Tools & Utilities Guide

Borland® C++

for OS/2®

Tools and Utilities Guide

Borland® C++ for OS/2®

Version 1.5

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1987, 1994 by Borland International. All rights reserved. All Borland product names are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

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

PRINTED IN THE UNITED STATES OF AMERICA

1E0R0294
9495969798-987654321
H1

Contents

Tables

Introduction

Borland C++ comes with a host of powerful standalone utilities that you can use with your Borland C++ files or other modules to ease your DOS and Windows programming.

This manual describes IMPDEF, IMPLIB, MAKE, TLIB, and TLINK and uses code and command-line examples to show how to use them. It also describes the Borland and OS/2 resource compilers. The rest of the Borland C++ utilities are documented in a text file called UTIL.DOC, which the INSTALL utility places in the DOC subdirectory. The Borland C++ error messages are listed and described in Appendix A.

Name	Description
Documented in this book	
TLINK	Turbo Linker (see Chapter 1)
MAKE	Standalone program manager (see Chapter 2)
TLIB	Turbo Librarian (see Chapter 3)
IMPLIB	Generates an import library (see Chapter 4)
IMPDEF	Creates a module definition file (see Chapter 4)
BRCC.EXE	Borland Resource Compiler (see Chapter 5)
RC.EXE	OS/2 Resource Compiler (See Chapter 5)
Documented in the online document UTIL.DOC	
CPP	Preprocessor
GREP	File-search utility
OBJXREF	Object module cross-reference
TEMC	Turbo Editor Macro Compiler
TOUCH	Updates file date and time
TRIGRAPH	Character-conversion utility

C H A P T E R 1

TLINK: The Turbo linker

This chapter explains the operation of Borland's command-line linker TLINK. TLINK combines object modules and library modules to produce .EXE files. When you invoke the command-line compiler BCC, TLINK is invoked automatically unless you suppress the linking stage with the **-c** compiler option. If you suppress the linking stage, you must invoke TLINK manually, as described in the next section.

Invoking TLINK

TLINK options are case-sensitive.

The general syntax of a TLINK command line is

```
TLINK options objfiles, exefile, mapfile, libfiles, deffile
```

This syntax specifies that you supply file names in the given order, and that you separate the file types with commas. You can invoke TLINK at the command line by typing TLINK with or without parameters. TLINK parameters are either options or file names.

Options let you control TLINK's output. For example, they let you specify whether you want to produce an .EXE or a DLL file. TLINK options must be immediately preceded by either a forward slash (/) or a hyphen (-). When invoked without parameters, TLINK displays an option summary. Table 1.1 briefly describes the TLINK options.

Table 1.1 TLINK options

TLINK options can also be preceded by a hyphen (-).

Option	What it does
/a*x*	Specifies application type, where /**aa** targets PM applications /**ai** targets full-screen character mode applications /**ap** targets PM-compatible character mode applications.
/**A:***dd*	Specifies page alignment within .EXE file.
/**B:***xxxxxx*	Specifies image base address (in hexadecimal).
/**c**	Treats case as significant in symbols.
/**E***nn*	Specifies maximum errors before termination.

Table 1.1: TLINK options (continued)

/Gm	Write mangled names in map file.
/L	Specifies library search paths.
/m	Creates map file with publics.
/Oc	Directs TLINK to use fixup-chaining optimization.
/s	Creates detailed map of segments.
/S:*xxxxxx*	Specifies stack size (in hexadecimal).
/To*x*	Specifies target, where **/Toe** means build an .EXE file. **/Tod** means build a DLL.
/v	Includes full symbolic debug information.
/w*xxx*	Enable or disable warnings (see page 12).
/x	Doesn't create map file.

File names can be grouped into different file types:

- *objfiles* specifies the .OBJ files you want linked into an .EXE or .DLL file.
- *exefile* specifies the name you want for the resulting .EXE or .DLL file.
- *mapfile* specifies the name you want for the link map file. If not given, the map file name is the same as *exefile*.
- *libfiles* specifies the library files you want to link with.
- *deffile* specifies the module definition file containing linker information.

Using TLINK with Borland C++ modules

When you create an executable Borland C++ file using TLINK, you must include the initialization module and libraries.

The general format for linking Borland C++ programs with TLINK is

```
TLINK options C02[D] myobjs, exename, [mapfile], [mylibs] [C2|C2MT] [OS2],
[deffile]
```

where

- *myobjs* are the .OBJ files you want linked. You must specify the path if the files are not in the current directory.
- *exename* is the name you want given to the executable file.

(optional)

- *mapfile* is the name you want given to the map file.

(optional)

- *mylibs* are the library files you want included at link time. You must specify the path if not in current directory, or use the **/L** option to specify search paths.
- *deffile* is the module definition file for a PM executable.

Be sure to include paths for the startup code and libraries (or use the **/L** option to specify a list of search paths for startup and library files).

The other file names on this general TLINK command line represent Borland C++ files, as follows:

- C02 and C02D are the Borland initialization modules for programs or DLLs. One of these must always be the first .OBJ file in the list.
- C2 is the Borland run-time library. C2MT is the multi-thread version.
- OS2 is the OS/2 import library. OS2.LIB provides access to the OS/2 API functions.

Initialization modules

C02.OBJ and C02D.OBJ are the application initialization modules for C and C++ code. When your program is executed, this code is run first. Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, or that no stack has been created.

The initialization module must appear first in the object file list. The initialization module arranges the order of the various segments of the program. If it is not first, the program segments might not be placed in memory properly, causing some frustrating program bugs.

Be sure you give an explicit name for the executable file name on the TLINK command line. Otherwise, your program name will be something like C02.EXE—probably not what you wanted.

See the *Library Reference*, Appendix A, for a summary of the libraries and .OBJ files provided by Borland.

If you want to create an OS/2 application or DLL, you must link OS2.LIB to provide access to the OS/2 API functions.

A TLINK example

To create a PM application executable, you might use this command line:

```
TLINK /Toe /c \BCOS2\lib\c02 pmapp1 pmapp2, pmapp, pmapp, \BCOS2\lib\c2
  \BCOS2\lib\os2, pmapp.def
```

where

Letting the command-line compiler, or the IDE project manager take care of linking is easier. They both provide correct options and libraries.

- The **/Toe** option tells TLINK to generate PM executables.
- The **/c** option tells TLINK to be case-sensitive during linking.
- BCOS2\LIB\C02 is the standard PM initialization file and PMAPP1 and PMAPP2 are the module's object files. The .OBJ extension is assumed for all three files.

- PMAPP.EXE is the name of the target PM executable. The .EXE extension is assumed.
- PMAPP.MAP is the name of the map file. The .MAP extension is assumed.
- BCOS2\LIB\C2 is the OS/2 run-time library, and BCOS2\LIB\OS2 is the import library that provides access to the OS/2 Application Program Interface (API) functions.
- PMAPP.DEF is the module definition file used to specify additional link options.

Invoking TLINK with BCC

You can also use BCC, the standalone Borland C++ compiler, as a "front end" to TLINK that invokes TLINK with the correct startup file, libraries, and executable program name.

See Chapter 6, "Command-line compiler," in the *User's Guide* for more on BCC.

To do this, you give file names on the BCC command line with explicit .OBJ and .LIB extensions. For example, given the following BCC command line,

```
BCC main.obj sub1.obj mylib.lib
```

BCC invokes TLINK with the files C02.OBJ, C2.LIB, and OS2.LIB (