointer, was declared with a prototype; the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *number* in call to *function*

Your source file declared the named function with a prototype, and the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *parameter*

Your source file declared the function called via a function pointer with a prototype, and the named parameter could not be converted to the declared parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in parameter *parameter* in call to *function*

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type. See entry for **Type mismatch in parameter *parameter***.

Compile-time error

Type mismatch in parameter *parameter* in template class name *template*
Type mismatch in parameter *number* in template class name *template*

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type. See the previous note on Type mismatch family.

Compile-time error

Type mismatch in redeclaration of *identifier*

Your source file redeclared with a different type than was originally declared. This can occur if a function is called and subsequently declared to return something other than an

integer. If this has happened, you must declare the function before the first call to it.

Compile-time error

Type name expected

One of these errors has occurred:

- In declaring a file-level variable or a **struct** field, neither a type name nor a storage class was given.
- In declaring a **typedef**, no type for the name was supplied.
- In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
- In supplying a C++ base class name, the name was not the name of a class.

Compile-time error

Type qualifier *identifier* must be a struct or class name

The C++ qualifier in the construction *qual::identifier* is not the name of a **struct** or **class**.

Compile-time fatal error

Unable to create output file *filename*

The work disk is full or write-protected or the output directory does not exist. If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write-protected, move the source files to a writable disk and restart the compilation.

Linker fatal error and Librarian error

Unable to open file *filename*
unable to open *filename*

This occurs if the named file does not exist or is misspelled.

Librarian error

unable to open *filename* for output

The librarian cannot open the specified file for output. This is usually due to lack of disk space for the target library, or a listing file. Additionally this error will occur if the target file exists but is marked as a read only file.

Compile-time error

Unable to open include file *filename*

The compiler could not find the named file. This could also be caused if an **#include** file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try `FILES=20`). Check whether the named file exists.

Compile-time error

Unable to open input file *filename*

This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is on the proper disk or directory.

Librarian error

unable to rename *filename* to *filename*

The librarian builds a library into a temporary file and then renames the temporary file to the target library file name. If there is an error, usually due to lack of disk space, this message will be posted.

Compile-time error

Undefined label *identifier*

The named label has a **goto** in the function, but no label definition.

Compile-time warning

Undefined structure *identifier*

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

Compile-time error

Undefined structure *structure*

Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.

Compile-time error

Undefined symbol *identifier*

The named identifier has no declaration. This could be caused by a misspelling either at this point or at the declaration. This could also be caused if there was an error in the declaration of the identifier.

Linker error

Undefined symbol *symbol* in module *module*

The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check to make sure the symbol is spelled correctly.

You will usually see this error from the linker for Turbo C++ symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs. If you are linking C++ code with C modules, you might have forgotten to wrap C external declarations in `extern "C" {...}`. You might have a case mismatch between two symbols.

Compile-time error

Unexpected }

An extra right brace was encountered where none was expected. Check for a missing **{**.

Compile-time error

Unexpected end of file in comment started on *line number*

The source file ended in the middle of a comment. This is normally caused by a missing close of comment (*/).

Compile-time error

Unexpected end of file in conditional started on line *line number*

The source file ended before the compiler encountered an **!endif**. The **!endif** was either missing or misspelled.

Compile-time error

union cannot be a base type

A union cannot be used as a base type for another class type.

Compile-time error

union cannot have a base type

A union cannot be derived from any other class.

Compile-time error

Union member *member* is of type class with constructor
Union member *member* is of type class with destructor
Union member *member* is of type class with operator=

A union may not contain members that are of type **class** with user-defined constructors, destructors, or operator=.

Compile-time error

unions cannot have virtual member functions

A union may not have virtual functions as its members.

Compile-time warning

Unknown assembler instruction

See Chapter 7 for more on opcode spelling.

The compiler encountered an inline assembly statement with a disallowed opcode. Check the spelling of the opcode. This warning is off by default.

Librarian warning

unknown command line switch *X* ignored

A forward slash character (/) was encountered on the command line or in a response file without being followed by one of the allowed options.

Compile-time error

Unknown language, must be C or C++

In the C++ construction

```
extern "name" type func( /*...*/ );
```

The name given in quotes must be "C" or "C++"; other language names are not recognized. For example, you can declare an external Pascal function without the compiler's renaming like this:

```
extern "C" int pascal func( /*...*/ );
```

A C++ (possibly overloaded) function may be declared Pascal and allow the usual compiler renaming (to allow overloading) like this:

```
extern int pascal func( /*...*/ );
```

Compile-time warning

Unreachable code

A **break, continue, goto** or **return** statement was not followed by a label or the end of a loop or function. The compiler checks **while, do** and **for** loops with a constant test condition, and attempts to recognize loops which cannot fall through.

Compile-time error

Unterminated string or character constant

The compiler found no terminating quote after the beginning of a string or character constant.

Compile-time error

Use . or –> to call *function*

You tried to call a member function without giving an object.

Compile-time error

Use . or –> to call *member*, or & to take its address

A reference to a non-static class member without an object was encountered. Such a member may not be used without an object, or its address must be taken using the **&** operator.

Compile-time error

Use :: to take the address of a member function

If **f** is a member function of class *c*, you take its address with the syntax **&c::f**. Note the use of the class type name, not the name of an object, and the **::** separating the class name from the function name. (Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)

Compile-time warning

Use qualified name to access nested type *type*

In older versions of the C++ specification, typedef and tag names declared inside classes were directly visible in the global scope. With the latest specification of C++, these names must be prefixed with a **class::** qualifier if they are to be used outside of their class' scope. To allow older code to compile, whenever such a name is uniquely defined in one single class, Turbo C++ will allow its usage without **class::** and issues this warning.

Linker or compile-time error

User break

You pressed *Cancel* while compiling in the IDE, aborting the process. (This is not an error, just a confirmation.)

Compile-time error

Value of type void is not allowed

A value of type **void** is really not a value at all, and thus may not appear in any context where an actual value is required. Such contexts include the right side of an assignment, an argument of a function, and the controlling expression of an **if, for,** or **while** statement.

Compile-time error

Variable *identifier* is initialized more than once

This variable has more than one initialization. It is legal to declare a file level variable more than once, but it may have only one initialization (even if two are the same).

Compile-time error

'virtual' can only be used with member functions

A data member has been declared with the **virtual** specifier; only member functions may be declared **virtual**.

Compile-time error

Virtual function *function1* conflicts with base class *base*

A virtual function has the same argument types as one in a base class, but a different return type. This is illegal.

Compile-time error

virtual specified more than once

The C++ reserved word **virtual** may appear only once in a member function declaration.

Compile-time error

void & is not a valid type

A reference always refers to an object, but an object cannot have the type void. Thus the type void is not allowed.

Compile-time warning

Void functions may not return a value

Your source file declared the current function as returning **void**, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.

Compile-time error

function* was previously declared with the language *language

Only one language can be used with **extern** for a given function. This function has been declared with different languages in different locations in the same module.

Compile-time error

While statement missing (

In a **while** statement, the compiler found no left parenthesis after the **while** keyword.

Compile-time error

While statement missing)

In a **while** statement, the compiler found no right parenthesis after the test expression.

Compile-time error

Wrong number of arguments in call of macro *macro*

Your source file called the named macro with an incorrect number of arguments.

A P P E N D I X

A

HC: The Windows Help compiler

A Help system provides users with online information about an application. Creating the system requires the efforts of both a Help writer and a Help programmer. The Help writer plans, writes, codes, builds, and keeps track of Help topic files, which are text files that describe various aspects of the application. The Help programmer ensures that the Help system works properly with the application.

This chapter describes the following topics:

- Providing and creating the Help system
- Planning the Help system
- Creating Help topic files
- Building the Help file
- Help examples and compiler error messages

This section and those that follow assume you are familiar with Windows Help. The sections use examples from sample applications provided on your disks. If you are unfamiliar with Windows Help, take a moment to run the sample application.

Creating a Help system: The development cycle

The creation of a Help system for a Windows application comprises the following major tasks:

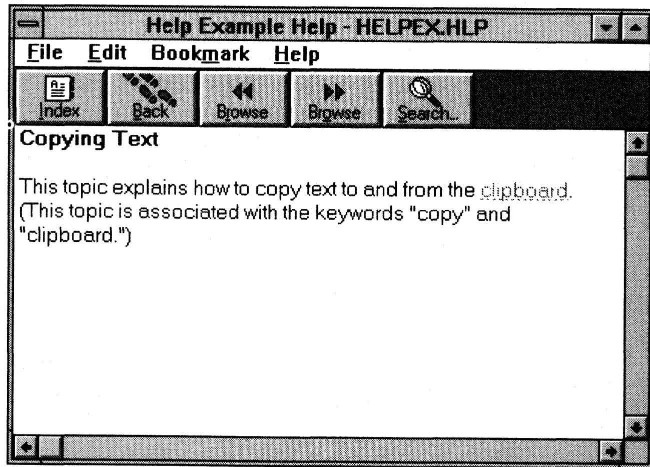
1. Gathering information for the Help topics.
2. Planning the Help system. The section, "Planning the Help system," describes considerations you should keep in mind when planning your Help system.
3. Writing the text for the Help topics.
4. Entering all required control codes into the text files. Control codes determine how the user can move around the Help system. The section titled, "How Help Appears to the Writer," includes an example of several control codes. A later section, "Creating the Help topic files," describes the codes in detail.
5. Creating a project file for the build. The Help project file provides information that the Help Compiler needs to build a Help resource file. A later section, "Building the Help files," describes the Help project file.
6. Building the Help resource file. The Help resource file is a compiled version of the topic files the writer creates. Later in this chapter, the section "Building the Help files" describes how to compile a Help resource file.
7. Testing and debugging the Help system.
8. Programming the application so that it can access Windows Help.

How Help appears to the user

To the user, the Help system appears to be part of the application, and is made up of text and graphics displayed in the Help window in front of the application. Figure A.1 illustrates the Help window that appears when the user asks for help with copying text in Helpex.

The Help window displays one sample Help topic, a partial description of how to perform one task. In Figure A.1, the first sentence includes a definition of the word "clipboard." By pressing the mouse button when the cursor is on the word (denoted by dotted underlined text), the user can read the definition, which appears in an overlapping box as long as the mouse button is held down.

Figure A.1
Helpex help window

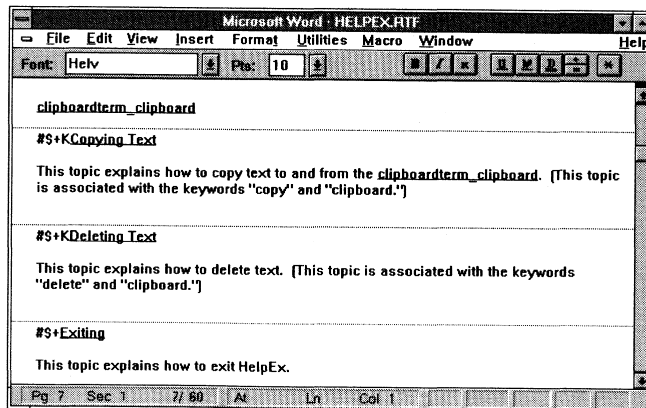


Cross-references to related topics are called jumps. By clicking on a jump term for a related topic (denoted by underlined text), the user changes the content of the Help window to a description of the new topic or command. Figure A.1 includes a look-up to the definition of "clipboard."

How Help appears to the help writer

To the writer, the Help system is a group of topic files, which are text files that include special codes. Figure A.2 illustrates the source text that corresponds to the topic shown in Figure A.1.

Figure A.2
Topic file



To create this topic, the Help writer describes the task, formats the text, and inserts codes using strikethrough text, underlined text, and footnotes. In place of strikethrough, the writer can use double

underlining if the word processor does not support strikethrough formatting. Footnotes in the text contain linking information required by the Help Compiler. The section "Planning the Help system" discusses formatting considerations. Another section, "Creating the Help topic files," describes how to create topics and enter the special codes that the Help system uses.

How Help appears to the help programmer

See "Building the Help files" for details about the Help application programming interface (API).

To the programmer, Windows Help is a stand alone Windows application which the user can run like any other application. Your application can call the **WinHelp** function to ask Windows to run the Help application and specify which topic to display in the Help window.

Planning the Help system

The initial task for the Help writer is to develop a plan for creating the system. This section discusses planning the Help system for a particular application; it covers these topics:

- Developing a plan
- Determining the topic file structure
- Designing the visual appearance of Help topics

Developing a plan

Before you begin writing Help topics using the information you have gathered, you and the other members of the Help team should develop a plan that addresses the following issues:

- The audience for your application
- The content of the Help topics
- The structure of topics
- The use of context-sensitive topics

You might want to present your plan in a design document that includes an outline of Help information, a diagram of the structure of topics, and samples of the various kinds of topics your system will include. Keep in mind that context-sensitive Help requires increased development time, especially for the application programmer.

Defining the audience

The audience you address determines what kind of information you make available in your Help system and how you present the information.

Users of Help systems might be classified as follows:

Table A.1
Your application audience

User	Background
Computer novice	Completely new to computing.
Application novice	Some knowledge of computing, but new to your kind of application. For example, if you are providing Help for a spreadsheet program, the application novice might be familiar only with word-processing packages.
Application intermediate	Knowledgeable about your kind of application.
Application expert	Experienced extensively with your type of application.

Keep in mind that one user may have various levels of knowledge. For example, the expert in word processors may have no experience using spreadsheets.

Planning the contents

You should create topics that are numerous enough and specific enough to provide your users with the help they need.

Novice users need help learning tasks and more definitions of terms. More sophisticated users occasionally seek help with a procedure or term, but most often refresh their memory of commands and functions. The expert user tends only to seek help with command or function syntax, keyboard equivalents, and shortcut keys.

There are no rules for determining the overall content of your Help system. If you are providing Help for all types of users, you will want to document commands, procedures, definitions, features, functions, and other relevant aspects of your application. If you are providing help for expert users only, you might want to omit topics that describe procedures. Let your audience definition guide you when deciding what topics to include.

Keep in mind that the decision to implement context-sensitive Help is an important one. Context-sensitive Help demands a close working relationship between the Help author and the

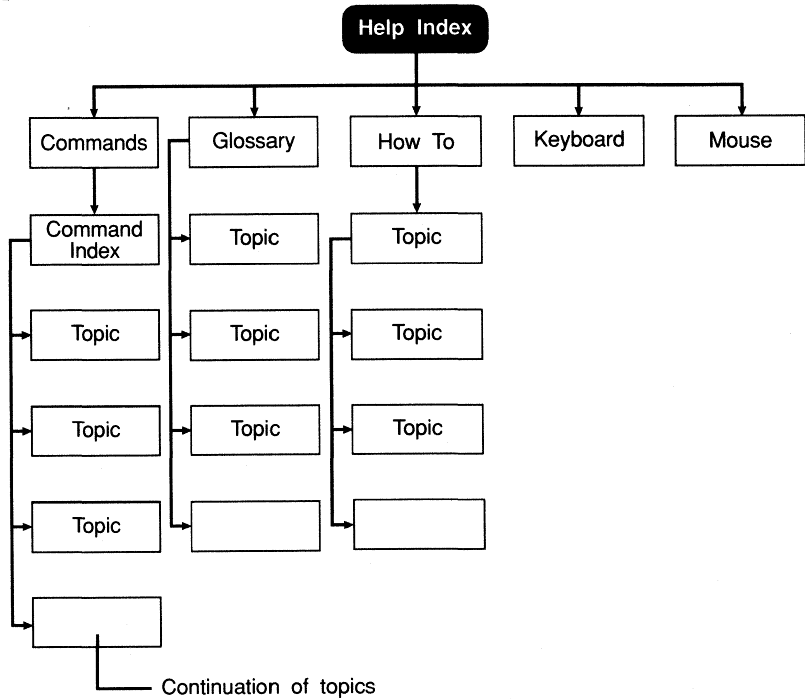
application programmer, and will therefore increase the development time necessary to create a successful Help system.

Planning the structure

Many Help systems structure topics hierarchically. At the top of the hierarchy is an index or a table of contents, or both. The index and table of contents list individual topics or categories of topics available to the user.

Topics themselves can be related hierarchically. Each successive step takes the user one level down in the hierarchy of the Help system until the user reaches topic information. The hierarchical relationship of Help topics determines in part how the user navigates through the Help system. Figure A.3 illustrates a possible hierarchy:

Figure A.3
Example of a help hierarchy



Helpex contains an index that lists several categories of topics. Each category includes a secondary index, which lists topics in the category, and the topics themselves.

Moving from the index to a topic, the user goes from the general to the specific.

The hierarchical structure provides the user a point of reference within Help. Users are not constrained to navigate up and down the hierarchy; they can jump from one topic to another, moving across categories of topics. The effect of jumps is to obscure hierarchical relationships. For example, the Windows Help application contains a search feature that lets the user enter a keyword into a dialog box and search for topics associated with that keyword. The Help application then displays a list of titles to choose from in order to access information that relates to the keyword.

For more about the search feature, see page 326.

Because users often know which feature they want help with, they can usually find what they want faster using the search feature than they can by moving through the hierarchical structure.

For more about browse sequences, see page 321.

In addition to ordering topics hierarchically, you can order them in a logical sequence that suits your audience. The logical sequence, or "browse sequence," lets the user choose the Browse button to move from topic to topic. Browse sequences are especially important for users who like to read several related topics at once, such as the topics covering the commands on the File menu.

Whichever structure you decide to use, try to minimize the number of lists that users must traverse in order to obtain information. Also, avoid making users move through many levels to reach a topic. Most Help systems function quite well with only two or three levels.

Displaying context-sensitive Help topics

Windows Help supports context-sensitive Help. When written in conjunction with programming of the application, context-sensitive Help lets the user press *F1* in an open menu to get help with the selected menu item. Alternatively, the user can press *Shift+F1* and then click on a screen region or command to get help on that item.

Developing context-sensitive Help requires coordination between the Help writer and the application programmer so that Help and the application pass the correct information back and forth.

For information on creating a Help project file, see page 337.

To plan for context-sensitive Help, the Help author and the application programmer should agree on a list of context numbers. Context numbers are arbitrary numbers that correspond to each menu command or screen region in the application, and are used to create the links to the corresponding Help topics. You

can then enter these numbers, along with their corresponding context-string identifiers, in the Help project file, which the Help Compiler uses to build a Help resource file.

For more on assigning context numbers, see page 349.

The context numbers assigned in the Help project file must correspond to the context numbers that the application sends at run time to invoke a particular topic.

See page 346 for more on context-sensitive Help.

If you do not explicitly assign context numbers to topics, the Help Compiler generates default values by converting topic context strings into context numbers.

Page 335 provides you with more information about using a tracker.

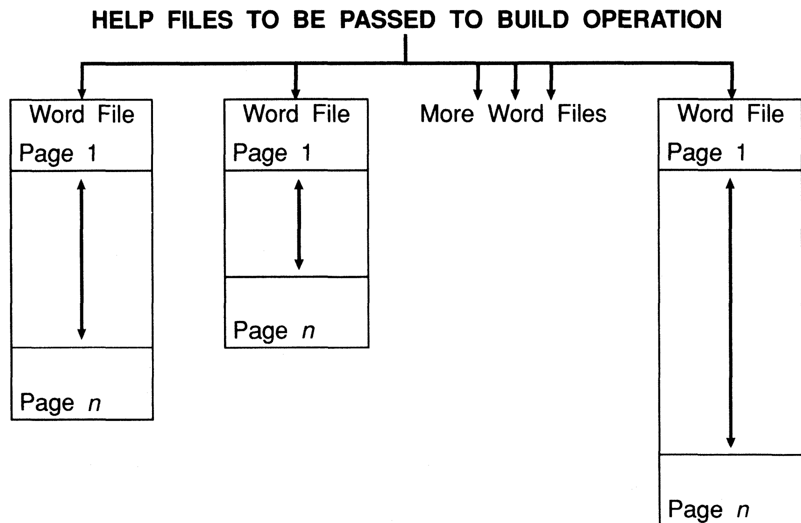
To manage context numbers and file information, you might want to create a Help tracker to list the context numbers for your context-sensitive topics.

Determining the topic file structure

The Help file structure remains essentially the same for all applications even though the context and number of topic files differ. Topic files are segmented into the different topics by means of page breaks. When you build the Help system, the compiler uses these topic files to create the information displayed for the user in the application's Help window.

Figure A.4 shows this basic file structure.

Figure A.4
Basic help file structure



Choosing a file structure for your application

When choosing a file structure for your Help system, consider the scope and content of the Help system you are planning. For example, you could place all Help topics in a single large topic file. Or, you could place each Help topic in a separate file. Neither of these file structures is generally acceptable. An enormous single file or too many individual files can present difficulties during the creation of the Help resource file.

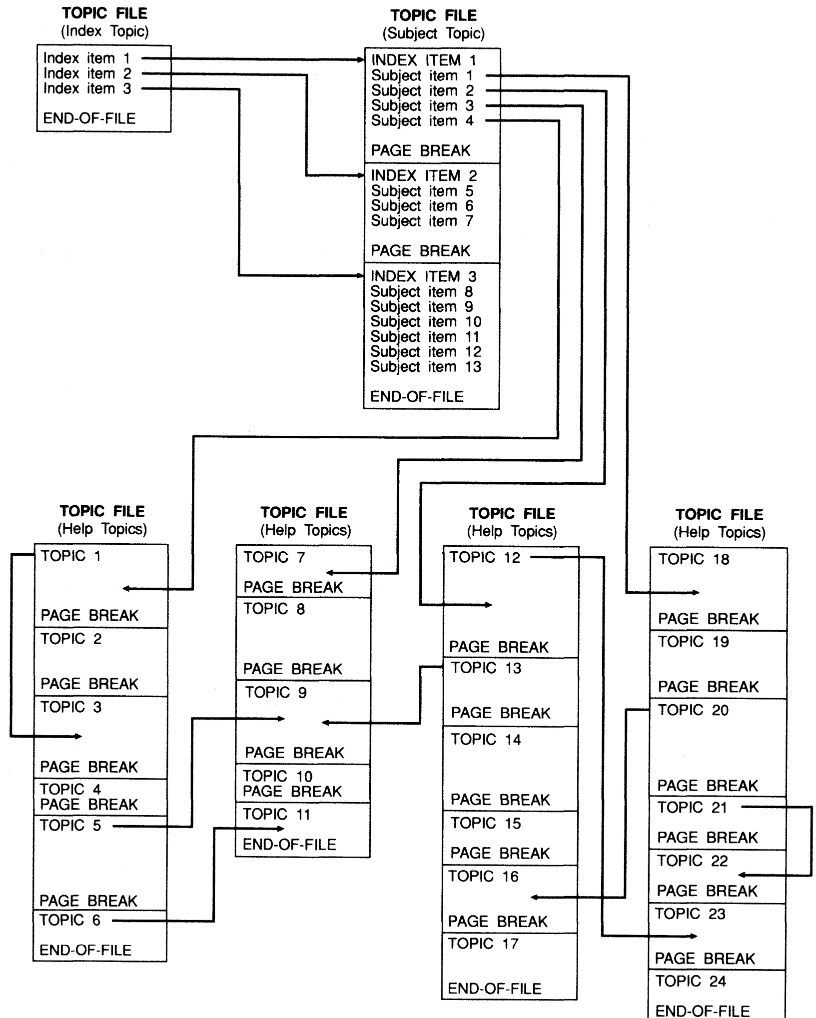


Figure A.5
Help file structure showing hypertext jumps

The number of topics relates to the number of features covered by the Help system. Consequently, you cannot make extensive changes to one without making changes to the other. For instance, if a number of additional product features are added to Help, then additional topics must be created to accommodate the new information.

Figure A.5 illustrates the file structure of a possible Help system. The number of topics and topic files is limited to simplify the diagram and more clearly show the concept of linking the topics together through jumps, shown in the figure as arrows. The figure is not intended to show the number of files that can be included in the Help file system. Moreover, the figure does not show how topic files are ordered using the browse feature.

Designing Help topics

How the information in the Help window appears to the user is primarily a function of the layout of the Help topic. The Windows Help application supports a number of text attributes and graphic images you can use to design your Help window.

This section provides general guidelines for designing a window and describes fonts and graphic images that Windows Help supports.

Layout of the Help text

Help text files are not limited to plain, unformatted text. You can use different fonts and point sizes, include color and graphics to emphasize points, indent paragraphs to present complex information, and use a variety of other visual devices to present your information.

Research on screen format and Help systems has produced general guidelines for presenting information to users. Table A.2 summarizes the findings of these studies.

Table A.2: Help design issues

Design Issue	Guideline
Language	*Use language appropriate for the audience you have defined.* Language that is too sophisticated for your audience can frustrate users by requiring them to learn the definition of unfamiliar terms and concepts.
Amount of text	*Use a minimum of text.* Studies indicate that reading speed decreases by 30 percent when users read online text rather than printed text. Minimal, concise text helps users compensate for the decreased reading speed.

Table A.2: Help design issues (continued)

Paragraph length	*Use short paragraphs.* Online users become overloaded with text more easily than do readers of printed material. Breaking your text into short paragraphs helps avoid this problem.
Whitespace	*Use it to help group information visually.* Whitespace is important to making online text more readable. Use it liberally, while also considering the overall size that a topic will occupy on the screen. Users tend to think there is more information on a screen than exists. For example, if the ratio of whitespace to text is 50:50, users perceive the ratio to be 40:60.
Highlighting	*Use highlighting techniques judiciously.* Windows Help provides a variety of highlighting devices, such as font sizes, font types, and color. Using a few devices can help users find information quickly. Using many devices will decrease the effectiveness of your visual presentation and frustrate users. As with print-based documentation, use only one or two fonts at a time.
Graphics and icons	*Use graphics to support the explanation of visual events.* Windows Help supports the use of bitmapped graphic images. Use appropriate images to help explain the function of icons and screen elements in your application. Remember that graphics will draw the user's eye before the accompanying text. Be sure to crop your images to remove distracting information. Use color images only if you are certain that all your users' systems have color capability. As with context-sensitive Help, consider the additional time necessary to create accurate and meaningful graphic images.
Design consistency	*Be rigorously consistent in your design.* Users expect the appearance of Help topics to be the same, regardless of the information presented. Consistent titling, highlighting, fonts, and positioning of text in the window is essential to an effective Help system.

Type fonts and sizes

The Windows Help application can display text in any font and size available to the system. When the topic files are passed to the build process, the Help Compiler attempts to use the fonts and sizes found in the topic files. If a font or point size cannot be matched exactly when the Help file is displayed by Windows Help, the closest available font and size on the user's system will be used.

Windows ships with only certain fonts in specific font sizes. If you write Help files using these fonts and sizes, the displayed Help file will closely match the printed word-processed file. Because fonts other than those shipped with Windows may not be available on users' machines, you might want to limit your font selection to the shipped Windows fonts.

The fonts included with Windows are shown in Table A.3:

Table A.3
Windows fonts

Font	Sizes
Courier	10,12,15
Helv	8,10,12,14,18,24
Modern	
Roman	
Script	
Symbol	8,10,12,14,18,24
Tms Rmn	8,10,12,14,18,24

Since Windows Help supports any Windows font, special symbols such as arrows can be included in your topics by using the Symbol font.

Graphic images

The Windows Help application allows you to embed graphics in the Help file. Graphics can be placed and displayed anywhere on the page. Text can appear next to the graphic.

Color graphic images can be included, provided you use only the available Windows system colors. If you use graphics tools that support an enhanced color palette to create or capture images, these images may not always display with the intended colors. And since you cannot control the color capabilities on a user's machine, you might want to limit your graphic images to black and white bitmaps.

For more information on placing graphics into your Help files, see page 332.

Keep in mind that graphics are most effective when they contribute to the learning process. Graphics not tied to the information are usually distracting rather than helpful and should be avoided.

For additional information about screen design, refer to the following books and journals:

- Bradford, Annette Norris. "Conceptual Differences Between the Display Screen and the Printed Page." *Technical Communication* (Third Quarter 1984): 13-16.
- Galitz, Wilbert O. *Handbook of Screen Format Design*. 3d ed. Wellesley, MA: QED Information Sciences, Inc., 1989.
- Houghton, Raymond C., Jr. "Online Help Systems: A Conspectus." *Communications of the ACM 27*(February 1984): 126-133.

■ Queipo, Larry. "User Expectations of Online Information." *IEEE Transactions on Professional Communications* PC 29 (December 1986): 11-15.

Creating the Help topic files

Probably the most time-consuming task in developing a Help system for your application is creating the Help topic files from which you compile the Help system. Help topic files are text files that define what the user sees when using the Help system. The topic files can define various kinds of information, such as an index to information on the system, a list of commands, or a description of how to perform a task.

Creating topic files entails writing the text that the user sees when using Help, and entering control codes that determine how the user can move from one topic to another. This section describes the following topics:

- Choosing an authoring tool
- Structuring Help topic files
- Coding Help topic files
- Managing Help topic files

Choosing an authoring tool

To write your text files, you will need a Rich Text Format (RTF) editor, which lets you create the footnotes, underlined text, and strikethrough or double-underlined text that indicate the control codes. These codes are described in the section titled "Coding Help Topic Files" on page 322. RTF capability allows you to insert the coded text required to define Help terms, such as jumps, keywords, and definitions.

Structuring Help topic files

A Help topic file typically contains multiple Help topics. To identify each topic within a file:

- Topics are separated by hard page breaks.
- Each topic accessible via a hypertext link must have a unique identifier, or context string.

- Each topic can have a title.
- Each topic can have a list of keywords associated with it.
- Each topic can have a build-tag indicator.
- Any topic can have an assigned browse sequence.

For information about inserting page breaks between topics, see the documentation for the editor you are using. For information about assigning context strings and titles to topics, see the following sections.

Coding Help topic files

The Help system uses control codes for particular purposes:

Table A.4
Help control codes

Control Code	Purpose
Asterisk (*) footnote	Build tag—Defines a tag that specifies topics the compiler conditionally builds into the system. Build tags are optional, but they must appear first in a topic when they are used.
Pound sign (#)	Context string—Defines a context string that uniquely identifies a topic. Because hypertext relies on links provided by context strings, topics without context strings can only be accessed using keywords or browse sequences.
Dollar sign ($) footnote	Title—Defines the title of a topic. Titles are optional.
Letter "K" footnote	Keyword—Defines a keyword the user uses to search for a topic. Keywords are optional.
Plus sign (+) footnote	Browse sequence number—Defines a sequence that determines the order in which the user can browse through topics. Browse sequences are optional. However, if you omit browse sequences, the Help window will still include the Browse buttons, but they will be grayed.
Strikethrough or double-underlined text	Cross-reference—Indicates the text the user can choose to jump to another topic.
Underlined text	Definition—Specifies that a temporary or "look-up" box be displayed when the user holds down the mouse button or *Enter* key. The box can include such information as the definition of a word or phrase, or a hint about a procedure.

Table A.4: Help control codes (continued)

Hidden text	Cross-reference context string—Specifies the context string for the topic that will be displayed when the user chooses the text that immediately precedes it.

If you are using build tags, footnote them at the very beginning of the topic. Place other footnotes in any order you want. For information about assigning specific control codes, see the following sections.

Assigning build tags

Build tags are strings that you assign to a topic in order to conditionally include or exclude that topic from a build. Each topic can have one or more build tags. Build tags are not a necessary component of your Help system. However, they do provide a means of supporting different versions of a Help system without having to create different source files for each version. Topics without build tags are always included in a build.

*For information about the **BUILD** option, the (BuildTags) section and the Help project file, see "Building the Help files."*

You insert build tags as footnotes using the asterisk (*). When you assign a build tag footnote to a topic, the compiler includes or excludes the topic according to build information specified in the **BUILD** option and [BuildTags] section of the Help project file.

To assign a build tag to a topic:

1. Place the cursor at the beginning of the topic heading line, so that it appears before all other footnotes for that topic.
2. Insert the asterisk (*) as a footnote reference mark.

   Note that a superscript asterisk (*) appears next to the heading.
3. Type the build tag name as the footnote.

   Be sure to allow only a single space between the asterisk (*) and the build tag.

Build tags can be made up of any alphanumeric characters. The build tag is not case-sensitive. The tag may not contain spaces. You can specify multiple build tags by separating them with a semicolon, as in the following example:

```
* AppVersion1; AppVersion2; Test_Build
```

Including a build tag footnote with a topic is equivalent to setting the tag to true when compared to the value set in the project file. The compiler assumes all other build tags to be false for that topic.

After setting the truth value of the build tag footnotes, the compiler evaluates the build expression in the Options section of the Help project file. Note that all build tags must be declared in the project file, regardless of whether a given conditional compilation declares the tags. If the evaluation results in a true state, the compiler includes the topic in the build. Otherwise, the compiler omits the topic.

The compiler includes in all builds topics that do not have a build tag footnote regardless of the build tag expressions defined in the Help project file. For this reason, you may want to use build tags primarily to exclude specific topics from certain builds. If the compiler finds any build tags not declared in the Help project file, it displays an error message.

By allowing conditional inclusion and exclusion of specific topics, you can create multiple builds using the same topic files. This saves time and effort for the Help development team. It also means that you can develop Help topics that will help you maintain a higher level of consistency across your product lines.

Assigning context strings

Context strings identify each topic in the Help system. Each context string must be unique. A given context string may be assigned to only one topic within the Help project; it cannot be used for any other topic.

For information about assigning jumps, see page 330; for assigning browse sequences, see page 328; for assigning keywords, see page 326.

The context string provides the means for creating jumps between topics or for displaying look-up boxes, such as word and phrase definitions. Though not required, most topics in the Help system will have context-string identifiers. Topics without context strings may not be accessed through hypertext jumps. However, topics without context-string identifiers can be accessed through browse sequences or keyword searches, if desired. It is up to the Help writer to justify the authoring of topics that can be accessed only in these manners.

To assign a context string to a Help topic:

1. Place the cursor to the left of the topic heading.
2. Insert the pound sign (#) as the footnote reference mark.

   Note that a superscript pound sign (#) appears next to the heading.
3. Type the context string as the footnote.

Be sure to allow only a single space between the pound sign (#) and the string.

Context strings are not case-sensitive.

Valid context strings may contain the alphabetic characters A – Z, the numeric characters 0 – 9, and the period (.) and underscore (_) characters. The following example shows the context string footnote that identifies a topic called "Opening an Existing Text File":

```
#OpeningExistingTextFile
```

Although a context string has a practical limitation of about 255 characters, there is no good reason for approaching this value. Keep the strings sensible and short so that they're easier to enter into the text files.

Assigning titles

Title footnotes perform the following functions within the Help system:

- They appear on the Bookmark menu.
- They appear in the "Topics found " list that results from a keyword search. (Topics that do not have titles, but are accessible via keywords are listed as `>>>>Untitled Topic<<<<` in the Topics found list.)

Although not required, most topics have a title. You might not assign a title to topics containing low-level information that Help's search feature, look-up boxes, and system messages do not access.

To assign a title to a topic:

1. Place the cursor to the left of the topic heading.
2. Insert a footnote with a dollar sign ($) as the footnote reference mark.

   Note that a superscript dollar sign ($^{\$}$) appears next to the heading.
3. Type the title as the footnote.

   Be sure to allow only a single space between the dollar sign ($) and the title.

The following is an example of a footnote that defines the title for a topic:

```
$ Help Keys
```

When adding titles, keep in mind the following restrictions:

Table A.5
Restrictions of Help titles

Item	Restrictions
Characters	Titles can be up to 128 characters in length. The Help compiler truncates title strings longer than 128 characters. The help system displays titles in a list box when the user searches for a keyword or enters a bookmark.
Formatting	Title footnote entries cannot be formatted.

Assigning keywords

Help allows the user to search for topics with the use of keywords assigned to the topics. When the user searches for a topic by keyword, Help matches the user-entered word to keywords assigned to specific topics. Help then lists matching topics by their titles in the Search dialog box. Because a keyword search is often a fast way for users to access Help topics, you'll probably want to assign keywords to most topics in your Help system.

Note You should specify a keyword footnote only if the topic has a title footnote, since the title of the topic will appear in the search dialog when the user searches for the keyword.

To assign a keyword to a topic:

1. Place the cursor to the left of the topic heading.
2. Insert an uppercase K as the footnote reference mark.

   Note that a superscript K (^{K}) appears next to the heading.
3. Type the keyword, or keywords, as the footnote.

   Be sure to allow only a single space between the K and the first keyword.

   If you add more than one keyword, separate each with a semi-colon.

The following is an example of keywords for a topic:

```
K open;opening;text file;ASCII;existing;text only;documents;
```

Whenever the user performs a search on any of these keywords, the corresponding titles appear in a list box. More than one topic may have the same keyword.

When adding keywords, keep in mind the following restrictions:

Table A.6
Help keyword restrictions

Item	Restrictions
Characters	Keywords can include any ANSI character, including accented characters. The maximum length for keywords is 255 characters. A space embedded in a key phrase is considered to be a character, permitting phrases to be keywords.
Phrases	Help searches for any word in the specified phrase.
Formatting	Keywords are unformatted.
Case sensitivity	Keywords are not case-sensitive.
Punctuation	Except for semicolon delimiters, you can use punctuation.

Creating multiple keyword tables

Multiple keyword tables are useful to allow a program to look up topics that are defined in alternate keyword tables. You can use an additional keyword table to allow users familiar with keywords in a different application to discover the matching keywords in your application.

*For information on the **MULTIKEY** option, see page 344.*

Authoring additional keyword tables is a two-part process. First, the **MULTIKEY** option must be placed in the [Options] section of the project file.

Second, the topics to be associated with the additional keyword table must be authored and labeled. Footnotes are assigned in the same manner as the normal keyword footnotes, except that the letter specified with the **MULTIKEY** option is used. With this version of the Help Compiler, the keyword footnote used is case-sensitive. Therefore, care should be taken to use the same case, usually uppercase, for your keyword footnote. Be sure to associate only one topic with a keyword. Help does not display the normal search dialog box for a multiple keyword search. Instead it displays the first topic found with the specified keyword. If you want the topics in your additional keyword table to appear in the normal Help keyword table, you must also specify a "K" footnote and the given keyword.

The application you are developing Help for can then display the Help topic associated with a given string in a specified keyword table. Keywords are sorted without regard to case for the keyword table. For information on the parameters passed by the

application to call a topic found in alternate keyword table, see page 355.

Assigning browse sequence numbers

The `Browse >>>>` and `Browse <<<<` buttons on the icon bar in the Help window let users move back and forth between related topics. The order of topics that users follow when moving from topic to topic is called a "browse sequence." A browse sequence is determined by sequence numbers, established by the Help writer.

To build browse sequences into the Help topics, the writer must

1. Decide which topics should be grouped together and what order they should follow when viewed as a group.

   Help supports multiple, discontiguous sequence lists.
2. Code topics to implement the sequence.

In this version of Help, topics defined in browse sequences are accessed using the Browse buttons at the top of the Help window. Future versions of Help will not normally display browse buttons for the user. However, if your Help resource file includes browse sequences authored in the format described here, these future versions will support the feature by automatically displaying browse buttons for the user.

Organizing browse sequences

When organizing browse sequences, the writer must arrange the topics in an order that will make sense to the user. Topics can be arranged in alphabetical order within a subject, in order of difficulty, or in a sensible order that seems natural to the application. The following example illustrates browse sequences for the menu commands used in a given application. The Help writer has subjectively defined the order that makes the most sense from a procedural point of view. You may, of course, choose a different order.

```
SampleApp Commands
     File Menu - commands:005
          New Command - file_menu:005
          Open Command - file_menu:010
          Save Command - file_menu:015
          Save As Command - file_menu:020
          Print Command - file_menu:025
          Printer Setup Command - file_menu:030
          Exit Command - file_menu:035
```

```
Edit Menu - commands:010
     Undo Command - edit_menu:025
     Cut Command - edit_menu:015
     Copy Command - edit_menu:010
     Paste Command - edit_menu:020
     Clear Command - edit_menu:005
     Select All Command - edit_menu:030
     Word Wrap Command - edit_menu:035
     Type Face Command - edit_menu:040
     Point Size Command - edit_menu:045
Search Menu - commands:015
     Find Command - search_menu:005
     Find Next Command - search_menu:010
     Previous Command - search_menu:015
Window Menu - commands:020
     Tile Command - window_menu:005
     Cascade Command - window_menu:010
     Arrange Icons Command - window_menu:015
     Close All Command - window_menu:020
     Document Names Command - window_menu:025
```

Each line consists of a sequence list name followed by a colon and a sequence number. The sequence list name is optional. If the sequence does not have a list name, as in the following example, the compiler places the topic in a "null" list:

```
Window Menu - 120
```

Note that the numbers used in the browse sequence example begin at 005 and advance in increments of 005. Generally, it is good practice to skip one or more numbers in a sequence so you can add new topics later if necessary. Skipped numbers are of no consequence to the Help Compiler; only their order is significant.

Sequence numbers establish the order of topics within a browse sequence list. Sequence numbers can consist of any alphanumeric characters. During the compiling process, strings are sorted using the ASCII sorting technique, not a numeric sort.

Both the alphabetic and numeric portions of a sequence can be several characters long; however, their lengths should be consistent throughout all topic files. If you use only numbers in the strings make sure the strings are all the same length; otherwise a higher sequence number could appear before a lower one in certain cases. For example, the number 100 is numerically higher than 99, but 100 will appear before 99 in the sort used by Help, because Help is comparing the first two digits in the strings.

In order to keep the topics in their correct numeric order, you would have to make 99 a three-digit string: 099.

Coding browse sequences

After determining how to group and order topics, code the sequence by assigning the appropriate sequence list name and number to each topic, as follows:

1. Place the cursor to the left of the topic heading.
2. Insert the plus sign (+) as the footnote reference mark.

   Note that a superscript plus sign ($^+$) appears next to the heading.
3. Type the sequence number using alphanumeric characters.

For example, the following footnote defines the browse sequence number for the Edit menu topic in the previous browse sequence example:

```
+ commands:010
```

While it may be easier to list topics within the file in the same order that they appear in a browse sequence, it is not necessary. The compiler orders the sequence for you.

Creating cross-references between topics

Cross-references, or "jumps," are specially-coded words or phrases that are linked to other topics. Although you indicate jump terms with strikethrough or double-underlined text in the topic file, they appear underlined in the Help window. In addition, jump terms appear in color on color systems. For example, the strikethrough text (double-underlined in Word for Windows) ~~**New Command**~~ appears as **New Command** in green text to the user.

To code a word or phrase as a jump in the topic file :

1. Place the cursor at the point in the text where you want to enter the jump term.
2. Select the strikethrough (or double-underline) feature of your editor.
3. Type the jump word or words in strikethrough mode.
4. Turn off strikethrough and select the editor's hidden text feature.

5. Type the context string assigned to the topic that is the target of the jump.

When coding jumps, keep in mind that:

- No spaces can occur between the strikethrough (or double-underlined) text and the hidden text.
- Coded spaces before or after the jump term are not permitted.
- Paragraph marks must be entered as plain text.

Defining terms

Most topic files contain words or phrases that require further definition. To get the definition of a word or phrase, the user first selects the word and then holds down the mouse button or *Enter* key, causing the definition to appear in a box within the Help window. The Help writer decides which words to define, considering the audience that will be using the application and which terms might already be familiar.

The look-up feature need not be limited to definitions. With the capability of temporarily displaying information in a box, you might want to show a hint about a procedure, or other suitable information for the user.

Defining a term requires that you

- Create a topic that defines the term.

  The definition topic must include a context string. See the section titled "Assigning Context Strings." on page 324.
- Provide a cross-reference for the definition topic whenever the term occurs.

  You don't need to define the same word multiple times in the same topic, just its first occurrence. Also, consider the amount of colored text you are creating in the Help window. See the following "Coding definitions" section.

Creating definition topics

You can organize definition topics any way you want. For example, you can include each definition topic in the topic file that mentions the term. Or you can organize all definitions in one topic file and provide the user with direct access to it. Helpex uses the latter method, with all definitions residing in the TERMS.RTF file. Organizing definition topics into one file provides you with a glossary and lets you make changes easily.

Coding definitions

After creating definition topics, code the terms as they occur, as follows:

1. Place the insertion point where you want to place the term that requires definition.
2. Select the underline feature of your editor.
3. Type the term.
4. Turn off the underline feature, and select the editor's hidden-text feature.
5. Type the context string assigned to the topic that contains the definition of the term.

Inserting graphic images

Bitmapped graphic images can be placed in Help topics using either of two methods. If your word processor supports the placement of Windows 2.1 or Windows 3.0 graphics directly into a document, you can simply paste your bitmaps into each topic file. Alternatively, you can save each bitmap in a separate file and specify the file by name where you want it to appear in the Help topic file. The latter method of placing graphics is referred to as "bitmaps by reference." The following sections describe the process of placing bitmaps directly or by reference into your Help topics.

Creating and capturing bitmaps

You can create your bitmaps using any graphical tools, as long as the resulting images can be displayed in the Windows environment. Each graphic image can then be copied to the Windows clipboard. Once on the clipboard, a graphic can be pasted into a graphics editor such as Paint, and modified or cleaned up as needed.

Windows Help 3.0 supports color bitmaps. However, for future compatibility, you might want to limit graphics to monochrome format. If you are producing monochrome images, you might have to adjust manually the elements of your source graphic that were originally different colors to either black, white, or a pattern of black and white pixels.

When you are satisfied with the appearance of your bitmap, you can either save it as a file, to be used as a bitmap by reference, or you can copy it onto the clipboard and paste it into your word

processor. If you save the graphic as a file, be sure to specify its size in your graphics editor first, so that only the area of interest is saved for display in the Help window. The tighter you crop your images, the more closely you will be able to position text next to the image. Always save (or convert and save if necessary) graphics in Windows' .BMP format.

Bitmap images should be created in the same screen mode that you intend Help to use when topics are displayed. If your Help files will be displayed in a different mode, bitmaps might not retain the same aspect ratio or information as their source images.

Placing bitmaps using a graphical word processor

The easiest way to precisely place bitmaps into Help topics is to use a graphical word processor. Microsoft Word for Windows supports the direct importation of bitmaps from the clipboard. Simply paste the graphic image where you want it to appear in the Help topic. You can format your text so that it is positioned below or alongside the bitmap. When you save your Help topic file in RTF, the pasted-in bitmap is converted as well and will automatically be included in the Help resource file.

Placing bitmaps by reference

If your word processor cannot import and display bitmaps directly, you can specify the location of a bitmap that you have saved as a file. To insert a bitmap reference in the Help topic file, insert one the following statements where you want the bitmap to appear in the topic:

{**bmc** *filename*.bmp}
{**bml** *filename*.bmp}
{**bmr** *filename*.bmp}

Do not specify a full path for *filename*. If you need to direct the compiler to a bitmap in a location other than the root directory for the build, specify the absolute path for the bitmap in the [Bitmaps] section of the project file.

The argument **bmc** stands for "bitmap character," indicating that the bitmap referred to will be treated the same as a character placed in the topic file at the same location on a line. Text can precede or follow the bitmap on the same line, and line spacing will be determined based upon the size of the characters (including the bitmap character) on the line. Don't specify negative line spacing for a paragraph with a bitmap image, or the image may inadvertently overwrite text above it when it's displayed in Help. When you use the argument **bmc**, there is no

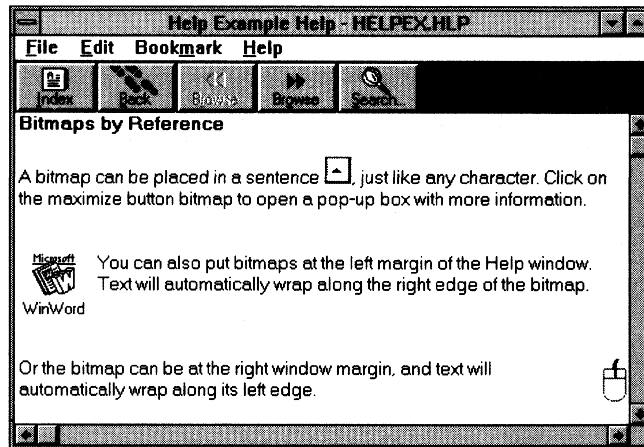
automatic text wrapping around the graphic image. Text will follow the bitmap, positioned at the baseline.

The argument **bml** specifies that the bitmap appear at the left margin, with text wrapping automatically along the right edge of the image. The argument **bmr** specifies that the bitmap appear at the right margin, with text to its left. Bitmap filenames must be the same as those listed in the [Bitmaps] section of the Help project file. The [Bitmaps] section is described in the section "Building the Help files."

Multiple references to a bitmap of the same name refer to the same bitmap when the Help file is displayed. This means that bitmap references can be repeated in your Help system without markedly increasing the size of the Help resource file.

Figure A.6 shows the placement of three bitmaps with related text in a topic as displayed in Help.

Figure A.6
Help topic display showing bitmaps by reference



Managing topic files

Help topic files can be saved in the default word-processor format or in RTF. If you always save your files in RTF, and later need to make a change, the word processor may take additional time to interpret the format as it reloads the file. If you anticipate making numerous changes during Help development, you might want to minimize this delay by saving topic files in both default and RTF formats, with different file extensions to distinguish them. The compiler needs only the RTF files, and you will have faster access

to the source files for changes. On a large project, this practice can save a considerable amount of development time.

Keeping track of files and topics

It is important to keep track of all topic files for the following reasons:

- To ensure that no topics are left out of the build process
- To ensure that each topic has been assigned a unique context string
- To double-check browse sequencing within general and specific lists
- To show keyword and title matches
- To allow writers to see where the text for each of the topics is located
- To keep track of changes to files and the current status
- To track any other aspect of the Help development process that you think essential

At a minimum, writers must keep track of their own topic files, and must pass the filenames to the person who is responsible for creating the Help project file.

Creating a help tracker

While it is important that you track topic files throughout the development cycle, the tracking tool can be anything that suits your needs. You can maintain a current list of topics in an ASCII text file, in a Quattro Pro spreadsheet, or in another format.

When you or another writer creates or revises a topic, you should update the Help tracking file to reflect the change. The contents of the tracking file are not rigidly defined, but should contain entries for filename, context string, title, browse sequence, and keywords. If your application makes use of the context-sensitive feature of Help, you may want to keep track of the context-sensitive information as well. This entry is necessary only if you are assigning context numbers to topics in the Help project file. You can also include optional information, such as date created, date modified, status, and author, if you want to keep track of all aspects of the Help development process. How you organize this information is entirely up to you.

The following sample text file and worksheet illustrate how the tracker might be organized for the Help system topics. The examples show both Help menu and context-sensitive Help

entries for the topic files. Typically, the same topics that the user accesses when choosing commands from the Help menus can be accessed by the context-sensitive Help feature. The topics with entries in the context ID column are used for context-sensitive help as well as for the Help menus. Notice that some topics have more than one context-sensitive help number. This enables the topic to be displayed when the user clicks on different regions of the screen. Of course, you're free to keep track of your topic files in any manner you choose.

Figure A.7
Help tracker text file example

Ctx. String	Title	Browse Seq.	Key Words	Ctx. No.	Filename	Modified	Status
hlpidx_id_mp	Multipad Help Index			0xFFFF	helpex.idx	5/16/89	Done
mc_cmd_mp	Multipad Commands	commands:0001	commands; menus	0x1000	helpex.cmd	5/16/89	Done
fm_cmd_mp	File Menu	commands:0004	commands; menus; files; documents	0x1001	helpex.cmd	5/16/89	Done
nc_cmd_mp	New Command	commands:0008	commands; new files; new documents	0x1002	helpex.cmd	5/16/89	Done
oc_cmd_mp	Open Command	commands:0012	commands; file; open; documents; read only	0x1003	helpex.cmd	5/16/89	Test
sac_cmd_mp	Save Command	commands:0016	commands; file; save; save as; documents; files	0x1004	helpex.cmd	5/16/89	Done
sasc_cmd_mp	Save As Command	commands:0020	commands; file; save as; save; documents; files	0x1005	helpex.cmd	5/16/89	Done
ptc_cmd_mp	Print Command	commands:0024	commands; file; print; documents; files	0x1006	helpex.cmd	5/16/89	Done
psc_cmd_mp	Print Setup Command	commands:0028	commands; file; printer setup; print	0x1007	helpex.cmd	5/16/89	Debug
ec_cmd_mp	Exit Command	commands:0032	commands; file; exit; exiting; close; closing; quit; quitting	0x1008	helpex.cmd	5/16/89	Done
em_cmd_mp	Edit Menu	commands:0036	commands; menus; editing; documents	0x1009	helpex.cmd	5/16/89	Done
uc_cmd_mp	Undo Command	commands:0040	commands; edit; editing; undo; typing	0x100A	helpex.cmd	5/16/89	Done
ctc_cmd_mp	Cut Command	commands:0044	commands; edit; editing; cut; cutting; text; Clipboard	0x100B	helpex.cmd	5/16/89	Done
cyc_cmd_mp	Copy Command	commands:0048	commands ; edit; editing; copy; copying; text; Clipboard	0x100C	helpex.cmd	5/16/89	Done
pec_cmd_mp	Paste Command	commands:0052	commands; edit; editing; paste; insert; inserting; text; Clipboard	0x100D	helpex.cmd	5/16/89	Done
crc_cmd_mp	Clear Command	commands:0056	commands; edit; editing; clear; text	0x100E	helpex.cmd	5/16/89	Done
salc_cmd_mp	Select All Command	commands:0060	commands; edit; editing; select all; select; selecting; text	0x100F	helpex.cmd	5/16/89	Done
wwc_cmd_mp	Word Wrap Command	commands:0064	commands; edit; editing; word wrap; wrapping; text; format	0x1010	helpex.cmd	5/16/89	Done
tfc_cmd_mp	Type Face Command	commands:0068	commands; edit; editing; type face; font; text; format	0x1011	helpex.cmd	5/16/89	Test
ptsc_cmd_mp	Point Size Command	commands:0072	commands; edit; editing; point size; text; format	0x1012	helpex.cmd	5/16/89	Test

Figure A.8
Help tracker worksheet example

	A	B	C	D	E	F	G	H
1	**Ctx. String**	**Text**	**Browse Seq.**	**Key Words**	**Ctx. No.**	**Filename**	**Modified**	**Status**
2	hlpidx_id_mp	Multipad Help Index			0xFFFF	helpex.idx	5/16/89	Done
3	mc_cmd_mp	Multipad Commands	commands:0001	commands; menus	0x1000	helpex.cmd	5/16/89	Done
4	fm_cmd_mp	File Menu	commands:0004	commands; menus; files;	0x1001	helpex.cmd	5/16/89	Done
5				documents				
6	nc_cmd_mp	New Command	commands:0008	commands; new files;	0x1002	helpex.cmd	5/16/89	Done
7				new documents				
8	oc_cmd_mp	Open Command	commands:0012	commands; file; open;	0x1003	helpex.cmd	5/16/89	Test
9				documents; read only				
10	sac_cmd_mp	Save Command	commands:0016	commands; file; save; save as;	0x1004	helpex.cmd	5/16/89	Done
11				documents; files				
12	sasc_cmd_mp	Save As Command	commands:0020	commands; file; save as; save;	0x1005	helpex.cmd	5/16/89	Done
13				documents; files				
14	ptc_cmd_mp	Print Command	commands:0024	commands; file; print;	0x1006	helpex.cmd	5/16/89	Done
15				documents; files				
16	psc_cmd_mp	Print Setup Command	commands:0028	commands; file; printer setup;	0x1007	helpex.cmd	5/16/89	Debug
17				print				
18	ec_cmd_mp	Exit Command	commands:0032	commands; file; exit; exiting;	0x1008	helpex.cmd	5/16/89	Done
19				close; closing; quit; quitting				
20	em_cmd_mp	Edit Menu	commands:0036	commands; menus; editing;	0x1009	helpex.cmd	5/16/89	Done
21				documents				
22	uc_cmd_mp	Undo Command	commands:0040	commands; edit; editing; undo;	0x100A	helpex.cmd	5/16/89	Done
23				typing				
24	ctc_cmd_mp	Cut Command	commands:0044	commands; edit; editing; cut;	0x100B	helpex.cmd	5/16/89	Done
25				cutting; text; Clipboard				
26	cyc_cmd_mp	Copy Command	commands:0048	commands ; edit; editing; copy;	0x100C	helpex.cmd	5/16/89	Done
27				copying; text; Clipboard				
28	pec_cmd_mp	Paste Command	commands:0052	commands; edit; editing; paste;	0x100D	helpex.cmd	5/16/89	Done
29				insert; inserting; text; Clipboard				
30	crc_cmd_mp	Clear Command	commands:0056	commands; edit; editing; clear;	0x100E	helpex.cmd	5/16/89	Done
31				text				
32	salc_cmd_mp	Select All Command	commands:0060	commands; edit; editing; select all;	0x100F	helpex.cmd	5/16/89	Done
33				select; selecting; text				
34	wwc_cmd_mp	Word Wrap Command	commands:0064	commands; edit; editing; word wrap;	0x1010	helpex.cmd	5/16/89	Done
35				wrapping; text; format				
36	tfc_cmd_mp	Type Face Command	commands:0068	commands; edit; editing; type face;	0x1011	helpex.cmd	5/16/89	Test
37				font; text; format				
38	ptsc_cmd_mp	Point Size Command	commands:0072	commands; edit; editing; point size;	0x1012	helpex.cmd	5/16/89	Test
39				text; format				

Building the Help file

While the examples in this chapter are in C, you can also do the same tasks in Turbo Pascal. Code examples for both languages are included on your Borland product disks.

After the topic files for your Help system have been written, you are ready to create a Help project file and run a build to test the Help file. The Help project file contains all information the compiler needs to convert help topic files into a binary Help resource file.

You use the Help project file to tell the Help Compiler which topic files to include in the build process. Information in the Help project file also enables the compiler to map specific topics to context numbers (for the context-sensitive portion of Help).

After you have compiled your Help file, the development team programs the application so the user can access it.

This section describes the following:

- Creating a Help project file
- Compiling the Help file
- Programming the application to access Help

Creating the Help project file

You use the Help project file to control how the Help Compiler builds your topic files. The Help project file can contain up to six sections that perform the following functions:

Table A.7
Help project file sections

Section	**Function**
[Files]	Specifies topic files to be included in the build. This section is mandatory.
[Options]	Specifies the level of error reporting, topics to be included in the build, the directory in which to find the files, and the location of your Help index. This section is optional.
[BuildTags]	Specifies valid build tags. This section is optional.
[Alias]	Assigns one or more context strings to the same topic. This section is optional.
[Map]	Associates context strings with context numbers. This section is optional.
[Bitmaps]	Specifies bitmap files to be included in the build. This section is optional.

You can use any ASCII text editor to create your Help project file. The extension of a Help project file is .HPJ. If you do not use the extension .HPJ on the **HC** command line, the compiler looks for a project file with this extension before loading the file. The .HLP output file will have the same name as the .HPJ file.

The order of the sections within the Help project file is arbitrary, except: that an [Alias] section must always precede the [Map] section (if an [Alias] section is used).

Section names are placed within square brackets using the following syntax:

[*section-name*]

You can use a semicolon to indicate a comment in the project file. The compiler ignores all text from the semicolon to the end of the line on which it occurs.

Specifying topic files

Use the [Files] section of the Help project file to list all topic files that the Help Compiler should process to produce a Help resource file. A Help project file must have a [Files] section.

The following sample shows the format of the [Files] section:

```
[FILES]
HELPEX.RTF  ;Main topics for HelpEx application
TERMS.RTF   ;Lookup terms for HelpEx application
```

For more information about the ***ROOT*** *option, see the section titled "Specifying the Root Directory: The Root Option."*

Using the path defined in the **ROOT** option, the Help Compiler finds and processes all files listed in this section of the Help project file. If the file is not on the defined path and cannot be found, the compiler generates an error.

You can include files in the build process using the C **#include** directive command. The **#include** directive uses this syntax:

#include <*filename*>

You must include the angle brackets around the filename. The pound sign (#) must be the first character in the line. The filename must specify a complete path, either the path defined by the **ROOT** option or an absolute directory path to the file.

You may find it easier to create a text file that lists all files in the Help project and include that file in the build, as in this example:

```
[FILES]
#include <hlpfiles.inc>
```

Specifying build tags

If you code build tags in your topic files, use the [BuildTags] section of the Help project file to define all the valid build tags for a particular Help project. The [BuildTags] section is optional.

The following example shows the format of the [BuildTags] section in a sample Help project file:

```
[BUILDTAGS]
WINENV              ;topics to include in Windows build
DEBUGBUILD          ;topics to include in debugging build
TESTBUILD           ;topics to include in a mini-build for testing
```

For information about coding build tags in topic files, see page 323.

The [BuildTags] section can include up to 30 build tags. The build tags are not case-sensitive and may not contain space characters. Only one build tag is allowed per line in this section. The compiler will generate an error message if anything other than a comment is listed after a build tag in the [BuildTags] section.

Specifying options

Use the [Options] section of the Help project file to specify the following options:

Table A.8
The Help (Options) options

Option	Meaning
BUILD	Determines what topics the compiler includes in the build.
COMPRESS	Specifies compression of the Help resource file.
FORCEFONT	Specifies the creation of a Help resource file using only one font.
INDEX	Specifies the context string of the Help index.
MAPFONT SIZE	Determines the mapping of specified font sizes to different sizes.
MULTIKEY	Specifies alternate keyword mapping for topics.
ROOT	Designates the directory to be used for the Help build.
TITLE	Specifies the title shown for the Help system.
WARNING	Indicates the kind of error message the compiler reports.

These options can appear in any order within the [Options] section. The [Options] section is not required.

Detailed explanations of the available options follow.

Specifying error reporting

Use the **WARNING** option to specify the amount of debugging information that the compiler reports. The **WARNING** option has the following syntax:

WARNING = *level*

You can set the **WARNING** option to any of the following levels:

Table A.9
WARNING levels

Level	Information Reported
1	Only the most severe warnings.
2	An intermediate level of warnings.
3	All warnings. This is the default level if no **WARNING** option is specified.

The following example specifies an intermediate level of error reporting:

```
[OPTIONS]
WARNING=2
```

Use the DOS Ctrl+PrtSc accelerator key before you begin your compilation to echo errors which appear on the screen to your printer. Type Ctrl+PrtSc again to stop sending information to the printer.

The compiler reports errors to the standard output file, typically the screen. You may want to redirect the errors to a disk file so that you can browse it when you are debugging the Help system. The following example shows the redirection of compiler screen output to a file.

```
HC HELPEX >> errors.out
```

Specifying build topics

If you have included build tags in your topic files, use the **BUILD** option to specify which topics to conditionally include in the build. If your topic files have no build tags, omit the **BUILD** option from the [Options] section.

All build tags must be listed in the [BuildTags] section of the project file, regardless whether or not a given conditional compilation declares the tags.

See "Creating the Help topic files" on page 321 for information on assigning build tags to topics in the Help topic files.

The **BUILD** option line uses the following syntax:

BUILD = *expression*

Build expressions cannot exceed 255 characters in length, and must be entered on only one line. Build expressions use Boolean

logic to specify which topics within the specified Help topic files the compiler will include in the build. The compiler evaluates all build expressions from left to right. The tokens of the language (listed in order of precedence from highest to lowest) are:

Table A.10 Build tag order of precedence

Token	Description
<tag>	Build tag
( )	Parentheses
~	NOT operator
&	AND operator
|	OR operator

For example, if you coded build tags called WINENV, APP1, and TEST_BUILD in your topic files, you could include one of the following build expressions in the [Options] section:

Table A.11: Build expression examples

Build expression	Topics built
BUILD = WINENV	Only topics that have the WINENV tag
BUILD = WINENV & APP1	Topics that have both the WINENV and APP1 tags
BUILD = WINENV | APP1	Topics that have either the WINENV tag or the APP1 tag
BUILD = (WINENV | APP1) & TESTBUILD	Topics that have either the WINENV or the APP1 tags and that also have the TESTBUILD tag
BUILD =~ APP1	Topics that do not have an APP1 tag

Specifying the root directory

Use the **ROOT** option to designate the root directory of the Help project. The compiler searches for files in the specified root directory.

The **ROOT** option uses the following syntax:

ROOT = *pathname*

For example, the following root option specifies that the root directory is \BUILD\TEST on drive D:

```
[OPTIONS]
ROOT=D:\BUILD\TEST
```

The **ROOT** option allows you to refer to all relative paths off the root directory of the Help project. For example, the following entry in the [Files] section refers to a relative path off the root directory:

```
TOPICS\FILE.RTF
```

To refer to a file in a fixed location, independent of the project root, you must specify a fully qualified or "absolute" path, including a drive letter, if necessary, as in the following line:

```
D:\HELPTEST\TESTFILE.RTF
```

If you do not include the **ROOT** option in your Help project file, all paths are relative to the current DOS directory.

Specifying the index

Use the **INDEX** option to identify the context string of the Help index. Because the Index button gives the user access to the index from anywhere in the Help system, you will probably not want to author terms to jump to the index. Users access this general index either from the Help menu of the application or by choosing the Index button from the Help window.

Assigning a context string to the index topic in the [Options] section lets the compiler know the location of the main index of Help topics for the application's Help file. If you do not include the **INDEX** option in the [Options] section, the compiler assumes that the first topic it encounters is the index.

The **INDEX** option uses the following syntax:

INDEX = *context-string*

For information on assigning context strings, see page 324.

The context string specified must match the context string you assigned to the Help index topic. In the following example, the writer informs the compiler that the context string of the Help index is "main_index":

```
[OPTIONS]
INDEX=main_index
```

Assigning a title to the Help system

You can assign a title to your Help system with the **TITLE** option. The title appears in the title bar of the Help window with the word "Help" automatically appended, followed by the DOS filename of the Help resource file.

The **TITLE** option uses the following syntax:

TITLE = *Help-system-title-name*

Titles are limited to 32 characters in length. If you do not specify a title using the **TITLE** option, only the word Help followed by the Help system filename will be displayed in the title bar. Because the compiler always inserts the word Help, don't duplicate it in your title.

Converting fonts

You can use the **FORCEFONT** option to create a Help resource file that is made up of only one typeface or font. This is useful if you must compile a Help system using topic files that include fonts not supported by your users' systems.

The **FORCEFONT** option uses the following syntax:

FORCEFONT = *fontname*

See page 320 for a list of the fonts Windows ships with.

The *fontname* parameter is any Windows system font. Note that the *fontname* used in the **FORCEFONT** option cannot contain spaces. Therefore, Tms Rmn font cannot be used with **FORCE-FONT**.

Font names must be spelled the same as they are in the Fonts dialog box in Control Panel. Font names do not exceed 20 characters in length. If you designate a font that is not recognized by the compiler, an error message is generated and the compilation continues using the default Helvetica (Helv) font.

Changing font sizes

The font sizes specified in your topic files can be mapped to different sizes using the **MAPFONTSIZE** option. In this manner, you can create and edit text in a size chosen for easier viewing in the topic files and have them sized by the compiler for the actual Help display. This may be useful if there is a large size difference between your authoring monitor and your intended display monitor.

The **MAPFONTSIZE** option uses the following syntax:

MAPFONTSIZE = *m*[–*n*]:*p*

The *m* parameter is the size of the source font, and the *p* parameter is the size of the desired font for the Help resource file. All fonts in the topic files that are size *m* are changed to size *p*. The optional parameter *n* allows you to specify a font range to be mapped. All fonts in the topic files falling between *m* and *n*, inclusive, are changed to size *p*. The following examples illustrate the use of the **MAPFONTSIZE** option:

```
MAPFONTSIZE=12-24:16 ;make fonts from 12 to 24 come out 16.
MAPFONTSIZE=8:12  ;make all size 8 fonts come out size 12.
```

Note that you can map only one font size or range with each **MAPFONTSIZE** statement used in the Options section. If you use more than one **MAPFONTSIZE** statement, the source font size or

range specified in subsequent statements cannot overlap previous mappings. For instance, the following mappings would generate an error when the compiler encountered the second statement:

```
MAPFONTSIZE=12-24:16 MAPFONTSIZE=14:20
```

Because the second mapping shown in the first example contains a size already mapped in the preceding statement, the compiler will ignore the line. There is a maximum of five font ranges that can be specified in the project file.

Multiple keyword tables

The **MULTIKEY** option specifies a character to be used for an additional keyword table.

The **MULTIKEY** option uses the following syntax:

MULTIKEY = *footnote-character*

The *footnote-character* parameter is the case-sensitive letter to be used for the keyword footnote. The following example illustrates the enabling of the letter *L* for a keyword-table footnote:

```
MULTIKEY=L
```

You must be sure to limit your keyword-table footnotes to one case, usually uppercase. In the previous example, topics with the footnote *L* would have their keywords incorporated into the additional keyword table, whereas those assigned the letter *l* would not.

You may use any alphanumeric character for a keyword table except *K* and *k*, which are reserved for Help's normal keyword table. There is an absolute limit of five keyword tables, including the normal table. However, depending upon system configuration and the structure of your Help system, a practical limit of only two or three may actually be the case. If the compiler cannot create an additional table, the excess table is ignored in the build.

Compressing the file

You can use the **COMPRESS** option to reduce the size of the Help resource file created by the compiler. The amount of file compression realized will vary according to the number, size and complexity of topics that are compiled. In general, the larger the Help files, the more they can be compressed.

The **COMPRESS** option uses the following syntax:

COMPRESS = TRUE | FALSE

Because the Help application can load compressed files quickly, there is a clear advantage in creating and shipping compressed Help files with your application. Compiling with compression turned on, however, may increase the compile time, because of the additional time required to assemble and sort a key-phrase table. Thus, you may want to compile without compression in the early stages of a project.

The **COMPRESS** option causes the compiler to compress the system by combining repeated phrases found within the source file(s). The compiler creates a phrase-table file with the .PH extension if it does not find one already in existence. If the compiler finds a file with the .PH extension, it will use the file for the current compilation. This is in order to speed compression when not a lot of text has changed since the last compilation.

Deleting the key-phrase file before each compilation will prevent the compiler from using the previous file. Maximum compression will result only by forcing the compiler to create a new phrase table.

Specifying alternate context strings

Use the [Alias] section to assign one or more context strings to the same topic alias. Because context strings must be unique for each topic and cannot be used for any other topic in the Help project, the [Alias] section provides a way to delete or combine Help topics without recoding your files. The [Alias] section is optional.

For example, if you create a topic that replaces the information in three other topics, and you delete the three, you will have to search through your files for invalid cross-references to the deleted topics. You can avoid this problem by using the [Alias] section to assign the name of the new topic to the deleted topics. You can also use the [Alias] section when your application program has multiple context identifiers for which you have only one topic. This can be the case with context-sensitive Help.

Each expression in the [Alias] section has the following format:

context_string=alias

In the alias expression, the *alias* parameter is the alternate string, or alias name, and the *context_string* parameter is the context string identifying a particular topic. An alias string has the same format and follows the same conventions as the topic context string. That is, it is not case-sensitive and may contain the

alphabetic characters A – Z, numeric characters 0 – 9, and the period and underscore characters.

The following example illustrates an [Alias] section:

```
[ALIAS]
sm_key=key_shrtcuts
cc_key=key_shrtcuts
st_key=key_shrtcuts;combined into keyboard shortcuts topic
clskey=us_dlog_bxs
maakey=us_dlog_bxs;covered in using dialog boxes topic
chk_key=dlogprts
drp_key=dlogprts
lst_key=dlogprts
opt_key=dlogprts
tbx_key=dlogprts;combined into parts of dialog box topic
frmtxt=edittxt
wrptxt=edittxt
seltxt=edittxt;covered in editing text topic
```

You can use alias names in the [Map] section of the Help project file. If you do, however, the [Alias] section must precede the [Map] section.

Mapping context-sensitive topics

For more information on context-sensitive Help, see page 315.

If your Help system supports context-sensitive Help, use the [Map] section to associate either context strings or aliases to context numbers. The context number corresponds to a value the parent application passes to the Help application in order to display a particular topic. This section is optional.

When writing the [Map] section, you can do the following:

- Use either decimal or hexadecimal numbers formatted in standard C notation to specify context numbers.
- Assign no more than one context number to a context string or alias.
  Assigning the same number to more than one context string will generate a compiler error.
- Separate context numbers and context strings by an arbitrary amount of whitespace using either space characters or tabs.

You can use the C **#include** directive to include other files in the mapping. In addition, the Map section supports an extended format that lets you include C files with the .H extension directly.

Entries using this format must begin with the **#define** directive and may contain comments in C format, as in this example:

```
#define   context_string context_number   /* comment */
```

The following example illustrates several formats you can use in the [Map] section:

These eight entries give hexadecimal equivalents for the context numbers.

These five entries show decimal context numbers.

These five entries show how you might include topics defined in a C include file.

*This entry shows a C **#include** directive for some generic topics.*

```
[MAP]

Edit_Window          0x0001
Control_Menu         0x0002
Maximize_Icon        0x0003
Minimize_Icon        0x0004
Split_Bar            0x0005
Scroll_Bar           0x0006
Title_Bar            0x0007
Window_Border        0x0008

dcmb_scr             30; Document Control-menu Icon
dmxi_scr             31; Document Maximize Icon
dmni_scr             32; Document Minimize Icon
dri_scr              33; Document Restore Icon
dtb_scr              34; Document Title Bar

#define vscroll   0x010A   /* Vertical Scroll Bar */
#define hscroll   0x010E   /* Horizontal Scroll Bar */
#define scrollthm 0x0111   /* Scroll Thumb */
#define upscroll  0x0112   /* Up Scroll Arrow */
#define dnscroll  0x0113   /* Down Scroll Arrow */

#include <sample.h>
```

If context numbers use the **#define** directive, and the file containing the **#define** statements is included in both the application code and the Help file, then updates made to the context numbers by the application programmers will automatically be reflected in the next Help build.

You can define the context strings listed in the [Map] section either in a Help topic or in the [Alias] section. The compiler generates a warning message if a context string appearing in the [Map] section is not defined in any of the topic files or in the [Alias] section.

If you use an alias name, the [Alias] section must precede the [Map] section in the Help project file.

Including bitmaps by reference

If your Help system uses bitmaps by reference, the filenames of each of the bitmaps must be listed in the [Bitmaps] section of the project file. The following example illustrates the format of the [Bitmaps] section.

```
[BITMAPS]
DUMP01.BMP
DUMP02.BMP
DUMP03.BMP
c:\PROJECT\HELP\BITMAPS\DUMP04.BMP
```

The [Bitmaps] section uses the same rules as the [Files] section for locating bitmap files.

Compiling Help files

After you have created a Help project file, you are ready to build a Help file using the Help Compiler. The compiler generates the binary Help resource file from the topic files listed in the Help project file. When the build process is complete, your application can access the Help resource file that results.

Before initiating a build operation to create the Help file, consider the locations of the following files:

- The Help Compiler, HC.EXE. The compiler must be in a directory from which it can be executed. This could be the current working directory, on the path set with the PATH environment variable, or a directory specified by a full pathname, as follows:

  ```
  C:\BIN\HC HELPEX.HPJ
  ```

- The Help project file, *filename*.HPJ. The project file can be located either in the current directory or specified by a path, as follows:

  ```
  C:\BIN\HC D:\MYPROJ\HELPEX.HPJ
  ```

- The topic files named in the Help project file, saved as RTF. The topic files may be located in the current working directory, a subdirectory of the current working directory specified in the [Files] section, or the location specified in the Root option.
- Files included with the **#include** directive in the Help project file. Since the **#include** directive can take pathnames, then any number of places will work for these files.

- All bitmap files listed by reference in the topic files.

You must also place any files named in an **#include** directive in the path of the project root directory or specify their path using the **ROOT** option. The compiler searches only the directories specified in the Help project file. For information about the **ROOT** option, see the section titled "Specifying the Root Directory," on page 341.

If you use a RAM drive for temporary files (set with the DOS environment variable TMP), it must be large enough to hold the compiled Help resource file. If your Help file is larger than the size of the available RAM drive, the compiler will generate an error message and the compilation will be aborted.

Using the Help Compiler

To run the Help Compiler, use the **HC** command. There are no options for **HC**. All options are specified in the Help project file.

The **HC** command uses the following syntax:

HC *filename*.HPJ

As the compiler program runs, it displays sequential periods on the screen, indicating its progress in the process. Error messages are displayed when each error condition is encountered. When the Help Compiler has finished compiling, it writes a Help resource file with an .HLP extension in the current directory and returns to the DOS prompt. The Help resource file that results from the build has the same name as does the Help project file.

Compiler errors and status messages can be redirected to a file using standard DOS redirection syntax. This is useful for a lengthy build where you may not be monitoring the entire process. The redirected file is saved as a text file that can be viewed with any ASCII editor.

Programming the application to access Help

The application development team must program the application so that the user can access both the Windows Help application and your Help file. The Help application is a stand alone Windows application, and your application can ask Windows to run the Help application and specify the topic that Help is to show the user. To the user, Help appears to be part of your application, but it acts like any other Windows application.

Calling WinHelp from an application

An application makes a Help system available to the user by calling the **WinHelp** function.

The **WinHelp** function uses the following syntax:

The C and Pascal samples are provided on disk.

```
BOOL WinHelp (hWnd, lpHelpFile, wCommand, dwData)
```

The *hWnd* parameter identifies the window requesting Help. The Windows Help application uses this identifier to keep track of which applications have requested Help.

The *lpHelpFile* parameter specifies the name (with optional directory path) of the Help file containing the desired topic.

The *wCommand* parameter specifies either the type of search that the Windows Help application is to use to locate the specified topic, or that the application no longer requires Help. It may be set to any one of the following values:

Table A.12
wCommand values

Value	Meaning
HELP_CONTEXT	Displays Help for a particular topic identified by a context number.
HELP_HELPONHELP	Displays the Using Help index topic.
HELP_INDEX	Displays the main Help index topic.
HELP_KEY	Displays Help for a topic identified by a keyword.
HELP_MULTIKEY	Displays Help for a topic identified by a keyword in an alternate keyword table.
HELP_QUIT	Informs the Help application that Help is no longer needed. If no other applications have asked for Help, Windows closes the Help application.
HELP_SETINDEX	Displays a designated Help index topic.

The *dwData* parameter specifies the topic for which the application is requesting Help. The format of *dwData* depends upon the value of *wCommand* passed when your application calls **WinHelp**. The following list describes the format of *dwData* for each value of*wCommand*.

Table A.13
dwData formats

***wCommand* value**	***dwData* format**
HELP_CONTEXT	An unsigned long integer containing the context number for the topic. Instead of using HELP_INDEX, HELP_CONTEXT can use the value –1.
HELP_HELPONHELP	Ignored.
HELP_INDEX	Ignored.
HELP_KEY	A long pointer to a string which contains a keyword for the desired topic.
HELP_MULTIKEY	A long pointer to the **MULTIKEYHELP** structure, as defined in WINDOWS.H. This structure specifies the table footnote character and the keyword.
HELP_QUIT	Ignored.
HELP_SETINDEX	An unsigned long integer containing the context number for the topic.

Because it can specify either a context number or a keyword, **WinHelp** supports both context-sensitive and topical searches of the Help file.

To ensure that the correct index remains set, the application should call **WinHelp** with *wCommand* set to HELP_SETINDEX (with *dwData* specifying the corresponding context identifier) following each call to **WinHelp** with a command set to HELP_CONTEXT. HELP_INDEX should never be used with HELP_SETINDEX.

Getting context-sensitive Help

Context-sensitive Help should be made available when a user wants to learn about the purpose of a particular window or control. For example, the user might pull down the File menu, select the Open command (by using the arrow keys), and then press *F1* to get Help for the command.

Implementing certain types of context-sensitive help requires advanced programming techniques. The Helpex sample application illustrates the use of two techniques. These techniques are described in the following sections.

Shift+F1 support

To implement a *Shift+F1* mode, Helpex responds to the *Shift+F1* accelerator key by calling **SetCursor** to change the shape of the cursor to an arrow pointer supplemented by a question mark.

```
case WM_KEYDOWN:
      if (wParam == VK_F1) {
          /* If Shift-F1, turn help mode on and set help
             cursor */

           if (GetKeyState(VK_SHIFT)) {
               bHelp = TRUE;
               SetCursor(hHelpCursor);
               return (DefWindowProc(hWnd, message,
                       wParam, lParam));
          }
          /* If F1 without shift, then call up help main
             index topic */
          else {
              WinHelp(hWnd,szHelpFileName,HELP_INDEX,0L);
          }
      }
      else if (wParam == VK_ESCAPE && bHelp) {
          /* Escape during help mode: turn help mode off */
          bHelp = FALSE;
          SetCursor((HCURSOR)GetClassWord(hWnd,GCW_HCURSOR));
      }

      break;
```

As long as the user is in Help mode (that is, until the user clicks the mouse or presses *Esc*), Helpex responds to WM_SETCURSOR messages by resetting the cursor to the arrow and question mark combination.

```
case WM_SETCURSOR:
      /* In help mode it is necessary to reset the cursor
         in response to every WM_SETCURSOR message.
         Otherwise, by default, Windows will reset the
         cursor to that of the window class. */

      if (bHelp) {
          SetCursor(hHelpCursor);
          break;
      }
      return (DefWindowProc(hWnd, message, wParam, lParam));
      break;

  case WM_INITMENU:
      if (bHelp) {
          SetCursor(hHelpCursor);
```

```
    }
    return (TRUE);
```

When the user is in *Shift+F1* Help mode and clicks the mouse button, Helpex will receive a WM_NCLBUTTONDOWN message if the click is in a nonclient area of the application window. By examining the *wParam* value of this message, the program can determine which context ID to pass to **WinHelp**.

```
case WM_NCLBUTTONDOWN:
  /* If we are in help mode (Shift+F1), then display
     context-sensitive help for nonclient area. */

  if (bHelp) {
    dwHelpContextId =
      (wParam == HTCAPTION)     ? (DWORD)HELPID_TITLE_BAR:
      (wParam == HTSIZE)        ? (DWORD)HELPID_SIZE_BOX:
      (wParam == HTREDUCE)      ? (DWORD)HELPID_MINIMIZE_ICON:
      (wParam == HTZOOM)        ? (DWORD)HELPID_MAXIMIZE_ICON:
      (wParam == HTSYSMENU)     ? (DWORD)HELPID_SYSTEM_MENU:
      (wParam == HTBOTTOM)      ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTBOTTOMLEFT)  ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTBOTTOMRIGHT) ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTTOP)         ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTLEFT)        ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTRIGHT)       ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTTOPLEFT)     ? (DWORD)HELPID_SIZING_BORDER:
      (wParam == HTTOPRIGHT)    ? (DWORD)HELPID_SIZING_BORDER:
      (DWORD)0L;

      if (!((BOOL)dwHelpContextId))
        return (DefWindowProc(hWnd, message, wParam, lParam));

      bHelp = FALSE;
      WinHelp(hWnd,szHelpFileName,HELP_CONTEXT,dwHelpContextId);
      break;
  }

  return (DefWindowProc(hWnd, message, wParam, lParam));
```

F1 support

Context-sensitive *F1* support for menus is relatively easy to implement in your application. If a menu is open and the user presses *F1* while one of the menu items is highlighted, Windows sends a WM_ENTERIDLE message to the application to indicate that the system is going back into an idle state after having determined that *F1* was not a valid key stroke for choosing a

menu item. You can take advantage of this idle state by looking at the keyboard state at the time of the WM_ENTERIDLE message.

If the *F1* key is down, then you can simulate the user's pressing the *Enter* key by posting a WM_KEYDOWN message using VK_RETURN. You don't really want your application to execute the menu command. What you do is set a flag (bHelp=TRUE) so that when you get the WM_COMMAND message for the menu item, you don't execute the command. Instead, the topic for the menu item is displayed by Windows Help.

The following code samples illustrate *F1* sensing for menu items.

```
case WM_ENTERIDLE:
  if ((wParam == MSGF_MENU) &&
      (GetKeyState(VK_F1) & 0x8000)) {
         bHelp = TRUE;
         PostMessage(hWnd, WM_KEYDOWN, VK_RETURN, 0L);
  }
  break;

  case WM_COMMAND:
    /* Was F1 just pressed in a menu, or are we in help mode
       (Shift+F1)? */

    if (bHelp) {
      dwHelpContextId =
        (wParam == IDM_NEW)?(DWORD)HELPID_FILE_NEW:
        (wParam == IDM_OPEN)? (DWORD)HELPID_FILE_OPEN:
        (wParam == IDM_SAVE)? (DWORD)HELPID_FILE_SAVE:
        (wParam == IDM_SAVEAS)?(DWORD)HELPID_FILE_SAVE_AS:
        (wParam == IDM_PRINT)?(DWORD)HELPID_FILE_PRINT:
        (wParam == IDM_EXIT)? (DWORD)HELPID_FILE_EXIT:
        (wParam == IDM_UNDO)? (DWORD)HELPID_EDIT_UNDO:
        (wParam == IDM_CUT)?(DWORD)HELPID_EDIT_CUT:
        (wParam == IDM_CLEAR)?(DWORD)HELPID_EDIT_CLEAR:
        (wParam == IDM_COPY)? (DWORD)HELPID_EDIT_COPY:
        (wParam == IDM_PASTE)?(DWORD)HELPID_EDIT_PASTE:
        (DWORD)0L;

      if (!dwHelpContextId) {
         Messagebox( hWnd, "Help not available for Help Menu item",
                     "Help Example", MB_OK);
         return (DefWindowProc(hWnd, message, wParam, lParam));
      }

      bHelp = FALSE;
      WinHelp(hWnd,szHelpFileName,HELP_CONTEXT,dwHelpContextId);
      break;
    }
```

Detecting *F1* in dialog boxes is somewhat more difficult than in menus. You must install a message filter, using the WH_MSGFILTER option of the **SetWindowsHook** function. Your message filter function responds to WM_KEYDOWN and WM_KEYUP messages for VK_F1 when they are sent to a dialog box, as indicated by the MSGF_DIALOGBOX code. By examining the message structure passed to the filter, you can determine the context of the *F1* help—what the dialog box is, and the specific option or item. You should not call **WinHelp** while processing the filtered message, but rather post an application-defined message to your application to call **WinHelp** at the first available opportunity.

Getting help on items on the Help menu

Sometimes users may want information about a general concept in the application rather than about a particular control or window. In these cases, the application should provide Help for a particular topic that is identified by a keyword rather than a context identifier.

For example, if the Help file for your application contains a topic that describes how the keyboard is used, you could place a "Keyboard" item in your Help menu. Then, when the user selects that item, your application calls **WinHelp** and requests that topic:

```
case IDM_HELP_KEYBOARD:
   WinHelp (hWnd, lpHelpFile, HELP_KEY, (LPSTR) "Keyboard");
   break;
```

Accessing additional keyword tables

Your application may have commands or terms that correspond to terms in a similar, but different, application. Given a keyword, the application can call **WinHelp** and look up topics defined in an alternate keyword table. This multikey functionality is accessed through the **WinHelp** hook with the *wCommand* parameter set to HELP_MULTIKEY.

You specify the footnote character for the alternate keyword table, and the keyword or phrase, via a **MULTIKEYHELP** structure which is passed as the *dwData* parameter in the call to **WinHelp**. This structure is defined in WINDOWS.H as:

```
typedef struct tagMULTIKEYHELP {
  WORD mdSize;
  BYTE mkKeyList;
  BYTE szKeyPhrase[1];
} MULTIKEYHELP;
```

The following table lists the format of the fields of the **MULTIKEY-HELP** structure:

Table A.14 MULTIKEYHELP structure formats

Parameter	Format
mkSize	The size of the structure, including the keyword (or phrase) and the associated key-table letter.
mkKeyList	A single character which defines the footnote character for the alternate keyword table to be searched.
szKeyPhrase	A null-terminated keyword or phrase to be looked up in the alternate keyword table.

The following example illustrates a keyword search for the word "frame" in the alternate keyword table designated with the footnote character "L":

```
MULTIKEYHELP mk;
char szKeyword[]="frame";
mk.mkSize = sizeof(MULTIKEYHELP) + (WORD)lstrlen(szKeyword);
mk.mkKeylist = 'L';
mk.szKeyphrase = szKeyword;
WinHelp(hWnd, lpHelpfile, HELP_MULTIKEY, (LPSTR)&mk);
```

Canceling Help

The Windows Help application is a shared resource that is available to all Windows applications. In addition, since it is a stand alone application, the user can execute it like any other application. As a result, your application has limited control over the Help application. While your application cannot directly close the Help application window, your application can inform the Help application that Help is no longer needed. Before closing its main window, your application should call **WinHelp** with the *wCommand* parameter set to HELP_QUIT, as shown in the following example, to inform the Help application that your application will not need it again.

```
case WM_DESTROY:
  WinHelp (hWnd, lpHelpFile, HELP_QUIT, NULL);
```

An application that has called **WinHelp** at some point during its execution must call **WinHelp** with the *wCommand* parameter set to HELP_QUIT before the application exits from WinMain (typically during the WM_DESTROY message processing).

If an application opens more than one Help file, then it must call **WinHelp** to quit help for each file.

If an application or DLL has opened a Help file but no longer wants the associated instance of the Help engine (WINHELP.EXE) to remain active, then the application or DLL should call **WinHelp** with the *wCommand* parameter set to HELP_QUIT to destroy the instance of the Help engine.

Under no circumstances should an application or DLL terminate without calling **WinHelp** for any of the opened Help files. A Help file is opened if any other **WinHelp** call has been previously made using the Help filename.

The Windows Help application does not exit until all windows that have called **WinHelp** have called it with *wCommand* set to HELP_QUIT. If an application fails to do so, then the Help application will continue running after all applications that requested Help have terminated.

Help examples

This section contains some examples of Help source files and their corresponding topics as displayed in Help. Each example shows a topic (or part of a topic) as it appears to the Help writer in the RTF-capable word processor and as it appears to the user in the Help window. You can use these examples as guides when creating your own topic files. The examples should help you predict how a particular topic file created in a word processor will appear to the user.

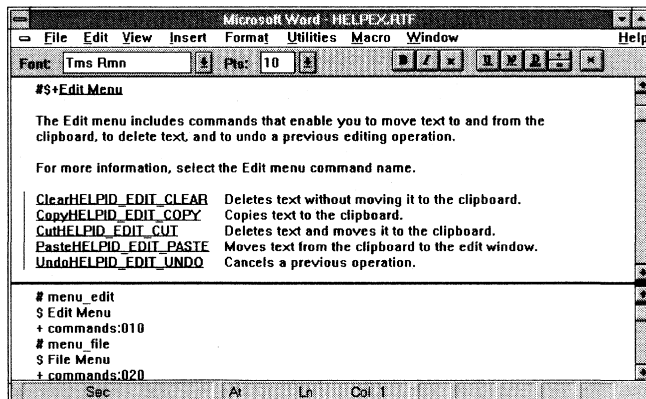


Figure A.9
Word for Windows topic

Figure A.10
Help topic display

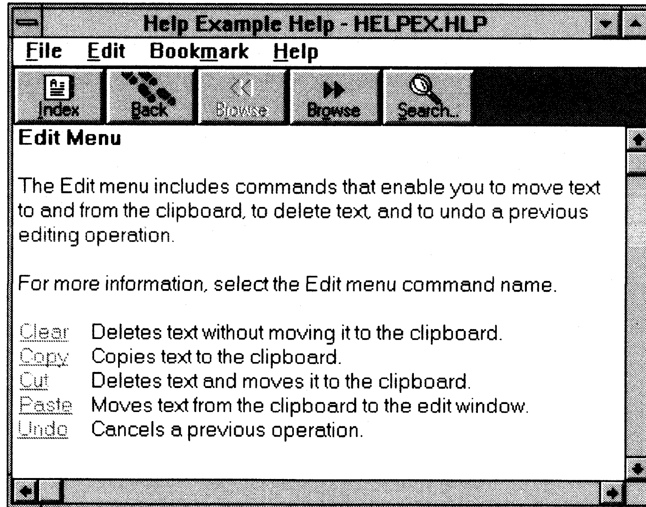


Figure A.11
Bitmap by reference in topic

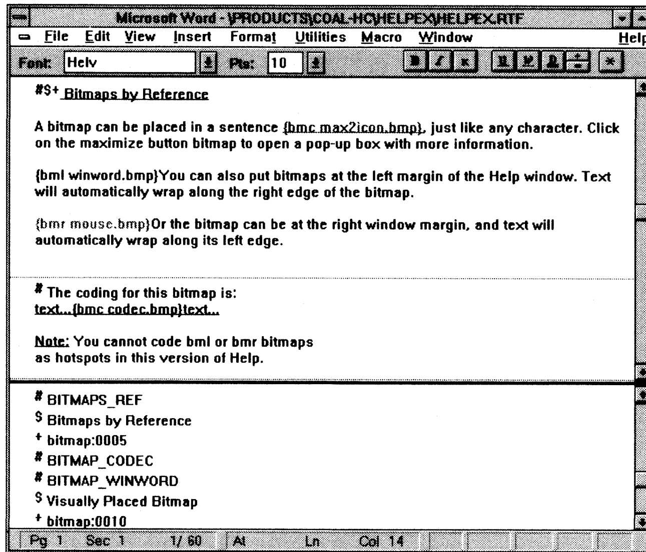
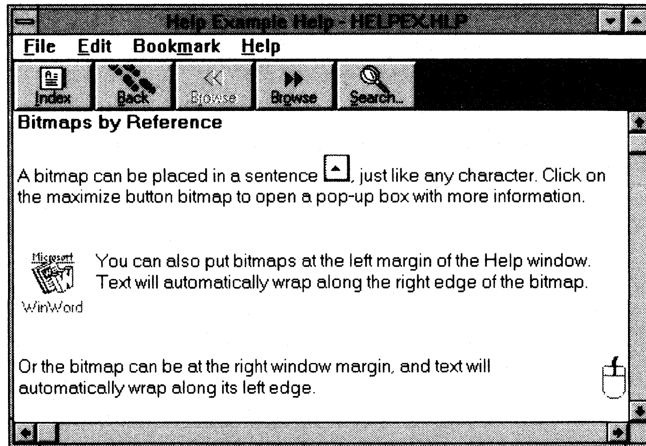


Figure A.12
Help topic display



The Helpex project file

The following is the Helpex (sample Help) project file:

```
[OPTIONS]
ROOT=c:\help
INDEX=main_index
TITLE=Help Example
COMPRESS=true

[FILES]
helpex.rtf  ; jump topics
terms.rtf   ; look-up terms

[MAP]
main_index 0xFFFF
#define HELPID_EDIT_CLEAR     100
#define HELPID_EDIT_COPY      101
#define HELPID_EDIT_CUT       102
#define HELPID_EDIT_PASTE     103
#define HELPID_EDIT_UNDO      104
#define HELPID_FILE_EXIT      200
#define HELPID_FILE_NEW       201
#define HELPID_FILE_OPEN      202
#define HELPID_FILE_PRINT     203
#define HELPID_FILE_SAVE      204
#define HELPID_FILE_SAVE_AS   205
#define HELPID_EDIT_WINDOW    300
#define HELPID_MAXIMIZE_ICON  301
#define HELPID_MINIMIZE_ICON  302
#define HELPID_SPLIT_BAR      303
#define HELPID_SIZE_BOX       304
```

```
#define HELPID_SYSTEM_MENU    305
#define HELPID_TITLE_BAR      306
#define HELPID_SIZING_BORDER  307
```

I N D E X

- ;
  - null statement *22, 99*
  - statement terminator *22, 99*
- /* */ (comments) *7*
- /**/ (token pasting) *7*
- // (comments) *8*
- – – operator
  - decrement *81, 84*
- ? : operator
  - conditional expression *82, 94*
- :: (scope resolution operator) *82, 108*
- .* and –>* operators (dereference pointers) *82, 97*
- \\ escape sequence (display backslash character) *15*
- \" escape sequence (display double quote) *15*
- \? escape sequence (display question mark) *15*
- \' escape sequence (display single quote) *15*
- : (labeled statement) *23*
- != operator
  - not equal to *82, 92*
- && operator
  - logical AND *81, 93*
- ++ operator
  - increment *81, 84*
- << operator
  - put to *See* overloaded operators, >> (put to)
  - shift bits left *81, 89*
- <= operator
  - less than or equal to *82, 91*
- == operator
  - equal to *91*
- >= operator
  - greater than or equal to *82, 91*
- >> operator
  - get from *See* overloaded operators, << (get from)
  - shift bits right *81, 89*
- | | operator
  - logical OR *81, 94*
- –> operator (selection) *82*
  - overloading *140*
  - structure member access *67, 83*
  - union member access *83*
- * (pointer declarator) *23*
- \ (string continuation character) *19*
- ## symbol
  - overloading and *135*
  - preprocessor directives *80*
  - token pasting *7, 159*
- ! operator
  - logical negation *81, 86*
- % operator
  - modulus *81, 88*
  - remainder *81, 88*
- & operator
  - address *81, 85*
  - bitwise AND *81, 92*
    - truth table *93*
  - position in reference declarations *39, 106*
- * operator
  - indirection *81, 86*
    - pointers and *58*
  - multiplication *81, 88*
- + operator
  - addition *81, 88*
  - unary plus *81, 86*
- , operator
  - evaluation *82, 96*
  - function argument lists and *22*
- – operator
  - subtraction *81, 89*
  - unary minus *81, 86*
- / operator
  - division *81, 88*
    - rounding *88*

- < operator
  - less than 82, 90
- = operator
  - assignment 81, 95
    - compound 96
    - overloading 139
  - equal to 82
  - initializer 23
- > operator
  - greater than 82, 91
- ^ operator
  - bitwise XOR 81, 93
    - truth table 93
- | operator
  - bitwise inclusive OR 81, 93
    - truth table 93
- ~ operator
  - bitwise complement 81, 86
- . operator (selection) 82
  - structure member access 67, 83
- 1's complement (~) 81, 86
- # symbol
  - conditional compilation and 162
  - converting strings and 159
  - null directive 154
  - overloading and 135
  - preprocessor directives 24, 80, 153
- 80x87 coprocessors *See* numeric coprocessors

A

- \a escape sequence (audible bell) 15
- abort (function)
  - destructors and 133
- abstract classes 172, *See* classes, abstract
- access
  - classes 120-122
    - base 120
    - default 120
    - derived 120
    - qualified names and 121
  - data members and member functions 118
  - friend classes 121
  - friend functions 118
  - overriding 119
  - structure members 67, 83, 119
  - unions
    - members 83, 119
- accounting applications 203
- addition operator (+) 81, 88
- address operator (&) 81, 85
- addresses, memory *See* memory addresses
- adjustfield, ios data member 186
- aggregate data types *See* data types
- alert (\a) 15
- algorithms
  - #include directive 161
- Alias section 345
- aliases *See* referencing and dereferencing
- alignment
  - word 69
- allocate, streambuf member function 194
- allocation, memory *See* memory, allocation
- ancestors *See* classes, base
- AND operator (&) 81, 92
  - truth table 93
- AND operator (&&) 81, 93
- angle brackets 160
- anonymous unions
  - member functions and 72
- ANSI *See* online ANSI.DOC
  - C standard
    - Turbo C++ and 3
  - keywords 9
    - predefined macro 170
- argsused pragma 166
- arguments *See also* parameters
  - actual
    - calling sequence 64
  - conversions 64
  - converting to strings 159
  - default
    - constructors and 124, 126
  - to #define directive 157
  - function calls and 64
  - functions taking none 62
  - matching number of 65
  - parameters vs. 3
  - passing
    - C-language style 48
  - type checking 61
  - variable number of 23
    - Pascal and 50
- arithmetic, pointers *See* pointers, arithmetic
- arithmetic types 40

arrays *59*
  of classes
    initializing *110*
  constructors for
    order of calling *129*
  delete operator and *109*
  elements
    comparing *90*
  indeterminate *59*
    structures and *60*
  initialization *43, 44*
  multidimensional *59*
  new operator and *109*
  sizeof and *87*
  subscripts *21, 82*
    overloading *140*
asm (keyword) *208*
  braces and *208*
  how to use *98*
.ASM files *See* assembly language
assembler
  built in *207*
assembly language *See also* opcodes
  built-in assembler *See* built-in assembler
  huge functions and *52, 168*
  inline *207*
    braces and *208*
    C structure members and *212*
      restrictions *213*
    calling functions *211*
    commenting *208*
    directives *211*
    floating point in *201*
    goto in *213*
    jump instructions *211, 213*
    referencing data in *211*
    register variables in *212*
    semicolons and *209*
    size overrides in *212*
    syntax *208*
    variable offsets in *212*
  statement syntax *98*
assignment operator
  overloading *139*
assignment operator (=) *81, 95*
  compound *96*
associativity *78, See also* precedence
  expressions *76*
asterisk (*) *23*
atexit (function)
  destructors and *133*
attach
  filebuf member function *182*
  fstreambase member function *184*
auto (keyword) *45*
  class members and *113*
  external declarations and *36*
  register keyword and *30*
automatic objects *30, See* objects, automatic

B

\b escape sequence (backspace) *15*
backslash character
  hexadecimal and octal numbers and *14*
  line continuation *159*
backslash character (\\) *15*
backspace character (\b) *15*
bad, ios member function *186*
banker's rounding *205*
base, streambuf member function *194*
base classes *See* classes
basefield, ios data member *186*
BASM (built-in assembler) *See* built-in assembler
BCD *203*
  converting *204*
  number of decimal digits *204*
  range *204*
  rounding errors and *204*
bell (\a) *15*
binary coded decimal *See* BCD
binary operators *See* operators
bit fields
  hardware registers and *70*
  integer *71*
  portable code and *71*
  structures and *70*
  unions and *73*
bitalloc, ios member function *186*
Bitmaps section *348*
bits
  shifting *81, 89*
bitwise
  AND operator (&) *81, 92*

truth table *93*
complement operator (~) *81, 86*
OR operator (|) *81, 93*
truth table *93*
XOR operator (^) *81, 93*
truth table *93*
blen, streambuf member function *194*
block
scope *28*
statements *98*
Boolean data type *99*
bp, ios data member *186*
braces *21*
asm keyword and *208*
brackets *21, 82*
overloading *140*
break statements *103*
loops and *103*
buffers
C++ streams and *182, 183*
BUILD option *340*
BuildTags section *339*
built-in assembler *207*

C

C++ *105-145*
C code and *168*
classes *See* classes
comments *8*
complex numbers *See* complex numbers
constants *See* constants
constructors *See also* constructors
filebuf *182*
fstream *183*
fstreambase *184*
ifstream *179, 185*
ios *186*
iostream *188*
iostream_withassign *188*
istream *189*
istream_withassign *190*
istrstream *191*
ofstream *179, 191, 192*
ofstream_withassign *192*
ostrstream *193*
streambuf *193*
strstream *198*
strstreambase *196*
strstreambuf *196, 197*
conversions *See* conversions, C++
data members *See* data members
declarations *See* declarations
destructors *See* destructors
DLLs and *233*
enumerations *See* enumerations
file operations *See* files
fill characters *176*
floating-point precision *176*
for loops *See* loops, for, C++
formatting *See* formatting, C++
Fourier transforms example *202*
functions
C functions and *32*
friend *113*
access *118*
inline *See* functions, inline
name mangling and *32*
pointers to *55*
taking no arguments *62*
virtual *140*
pure keyword and *142*
inheritance *See* inheritance
initializers *45*
keywords *9*
member functions *See* member functions
members *See* data members; member functions
name spaces *69*
operators *See* operators, C++; overloaded operators
output *See* output, C++
parameters *See* parameters
referencing and dereferencing *See* referencing and dereferencing
scope *See* scope
streams *See* streams, C++
structures *See* structures
templates *See* templates
this
nonstatic member functions and *113*
static member functions and *115*
unions *See* unions
visibility *See* visibility

C language
argument passing *48*
C++ code and *168*
calling conventions *168, 169*
calling conventions *See also* parameters, passing; Pascal
calls
far, functions using *52*
near, functions using *52*
carriage return character *15*
case
preserving *50*
sensitivity
forcing *48*
global variables and *48*
identifiers and *10*
pascal identifiers and *11*
statements *See* switch statements
cast expressions
syntax *85*
__CDECL__ macro *168*
cdecl (keyword) *48, 50*
function modifiers and *52*
characters
constants *See* constants, character
data type char *See* data types, char
fill
setting *176*
newline
inserting *176*
unsigned char data type
range *19*
whitespace
extracting *176*
class template *See* templates
classes *111-124, See also* C++; individual class names; inheritance
abstract *142, 172*
access *120-122*
default *120*
qualified names and *121*
arrays of
initialization *110*
auto keyword and *113*
base
calling constructor from derived class *130*
constructors *131*
pointers to
destructors and *134*
private
friend keyword and *121*
protected keyword and *120*
unions and *120*
virtual *122*
constructors and *128*
class names and *112*
data types and *38*
declarations
incomplete *112*
derived
base class access and *120*
calling base class constructor from *130*
constructors *131*
DLLs and *233*
extern keyword and *113*
friends *122-124*
access *121*
hierarchies
ios family *173*
streambuf *172*
initialization *See* initialization, classes
member functions *See* member functions
members, defined *113*
naming *See* identifiers
objects *111, 113*
initialization *See* initialization, classes, objects
register keyword and *113*
scope *See* scope, classes
sizeof operator and *87*
streams and *171*
files *171*
formatted I/O *172*
memory buffers *171, 172*
strings *171*
syntax *111*
unions and *73*
_clear87 (function)
floating point exceptions and *201*
clear, ios member function *186*
close
filebuf member function *182*
fstreambase member function *184*
Code Generation dialog box *69*

code models *See* memory models
code segment
  storing virtual tables in *224*
colons *23*
comma
  operator *82, 96*
  separator *22*
command-line compiler
  options
    .DLLs with all exportables *224*
    floating point
      emulation (–f) *200*
    .OBJs with explicit exports (–WN) *222*
    smart callbacks (–WS) *223*
commas
  nested
    macros and *158*
comments *7*
  // *8*
  /* */ *7*
  as whitespace *6*
  inline assembly language code *208*
  nested *7*
  token pasting and *7*
  whitespace and *8*
__COMPACT__ macro *168*
compact memory model *See* memory models, compact
compilation
  speeding up *167*
compiler
  diagnostic messages *236-307*
compiling
  conditional
    # symbol and *162*
complement
  bitwise *81, 86*
complex declarations *See* declarations
complex.h (header file)
  complex numbers and *202*
complex numbers
  << and >> operators and *202*
  C++ operator overloading and *202*
  example *202*
  header file *202*
  using *202*
component selection *See* operators, selection (. and –>)
compound assignment operators *96*
COMPRESS option *344*
concatenating strings *See* strings, concatenating
conditional compilation
  # symbol and *162*
  __cplusplus macro and *168*
conditional operator (? :) *94*
conforming extensions *3*
const (keyword) *47*
  C++ and *47*
  formal parameters and *64*
  pointers and *47, 57*
constant expressions *20*
constants *11, 47, See also* numbers
  Turbo C++ *15*
  C++ *47*
  case statement
    duplicate *100*
  character *12, 14*
    extending *15*
    integer and *42*
    two-character *15*
    wide *16*
  data types *13*
  decimal *11, 12*
    data types *13*
    suffixes *13*
  enumerations *See* enumerations
  expressions *See* constant expressions
  floating point *12, 16*
    data types *16*
    negative *16*
    ranges *17*
  fractional *12*
  hexadecimal *12, 13*
  integer *11, 12*
  internal representations of *19*
  manifest *168*
  octal *12*
  pointers and *57*
  string *See* strings, literal
  suffixes and *13*
  syntax *12*
  ULONG_MAX and UINT_MAX *89*

constructors 124-129
arrays
order of calling 129
base class
calling
from derived class 130
order 131
calling 125
class initialization and 129
classes
virtual base 128
copy 127
class object initialization and 129
default arguments and 124, 126
default parameters 126
defaults 126
delete operator and 125
derived class
order of calling 131
inheritance and 124
invoking 125
new operator and 125
non-inline
placement of 132
order of calling 128
overloaded 127
unions and 125
virtual 124
consumer (streams) 171
continue statements 103
loops and 103
continuing lines 6, 19, 159
_control87 (function)
floating point exceptions and 201
control lines *See* directives
conversions 41
argument *See* arguments, conversions
arguments to strings 159
arrays 60
BCD 204
C++ 176
setting base for 176
character
integers and 42
decimal 176
hexadecimal 176
integers
character and 42
octal 176
pointers 59
sign extension and 42
special 42
standard 42
coprocessors *See* numeric coprocessors
copy constructors *See* constructors, copy
_ _cplusplus macro 168
_cs (keyword) 48

D

\D escape sequence (display a string of octal digits) 15
data
static, DLLs and 233
data members *See also* member functions
access 118
dereference pointers 82, 97
private 118
protected 118
public 118
scope 116-119
static 115
declaration 116
definition 116
uses 116
data models *See* memory models
data segment
removing virtual tables from 224
data structures *See* structures
data type
template argument 147
data types 25, *See also* constants; floating point; integers; numbers
aggregate 38
arithmetic 40
BCD *See* BCD
Boolean 99
C++ streams and 174, 178
char 40
range 19
signed and unsigned 15, 40
classes and 38
conversions *See* conversions
declarations 39

declaring *38*
default *38*
derived *38*
enumerations *See* enumerations
range *19*
function return types *61*
fundamental *38, 39*
creating *40*
identifiers and *26, 27*
integers *See* integers
integral *40*
internal representations *40*
memory use *87*
new, defining *46*
parameterized *See* templates
ranges *19*
scalar *38*
initializing *43*
size_t *87, 137, 138*
table of *19*
taxonomy *38*
types of *38*
unsigned char
range *19*
void *39*
wchar_t *16*
date *See also* time
macro *169*
__DATE__ macro *169*
#define and #undef directives and *157*
deallocation, memory *See* memory, allocation
dec (manipulator) *176*
decimal constants *See* constants, decimal
declarations *25*
arrays *59*
C++ *38*
incomplete *112*
complex *52*
examples *53, 54*
data types *38*
default *38*
defining *26, 31, 33, 44*
extern keyword and *45*
examples *39*
external *31, 36*
storage class specifiers and *36*
function *See* functions, declaring
incomplete class *112*
with initializers
bypassing *103*
mixed languages *50*
modifiers and *47*
objects *34*
Pascal *50*
point of *143*
pointers *56*
referencing *26, 33*
extern keyword and *45*
simple *44*
static data members *116*
structures *See* structures, declaring
syntax *33, 34*
tentative definitions and *33*
unions *73*
declarators
pointers (*) *23*
syntax *54*
decrement operator (– –) *81, 84*
default (label)
switch statements and *100*
default constructors *See* constructors, default
#define directive *155*
argument lists *157*
global identifiers and *157*
keywords and *157*
redefining macros with *156*
with no parameters *155*
with parameters *157*
defined operator *163*
defining declarations *See* declarations, defining
definitions *See* declarations, defining
function *See* functions, definitions
tentative *33*
delete (operator) *108*
arrays and *109*
constructors and destructors and *125*
destructors and *132, 133*
dynamic duration objects and *30*
overloading *137*
pointers and *132*
dereferencing *See* referencing and dereferencing
derived classes *See* classes

derived data types *See* data types
descendants *See* classes, derived
destructors *124, 132-135, See also* initialization
abort function and *133*
atexit function and *133*
base class pointers and *134*
calling *125*
class initialization and *129*
delete operator and *125, 132, 133*
exit function and *133*
global variables and *133*
inheritance and *124*
invoking *125, 132*
explicitly *133*
new operator and *125, 133*
pointers and *132*
#pragma exit and *133*
unions and *125*
virtual *124, 134*
diagnostic messages
compiler *236-307*
digits
hexadecimal *12*
nonzero *12*
octal *12*
direct member selector *See* operators, selection (. and ±>)
directives *162, 153-170, See also* individual directive names; macros
symbol
overloading and *135*
symbol *24*
overloading and *135*
conditional *162*
nesting *162*
conditional compilation and *162*
error messages *165*
keywords and *157*
line control *164*
pragmas *See* pragmas
sizeof and *87*
syntax *154*
usefulness of *153*
division operator (/) *81, 88*
rounding *88*
_ _DLL_ _ macro *169*
DLLs
building *217-234*
C++ and
classes *233*
mangled names *234*
compiler options and *233*
compiling and linking *229*
creating *224, 230*
defined *228*
exit point *231*
import libraries and *229*
initialization functions *230*
LibMain function and *230*
macro *169*
memory models and *226*
pointers and *232*
smart callbacks and *223*
static data *233*
virtual tables and *233*
WEP function *231*
do while loops *See* loops, do while
doallocate, strstreambuf member function *197*
dot operator (selection) *See* operators, selection (. and ±>)
double quote character
displaying *15*
_ds (keyword) *48*
duplicate case constants *100*
duration *29*
dynamic
memory allocation and *30*
local
scope and *30*
pointers *56*
static *29*
dynamic duration
memory allocation and *30*
dynamic memory allocation *See* memory, allocation

E

eback, streambuf member function *194*
ebuf, streambuf member function *194*
egptr, streambuf member function *194*
elaborated type specifier *112*
elements
parsing *6*

#elif directive *162*
ellipsis (...) *23*
  prototypes and *62, 65*
#else directive *162*
empty statements *99*
empty strings *18*
emulating the 80x87 math coprocessor *See* floating point, emulating
enclosing block *28*
#endif directive *162*
endl (manipulator) *176*
ends (manipulator) *176*
enum (keyword) *See* enumerations
enumerations *73*
  C++ *74*
  class names and *112*
  constants *12, 17, 74*
    default values *17*
  conversions *42*
  default type *73*
  name space *28*
  range *19*
  scope, C++ *75*
  structures and
    name space in C++ *69*
  tags *74*
    name spaces *75*
eof, ios member function *186*
epptr, streambuf member function *194*
equal to operator (=) *82*
equal-to operator (==) *91*
equality operators *See* operators, equality
#error directive *165*
errors
  compiler *236-307*
  defined *236*
  disk access *236*
  expressions *79*
  fatal *236*
  floating point
    disabling *201*
  Help compiler *237*
  Librarian *239*
  math, masking *201*
  memory access
    defined *236*
  messages
    list *236-307*
  preprocessor directive for *165*
  run-time *238*
  syntax
    defined *236*
  Linker (list) *239*
_es (keyword) *48*
escape sequences *12, 14*
  length *14*
  number of digits in *14*
  octal
    non-octal digits and *15*
  table of *15*
evaluation order *See* precedence
exclusive OR operator (^) *81, 93*
  truth table *93*
exit (functions)
  destructors and *133*
exit pragma *166*
exit procedure, Windows *231*
exponents *12*
_export (keyword) *48, 52*
  Windows applications and *222, 224*
expressions
  associativity *76*
  cast, syntax *85*
  constant *20*
  conversions and *41*
  decrementing *84*
  empty (null statement) *22, 99*
  errors and overflows *79*
  floating point
    precedence *79*
  function
    sizeof and *87*
  grouping *21*
  incrementing *84*
  precedence *76, 78*
  statements *22, 99*
  syntax *77*
  table *77*
extensions *9*
extent *See* duration
extern (keyword) *45*
  arrays and *59*
  class members and *113*

const keyword and *47*
linkage and *31*
name mangling and *32*
external
declarations *31*
linkage *See* linkage
extraction operator (<<) *See* overloaded operators, << (get from)
extractors *See* input, C++

F

–f command-line compiler option (emulate floating point) *200*
\f escape sequence (formfeed) *15*
fail, ios member function *186*
far
functions *See* functions, far
pointers *See* pointers, far
far (keyword) *48*
fatal errors
Compile-time *236*
fd, filebuf member function `int fd()` *182*
field width *See* formatting, width (C++)
__FILE__ macro *169*
#define and #undef directives and *157*
file descriptor *182*
file scope *See* scope
filebuf (class) *182*
files *See also* individual file-name extensions
.ASM *See* assembly language
buffers
C++ *182, 183*
current
macro *169*
header *See* header files
include *See* include files
including *160*
opening
default mode *180*
scope *See* scope
streams
C++ operations *184*
Files section
Help project file *338*
fill, ios member function *186*
fill characters
C++ *176, 177*
financial applications *203*
flags
format state *See* formatting, C++, format state flags
ios (class)
setting *176*
ios member function *187*
floatfield, ios data member *186*
floating point *See also* data types; integers; numbers
arithmetic
interrupt functions and *214*
constants *See* constants
conversions *See* conversions
double
range *19*
emulating *200*
exceptions
disabling *201*
expressions
precedence *79*
fast *200*
libraries *199*
long double
range *19*
precision
setting *176*
ranges *19*
registers and *201*
using *199*
flow-control statements *See* if statements; switch statements
flush
ostream member function *192*
flush (manipulator) *176*
for loops *See* loops, for
FORCEFONT option *342*
formal parameters *See* parameters, formal
format state flags *See* formatting, C++, format state flags
formatting
C++
classes for *172*
fill character *176, 177*
format state flags *174*
I/O *176, See also* manipulators
output *174*

padding *177*
width functions *See also* manipulators
setting *176*
streams and
clearing *176*
formatting flags *186*
formfeed character *15*
forward references *26*
Fourier transforms
complex number example *202*
free (function)
delete operator and *108*
dynamic duration objects and *30*
freeze, strstreambuf member function *197*
friend (keyword) *113, 122-124*
base class access and *121*
functions and *See* C++, functions, friend
fstream (class) *183*
fstreambase (class) *184*
function call operator *See* parentheses
function operators *See* overloaded operators
function template *148, See* templates
functions *60-65*
arguments
no *62*
calling *64, See also* parentheses
in inline assembly code *211*
operators ( ) *83*
overloading operator for *140*
rules *64*
cdecl and *51*
class names and *112*
comparing *92*
declaring *60, 61*
default types for memory models *52*
definitions *60, 63*
duration *30*
exit *166*
export
Windows applications and *222*
exporting *224*
external *45*
declarations *31*
far *52*
friend *See* C++, functions, friend
graphics *See also* graphics
huge *52*
assembly language and *52*
_loadds and *52*
saving registers *168*
inline
assembly language *See* assembly language, inline
C++ *114*
linkage *115*
internal linkage *46*
interrupt *See* interrupts, functions
linking C and C++ *32*
main *60*
member *See* member functions
memory
models and *48*
name mangling and *32*
near *52*
no arguments *39*
not returning values *39*
operators *See* overloaded operators
overloaded *See* overloaded functions
Pascal
calling *50*
pointers *55*
object pointers vs. *54*
pointers to
void *55*
prototypes *See* prototypes
return statements and *104*
return types *61*
scope *See* scope
sizeof and *87*
startup *166*
static *31*
stdarg.h header file and *62*
storage class specifiers and *32*
structures and *67*
type
modifying *52*
Windows *224*
fundamental data types *See* data types

G

gbump, streambuf member function *194*
gcount, istream member function *189*
generic pointers *39, 56*
generics *See* 3

get
  istream member function *189*
get from operator (>>) *See* overloaded
  operators, >> (get from)
getline, istream member function *190*
global identifiers *See* identifiers, global
global variables *28*, *See also* variables
  case sensitivity and *48*
  destructors and *133*
  underscores and *48*
good, ios member function *187*
goto statements *103*
  assembly language and *213*
  labels
    name space *28*
gptr, streambuf member function *194*
grammar
  tokens *See* tokens
greater-than operator (>) *82, 91*
greater-than or equal-to operator (>=) *82, 91*

H

hdrfile
  pragma *167*
hdrstop
  pragma *167*
header files *See also* include files
  complex numbers *202*
  extern keyword and *33*
  function prototypes and *62*
  #include directive and *160*
  name mangling and *33*
  precompiled *167*
  prototypes and *60*
  variable parameters *62*
heap *30*
hex (manipulator) *176*
hexadecimal
  constants *See* constants, hexadecimal
  digit *12*
hidden objects *29*
hiding *See* scope, C++
hierarchy *See* classes, hierarchies
horizontal tab *15*
huge
  functions
    saving registers and *168*
  pointers *See* pointers, huge
huge (keyword) *48*
  assembly language and *52*

I

IDE *See* integrated development environment
identifiers *10*
  Turbo C++ keywords as *3*
  case *50*
    sensitivity and *10*
  classes *111*
  data types and *26, 27*
  declarations and *26*
  declaring *44*
  defined operator and *163*
  defining *157*
  duplicate *29*
  duration *29*
  enumeration constants *17*
  external
    name mangling and *32*
  global *168*
    #define and #undef directives and *157*
  linkage *31*
    attributes *31*
  mixed languages *50*
  name spaces *See* name spaces
  no linkage attributes *32*
  Pascal *50*
  pascal (keyword)
    case sensitivity and *11*
  rules for creating *10*
  scope *See* scope
  storage class and *27*
  testing for definition *163*
  unique *31*
IEEE
  floating-point formats *41*
  rounding *205*
#if directive *162*
if statements *99*
  nested *99*
#ifdef directive *163*
#ifndef directive *163*
ifstream (class) *185*
  constructor *179*
  insertion operations *179*

ignore, istream member function *190*
import libraries
  DLLs and *229*
  module definition files and *229*
in_avail, streambuf member function *194*
include files *See also* header files
  #include directive and *160*
  search algorithm for *161*
#include directive *160*
  search algorithm *161*
inclusive OR operator (|) *81, 93*
  truth table *93*
incomplete declarations
  classes *112*
  structures *70*
increment operator (++) *81, 84*
indeterminate arrays *59*
  structures and *60*
INDEX option *342*
indirect member selector *See* operators, selection
indirection operator (*) *81, 86*
  pointers and *58*
inequality operator (!=) *82, 92*
inheritance *See also* classes
  constructors and destructors *124*
  multiple
    base classes and *122*
  overloaded assignment operator and *139*
  overloaded operators and *136*
init, ios member function *187*
initialization *42, See also* constructors; destructors
  arrays *43*
  classes *129*
    objects *129*
      copy constructor and *129*
  operator *23*
  pointers *56*
  static member definitions and *116*
  structures *43*
  unions *43, 73*
  variables *44*
initializers
  automatic objects *45*
  C++ *45*
  new operator and *110*
inline
  assembly language code *See* assembly language, inline
  expansion *114*
  functions *See* functions, inline
  keyword *114*
input
  C++
    user-defined types *178*
inserter types *174*
inserters *See* output, C++
insertion operator *See* overloaded operators, << (put to)
instances *See* classes, objects
INT instruction *214*
integers *40, See also* data types; floating point; numbers
  C++ streams and *174*
  constants *See* constants
  conversions *See* conversions
  expressions
    precedence *79*
  long *40*
    range *19*
  memory use *40*
  range *19*
  short *40*
  sizes *40*
  suffix *12*
  unsigned
    range *19*
integral data types *See* characters; integers
integrated development environment
  DLLs and *229*
  module definition files and *228*
  nested comments command *8*
  Windows and *221*
    linking *228*
internal linkage *See* linkage
internal representations of data types *40*
interrupt (keyword) *48, 49, 214*
interrupts
  beep
    example *215*
  functions
    example of *215*
    floating-point arithmetic in *214*

memory models and *49*
void *49*
handlers *48*
calling *216*
installing *216*
programming *214*
registers and *49*
I/O
C++
formatting *176*
precision *176*
iomanip.h (header file)
manipulators in *175*
ios (class) *172, 185*
flags
format state *174*
setting *176*
ios data members *186*
iosteram.h (header file)
manipulators in *176*
iostream (class) *188*
iostream library *172*
iostream_withassign (class) *188*
is_open, filebuf member function *182*
istream (class) *189*
derived classes of *179*
istream_withassign (class) *190*
istrstream (class) *191*
iteration statements *See* loops

J

jump instructions, inline assembly language
table *211*
using *213*
jump statements *See* break statements; continue statements; goto statements; return statements

K

keywords *9*, *See also* individual keyword names
ANSI
predefined macro *170*
Turbo C++
using as identifiers *3*
C++ *9*
combining *40*
macros and *157*

L

labeled statements *98*
labels
creating *23*
default *100*
function scope and *28*
goto statement and *103*
in inline assembly code *213*
language extensions
conforming *3*
__LARGE__ macro *168*
large code
data
and memory models *See* memory models
less-than operator (<) *82, 90*
less-than or equal-to operator (<=) *82, 91*
lexical elements or grammar *See* elements
LibMain (function) *230*
return values *231*
libraries
C
linking to C++ code *32*
container class *See* container class library
floating point
using *199*
iostream *172*
prototypes and *65*
stream class *171*
__LINE__ macro *169*
#define and #undef directives and *157*
#line directive *164*
lines
continuing *6, 19, 159*
numbers *164*
macro *169*
linkage *31*
C and C++ programs *32*
external *31*
C++ constants and *47*
name mangling and *32*
internal *31*
no *31, 32*
rules *31*
static member functions *115*
storage class specifiers and *31*

Linker
  dialog box
    Windows and *228*
literal strings *See* strings, literal
_loadds (keyword) *48*
  huge functions and *52*
  uses for *52*
local duration *30*
logical AND operator (&&) *81, 93*
logical negation operator (!) *81, 86*
logical OR operator (| |) *81, 94*
long integers *See* integers, long
loops *101*
  break statement and *103*
  continue statement and *103*
  do while *101*
  for *102*
    C++ *102*
  while *101*
    string scanning and *101*
lvalues *26*, *See also* rvalues
  examples *53*
  modifiable *26*

M

macros *See also* directives
  argument lists *157*
  calling *157*
  commas and
    nested *158*
  defining *155*
    conflicts *156*
    global identifiers and *157*
  expansion *155*
  keywords and *157*
  parameters and *157*
    none *155*
  parentheses and
    nested *158*
  precedence in
    controlling *21*
  predefined *168*, *See also* individual macro names
    ANSI keywords *170*
    C and C++ compilation *170*
    C calling conventions *168*
    conditional compilation *168*
    current file *169*
    current line number *169*
    date *169*
    DLLs *169*
    DOS *169*
    memory models *168*
    Pascal calling conventions *169*
    templates *170*
    time *170*
    Windows applications *170*
  redefining *156*
  side effects and *159*
  undefining *156*
    global identifiers and *157*
main (function) *60*
malloc (function)
  dynamic duration objects and *30*
  new operator and *108*
mangled names *32*
  DLLs and *234*
manifest constants *168*
manipulators *175*, *See also* C++, formatting, width; individual manipulator names
  parameterized *175*
  syntax *176*
Map section *346*
MAPFONTSIZE option *343*
math
  BCD *See* BCD
  coprocessors *See* numeric coprocessors
  errors
    masking *201*
matherr (function)
  proper use of *201*
__MEDIUM__ macro *168*
medium memory model *See* memory models
member functions *113*, *See also* data members
  access *118*
  constructors *See* constructors
  defined *113*
  destructors *See* destructors
  friend *113*
  inline *See* functions, inline, C++
  nonstatic *113*
  private *118*
  protected *118*
  public *118*

scope 116-119
static 115
linkage 115
this keyword and 115
structures and 67
this keyword and 113, 115
unions and 72
members, classes *See* data members; member functions
members, structures *See* structures, members
memory *See also* memory addresses; online WINMEM.DOC
allocation 30
assembly language code and huge functions and 52
new and delete operators and 108
structures 69
data types 87
heap 30
word alignment and
structures 69
memory addresses *See also* memory
constructors and destructors 124
memory models
compact
default function type 52
default
overriding 52
function pointers and 55
functions
default type
overriding 48
interrupt functions and 49
large
default function type 52
macros and 168
medium
default function type 52
pointers
modifiers and 51
predefined macros and 168
small
default function type 52
smart callbacks and 223
Windows applications and 226
memory-resident routines 215
messages
Compile-time 236
Help compiler 237
Librarian 239
run-time 238
Linker (list) 239
methods *See* member functions
Microsoft Windows applications
preprocessor macro 170
Microsoft Windows Help
audience definition 312, 313
cancelling 356, 357
context-sensitive 315-316, 351, 353-354, 355
control codes 322
F1 support 354, 355
file structure 316-318
keywords 326-328
keywords table
accessing 355, 356
on Help menu item 355
planning, overview 312
Microsoft Windows Help compiler 348, 349
Microsoft Windows Help graphics 320
bitmaps, creating
capturing 332, 333
bitmaps, placing 333, 334
Microsoft Windows Help index 314
Microsoft Windows Help Project file
accessing from an application 350
Alias section 345, 346
bitmaps, including by reference 348
Bitmaps section 348
BUILD option 340, 341
BuildTags section 339
compiling 348, 349
COMPRESS option 344, 345
context-sensitive Help 351, 352, 353, 354, 355
context-sensitive topics 346, 347, 348
context strings
alternate 345, 346
creating 337, 338
F1 support 354, 355
Files section 338, 339
FORCEFONT option 343
INDEX option 342

keyword table, accessing *355, 356*
Map section *346, 347, 348*
MAPFONTSIZE option *343, 344*
MULTIKEY option *344*
on Help menu item *355*
Options section *339, 340*
ROOT option *341, 342*
TITLE option *342*
WARNING option *340*
Warning option *340*
Microsoft Windows Help system
appearance to programmer *312*
appearance to user *310, 311*
appearance to writer *311, 312*
calling WinHelp *350, 351*
development cycle described *309, 310*
topics *313*
Microsoft Windows Help text
fonts *319, 320*
layout *318, 319*
Microsoft Windows Help topic files
authoring tool *321*
browse sequence numbers *328, 329, 330*
build tags *323, 324*
context strings *324, 325*
control codes *322*
cross references *330*
definitions *331, 332*
graphics *332*
jumps *330*
keywords *326, 327, 328*
managing *335*
title footnotes *325, 326*
tracking *335, 336*
Microsoft Windows Help topics
content *313*
context-sensitivity *315, 316*
cross-references *330*
definitions *331, 332*
file structure *316, 317, 318*
jumps *330*
structure of *314, 315*
text. See Help text *318*
Microsoft Windows Help Tracker *335, 336*
modifiable lvalues *See* lvalues
modifiable objects *See* objects
modifiers *47*
function type *52*
pointers *51*
table *47*
Modula-2
variant record types *71*
module definition files *219*
IDE options and *228*
import libraries and *229*
LibMain function and *230*
modulus operator (%) *81, 88*
__MSDOS__ macro *169*
multidimensional arrays *See* arrays
MULTIKEY option *344*
multiple inheritance *See* inheritance
multiplication operator (*) *81, 88*

N

\n (newline character) *15*
name mangling *32*
name spaces
scope and *28*
structures *69*
C++ *69*
names *See* identifiers
mangled
DLLs and *234*
qualified *117*
near (keyword) *48*
near functions *See* functions, near
near pointers *See* pointers, near
negation
logical (!) *81, 86*
nested
classes *117*
comments *7, 8*
conditional directives *162*
types *117*
new (operator) *108*
arrays and *109*
constructors and destructors and *125*
destructors and *133*
dynamic duration objects and *30*
handling return errors *109*
initializers and *110*
overloading *110, 137*
prototypes and *109*
_new_handler (for new operator) *109*

newline characters
creating in output *15*
inserting *176*
no linkage *See* linkage
nondefining declarations *See* declarations, referencing
nonzero digit *12*
normalized pointers *See* pointers, normalized
not equal to operator (!=) *82, 92*
not operator (!) *81, 86*
NULL
pointers and *56*
using *56*
null
directive (#) *154*
inserting in string *176*
pointers *56*
statement *22, 99*
strings *18*
number of arguments *23*
numbers *See also* constants; data types; floating point; integers
base
setting for conversion *176*
BCD *See* BCD
converting *See* conversions
decimal
conversions *176*
hexadecimal *12*
backslash and *14*
conversions *176*
displaying *15*
lines *See* lines, numbers
octal *12*
backslash and *14*
conversions *176*
displaying *15*
escape sequence *15*
numeric coprocessors
built in *199*
floating-point emulation *200*
registers and *201*

O

objects *25*, *See also* C++
aliases *105*
automatic *30*
initializers *45*
class names and *112*
duration *29*
hidden *29*
initializers *45*
list of declarable *34*
modifiable *49*
pointers *55*
function pointers vs. *54*
static
initializers *45*
temporary *107*
volatile *49*
oct (manipulator) *176*
octal constants *See* constants, octal
octal digit *12*
ofstream (class) *191*
base class *179*
constructor *179*
insertion operations *179*
opcodes *209*, *See also* assembly language
defined *208*
mnemonics
table *209*
repeat prefixes *211*
open
filebuf member function *182*
fstream member function *184*
fstreambase member function *184*
ifstream member function *185*
ofstream member function *191*
open mode *See* files, opening, C++
operands (assembly language) *208*
operator (keyword)
overloading and *135*
operator functions *See* overloaded operators
operators *79, 79-82*
1's complement (~) *81, 86*
addition (+) *81, 88*
address (&) *81, 85*
AND (&) *81, 92*
truth table *93*
AND (&&) *81, 93*
assignment (=) *81, 95*
compound *96*
overloading *139*
binary *81*

overloading *139*
bitwise
AND (&) *81, 92*
truth table *93*
complement (~) *81, 86*
inclusive OR (|) *81, 93*
truth table *93*
truth table *93*
XOR (^) *81, 93*
truth table *93*
C++ *80*
delete *108*, *See* delete (operator)
dereference pointers *82, 97*
new *See* new (operator)
pointer to member *See* operators, C++, dereference pointers
scope (::) *82, 108*
conditional (? :) *82, 94*
context and meaning *80*
decrement (– –) *81, 84*
defined operator *163*
division (/) *81, 88*
rounding *88*
equality *82, 91*
evaluation (comma) *82, 96*
exclusive OR (^) *81, 93*
truth table *93*
function call ( ) *83*
inclusive OR (|) *81, 93*
truth table *93*
increment (++) *81, 84*
indirection (*) *81, 86*
pointers and *58*
inequality (!=) *82, 92*
list *80*
logical
AND (&&) *81, 93*
negation (!) *81, 86*
OR (| |) *81, 94*
manipulators *See* manipulators
modulus (%) *81, 88*
multiplication (*) *81, 88*
OR (^) *81, 93*
truth table *93*
OR (|) *81, 93*
truth table *93*
OR (| |) *81, 94*
overloading *See* overloaded operators
postfix *82*
prefix *82*
relational *82, 90*
remainder (%) *81, 88*
selection (. and –>) *82, 83*
overloading *140*
structure member access and *67, 83*
shift bits (<< and >>) *81, 89*
sizeof *87*
subtraction (–) *81, 89*
unary
overloading *138*
unary minus (–) *81, 86*
unary plus (+) *81, 86*
Options section *339*
OR operator
bitwise inclusive (|) *81, 93*
truth table *93*
logical (| |) *81, 94*
ostream (class) *192*
derived classes of *179*
flushing *176*
ostream_withassign (class) *192*
ostrstream (class) *193*
out_waiting, streambuf member function *194*
output
C++
user-defined types *178*
overflow
filebuf member function *183*
strstreambuf member function *197*
overflows
expressions and *79*
overloaded constructors *See* constructors, overloaded
overloaded functions
defined *113*
templates and *147*
overloaded operators *78, 80, 135-140*
>> (get from) *177*
complex numbers and *202*
<< (put to) *173*
complex numbers and *202*
assignment (=) *139*
binary *139*
brackets *140*

complex numbers and *202*
creating *114*
defined *113*
delete *137*
functions and *78*
inheritance and *136*
new *110, 137*
operator functions and *135, 136*
operator keyword and *135*
parentheses *140*
precedence and *78*
selection (–>) *140*
unary *138*

P

padding (C++) *177*
paragraphs *See* memory, paragraphs
parameterized
manipulators *See* manipulators
types *See* templates
parameters *See also* arguments
arguments vs. *3*
default
constructors *126*
ellipsis and *23*
empty lists *39*
fixed *62*
formal *64*
C++ *64*
scope *64*
function calls and *64*
passing
C *48, 50*
Pascal *48, 50*
variable *62*
parentheses *21*
as function call operators *83*
macros and *21*
nested
macros and *158*
overloading *140*
parsing *6*
Pascal
functions *50*
identifiers *50*
case sensitivity and *11*
parameter-passing sequence *48*
variant record types *71*
__PASCAL__ macro *169*
pascal (keyword) *48, 50*
function modifiers and *52*
preserving case while using *50*
pass-by-address, pass-by-value, and pass-by-var *See* parameters; referencing and dereferencing
pbase, streambuf member function *194*
pbump, streambuf member function *194*
pcount, ostrstream member function *193*
peek, istream member function *190*
period as an operator *See* operators, selection (. and –>)
phrase structure grammar *See* elements
pointer-to-member operators *See* operators, C++, dereference pointers
pointer types *See* online WINMEM.DOC
pointers *54*, *See also* referencing and dereferencing
advancing *58*
arithmetic *58*
assignments *56*
base class
destructors and *134*
C++ *105*
reference declarations *59*
to class members *82, 97*
comparing *90, 92, 99*
while loops *101*
const *47*
constants and *57*
conversions *See* conversions
declarations *56*
declarator (*) *23, 58*
delete operator and *132*
dereference *82, 97*
DLLs and *232*
far *48*
function *55*
C++ *55*
modifying *52*
object pointers vs. *54*
void *55*
generic *39, 56*
huge *48*
initializing *56*

keywords for *48*
modifiers *51*
near *48*, *See also* segments, pointers
null *56*
NULL and *56*
operator (–>)
overloading *140*
structure and union access *67, 82, 83*
pointers to *55*
range *19*
reassigning *56*
referencing and dereferencing *85*
segment *48*
structure members as *67*
typecasting *59*
virtual table
32-bit *233*
void *56*
portable code
bit fields and *71*
postdecrement operator (– –) *81, 84*
postfix operators *82*
postincrement operator (++) *81, 84*
pptr, streambuf member function *194*
#pragma exit
destructors and *133*
#pragma directives *165*
argsused *166*
exit *166*
hdrfile *167*
hdrstop *167*
saveregs *168*
startup *166*
precedence *78*, *See also* associativity
controlling *21*
expressions *76*
floating point *79*
integer *79*
overloading and
operators *78*
precision, ios member function *187*
precompiled headers
storage file *167*
predecrement operator (– –) *81, 84*
predefined macros *See* macros, predefined
prefix opcodes, repeat *211*
prefix operators *82*
preincrement operator (++) *81, 84*
preprocessor directives *See* directives
private (keyword)
data members and member functions *118*
derived classes and *120*
unions and *73*
procedures *See* functions
producer (streams) *171*
Programmer's Platform *See* Integrated Development Environment
programs
creating *5*
performance
improving *45*
size
reducing *45*
terminate and stay resident
interrupt handlers and *215*
prolog and epilog code
generating *222*
promotions *See* conversions
protected (keyword)
data members and member functions *118*
derived classes and *120*
unions and *73*
prototypes *61-63*
arguments and
matching number of *65*
C++ *60*
ellipsis and *62, 65*
examples *61, 62*
function calls and *64*
function definitions and
not matching *65*
header files and *62*
libraries and *65*
new operator and *109*
scope *See* scope
pseudovariables
register *10*
public (keyword)
data members and member functions *118*
derived classes and *120*
unions and *73*
punctuators *21, 21-24*
pure (keyword)
virtual functions and *142*

pure specifier *37*
put
  ostream member function *192*
put to operator (<<) *See* overloaded operators, >> (put to)
putback, istream member function *190*

Q

qualified names *117*
question mark
  colon conditional operator *82, 94*
  displaying *15*
quotes, displaying *15*

R

\r (carriage return character) *15*
ranges
  floating-point constants *17*
rdbuf
  fstream member function *184*
  fstreambase member function *184*
  ifstream member function *185*
  ios member function *187*
  ofstream member function *191*
  strstreambase member function *196*
rdstate, ios member function *187*
read, istream member function *190*
records *See* structures
reference declarations *59*
  position of & *39, 106*
references
  forward *26*
referencing and dereferencing *85*, *See also* pointers
  asterisk and *23*
  C++ *105*
    functions *106*
    simple *106*
  pointers *82, 97*
referencing data in inline assembly code *211*
referencing declarations *See* declarations
register (keyword) *45*
  class members and *113*
  external declarations and *36*
  formal parameters and *64*
  local duration and *30*
registers
  DI
    assembly language and *212*
  DS
    _loadds and *52*
  hardware
    bit fields and *70*
  interrupts and *49*
  numeric coprocessors and *201*
  pseudovariables *10*
  saving with huge functions *168*
  SI
    assembly language and *212*
  values
    preserving *52*
  variable declarations and *45*
  variables *45*
    in inline assembly code *212*
relational operators *See* operators, relational
remainder operator (%) *81, 88*
repeat prefix opcodes *211*
resetiosflags (manipulator) *175, 176*
resource compiler
  Windows applications and *219*
resource linker
  Windows and *217*
resources
  defined *219*
return
  statements
    functions and *104*
  types *61*
ROOT option *341*
rounding
  banker's *205*
  direction
    division *88*
  errors *203*
routines, assembly language *See* assembly language
rvalues *27*, *See also* lvalues

S

saveregs pragma *168*
_saveregs (keyword) *48, 52*
  uses for *52*
sbumpc, streambuf member function *195*

- scalar data types *See* data types
- scanf (function)
  - >> operator and 177
- scope 27-29, *See also* visibility
  - block 28
  - block statements and 98
  - C++ 29, 143-145
    - hiding 144
    - operator (::) 82, 108
    - rules 144
  - classes 28
    - names 112
  - enclosing 143
  - enumerations 28
    - C++ 75
  - file 28
    - static storage class specifier and 31
  - formal parameters 64
  - function 28
    - prototype 28
  - global 28
  - goto and 28
  - identifiers and 11
  - local
    - duration and 30
  - members 116-119
  - name spaces and 28
  - pointers 56
  - storage class specifiers and 45-47
  - structures 28
  - unions 28
  - variables 28
  - visibility and 29
- searches
  - #include directive algorithm 161
- seekg, istream member function 190
- seekoff
  - filebuf member function 183
  - streambuf member function 195
  - strstreambuf member function 197
- seekp, ostream member function 192
- seekpos, streambuf member function 195
- _seg (keyword) 48
- segments
  - pointers 48
- selection
  - operators *See* operators, selection
  - statements *See* if statements; switch statements
- semicolons 22, 99
- setb, streambuf member function 195
- setbase (manipulator) 175, 176
- setbuf
  - filebuf member function 183
  - fstreambase member function 185
  - streambuf member function 195
  - strstreambuf member function 197
- setf (function) 177
- setf, ios member function 187
- setfill (manipulator) 175, 176
- setg, streambuf member function 195
- setiosflags (manipulator) 175, 176
- set_new_handler (for new operator) 109
- setp, streambuf member function 195
- setprecision (manipulator) 175, 176
- setstate, ios member function 187
- setw (manipulator) 175, 176
- sgetc, streambuf member function 195
- sgetn, streambuf member function 195
- shift bits operators (<< and >>) 81, 89
- short integers *See* integers, short
- side effects
  - macro calls and 159
- sign 12
  - extending 15
    - conversions and 42
- signed (keyword) 40
- single quote character
  - displaying 15
- sink (streams) 171
- size overrides in inline assembly code 212
- size_t (data type) 87, 137, 138
- sizeof (operator) 87
  - arrays and 87
  - classes and 87
  - example 27
  - function-type expressions and 87
  - functions and 87
  - preprocessor directives and 87
  - unions and 72
- __SMALL__ macro 168

small code
  data
    and memory models *See* memory models
smart callbacks
  DLLs and *223*
  memory models and *223*
  Windows applications and *223*
snextc, streambuf member function *195*
software interrupt instruction *214*
sounds
  beep *215*
source (streams) *171*
source code *5*
specifiers *See* type specifiers
splicing lines *6, 19*
sputbackc, streambuf member function *195*
sputc, streambuf member function *195*
sputn, streambuf member function *196*
_ss (keyword) *48*
standard conversions *See* conversions
startup pragma *166*
state, ios data member *186*
statements *97-104,* *See also* individual statement names
  assembly language *98*
  block *98*
    marking start and end *21*
  default *100*
  do while *See* loops, do while
  expression *22, 99*
  for *See* loops, for
  if *See* if statements
  iteration *See* loops
  jump *See* break statements; continue statements; goto statements; return statements
  labeled *98*
  null *99*
  syntax *98*
  while *See* loops, while
static
  data
    DLLs and *233*
  duration *29*
  functions *31*
  members *See* data members, static; member functions, static
  objects *See* objects, static
  variables *See* variables, static
static (keyword) *46*
  linkage and *31*
_status87 (function)
  floating point exceptions and *201*
stdarg.h (header file)
  user-defined functions and *62*
__STDC__ macro *170*
  #define and #undef directives and *157*
storage class
  identifiers and *27*
  specifiers *45*
    functions and *32*
    linkage and *31*
    static
      file scope and *31*
stossc, streambuf member function *196*
str
  ostrstream member function *193*
  strstream member function *198*
  strstreambuf member function *197*
streambuf (class) *172, 193*
  derived classes of *172*
streams
  C++
    classes and *171*
    clearing *176*
    data types *174*
    defined *171*
    errors *179*
    file class *171*
    flushing *176*
    formatted I/O *172*
    manipulators and *See* manipulators
    memory buffer class *171, 172*
    output *173*
    string class *171*
    tied *188*
strings
  concatenating *18*
  continuing across line boundaries *19*
  converting arguments to *159*
  empty *18*
  inserting terminal null into *176*

literal *6, 18*
null *18*
scanning
while loops and *101*
streams
C++ *180*
streams and *171*
stroked fonts *See* fonts
strstrea.h (header file)
string streams and *180*
strstream (class) *198*
strstreambase (class) *196*
strstreambuf (class) *196*
struct (keyword) *66, See also* structures
C++ and *67, 112*
structures *65-71*
access
C++ *120*
bit fields *See* bit fields
C++ *111*
C vs. *112*
complex *202*
declaring *66*
functions and *67*
incomplete declarations of *70*
indeterminate arrays and *60*
initializing *43*
member functions and *67*
members
access *67, 83, 119*
as pointers *67*
C++ *67*
comparing *90*
declaring *66*
in inline assembly code *212*
restrictions *213*
names *69*
memory allocation *69*
name spaces *28, 69*
tags *66*
typedefs and *66*
typedefs and *66*
unions vs. *71*
untagged *66*
typedefs and *66*
within structures *67*
word alignment
memory and *69*
subscripting operator *See* brackets
subscripts for arrays *21, 82*
overloading *140*
subtraction operator (–) *81, 89*
switch statements *100*
case statement and
duplicate case constants *100*
default label and *100*
sync, filebuf member function *183*
sync_with_stdio, ios member function *188*
syntax
assembly language statements *98*
classes *111*
declarations *33, 34*
declarator *54*
directives *154*
expressions *77*
inline assembly language *208*
manipulators *176*
notation *3*
statements *98*
templates *146*

T

\t (horizontal tab character) *15*
tags
enumerations *74*
name spaces *75*
structure *See* structures, tags
taxonomy
types *38*
TCDEF.SYM *167*
__TCPLUSPLUS__ macro *170*
tellg, istream member function *190*
tellp, ostream member function *192*
template function *148*
templates ***145**, See also* syntax
angle brackets *150*
arguments *150*
class *149*
eliminating pointers *152*
function *146*
implicit and explicit *148*
overriding *148*
macro *170*

type-safe
generic lists *151*
__TEMPLATES__ macro *170*
temporary objects *107*
tentative
definitions *33*
terminate and stay resident programs
interrupt handlers and *215*
this (keyword)
nonstatic member functions and *113*
static member functions and *115*
tie, ios member function *188*
tied streams *188*
time *See also* dates
macro *170*
__TIME__ macro *170*
#define and #undef directives and *157*
TITLE option *342*
Linker (linker)
segment limit *281*
Linker
warnings
defined *239*
list *239*
tokens
continuing long lines of *159*
kinds of *8*
parsing *6*
pasting *7, 159*
replacement *155*
replacing and merging *24*
topic numbers
Help compiler *237*
translation units *31*
truth table
bitwise operators *93*
Turbo Assembler *207*
Turbo C++
extensions *9*
keywords
using as identifiers *3*
__TURBOC__ macro *170*
type-safe
lists *152*
type-safe linkage *See* linkage, type-safe
type specifiers
elaborated *112*
pure *37*
type taxonomy *38*
typecasting
pointers *59*
typed constants *See* constants, data type
typedef (keyword) *46*
name space *28*
structure tags and *66*
structures and *66*
typedefs
untagged structures and *66*
types *See* data types

U

UINT_MAX (constant) *89*
ULONG_MAX (constant) *89*
unary operators *81*
minus (–) *81, 86*
plus (+) *81, 86*
syntax *85*
unbuffered, streambuf member function *196*
#undef directive *156*
global identifiers and *157*
underbars *See* underscores
underflow
filebuf member function *183*
strstreambuf member function *197*
underscores
generating *48*
ignoring *48*
union (keyword)
C++ *112*
unions *71*
anonymous
member functions and *72*
base classes and *120*
bit fields and *See* bit fields
C++ *73, 111*
C vs. *112*
classes and *73*
constructors and destructors and *125*
declarations *73*
initialization *43, 73*
members
access *83, 119*
name space *28*
sizeof and *72*

structures vs. *71*
units, translation *See* translation units
unsetf (function) *177*
unsetf, ios member function *188*
unsigned (keyword) *40*
untagged structures *See* structures, untagged
user-defined formatting flags *188*

V

\v (vertical tab character) *15*
value, passing by *See* parameters
values
comparing *90*
var, passing by *See* parameters
variable number of arguments *23*
variables
automatic *See* auto (keyword)
declaring *44*
external *45*
global *See* global variables
initializing *44*
internal linkage *46*
name space *28*
offsets in inline assembly code *212*
pseudo *See* pseudovariables
register *See* registers, variables
volatile *49*
variant record types *See* unions
vectors, interrupt *See* interrupts
vertical tab *15*
virtual
base classes *See* classes, base, virtual
destructors *See* destructors, virtual
functions *See* member functions, virtual
tables
32-bit pointers and *233*
DLLs and *233*
storing in the code segment *233*
virtual (keyword)
constructors and destructors and *124*
functions and *140*
visibility *29*, *See also* scope
C++ *29*
pointers *56*
scope and *29*
void (keyword) *39*
function pointers and *55*
functions and *62*
interrupt functions and *49*
pointers *56*
typecasting expressions and *39*
volatile (keyword) *47, 49*
formal parameters and *64*

W

WARNING option *340*
warnings
Compile-time *236*
defined *236*
disabling *166*
Help compiler *237*
Librarian *239*
Linker
defined *239*
Linker (list) *239*
wchar_t (wide character constants) *16*
arrays and *44*
WEP (function) *231*
return values *231*
WHELLO (Windows program) *218*
compiling and linking *219*
while loops *See* loops, while
whitespace *6*
comments and *8*
comments as *6*
extracting *176*
wide character constants (wchar_t) *16*
width, ios member function *188*
Windows
_export and *225*
modules
compiling and linking *217*
prolog and epilog code *222*
Windows (Microsoft) *See* Microsoft Windows
Windows All Functions Exportable command *222*
Windows applications *217-234*
command-line compiler options *222, 223, 224*
_export and *224*
export functions and *222*
IDE and *221*
linking *228*
memory models and *226*

prolog and epilog code *222*
resource linker and *217*
smart callbacks and *223*
WHELLO *218*
WinMain function and *221*
Windows DLL All Functions Exportable command *224*
Windows DLL Explicit Functions Exported command *224*
Windows Explicit Functions Exported command *222*
Windows Smart Callbacks command *223*
_Windows macro *170*
WinMain (function) *221*
return value *222*
–WN BCC options (.OBJs with explicit exports) *222*
word alignment *69*
memory and
structures *69*
write, ostream member function *192*
ws (manipulator) *176*
–WS BCC options (smart callbacks) *223*

X

x_fill, ios data member *186*
x_flags, ios data member *186*
x_precision, ios data member *186*
x_tie, ios data member *186*
x_width, ios data member *186*
xalloc, ios member function *188*
\xH (display a string of hexadecimal digits) *15*
XOR operator (^) *81, 93*
truth table *93*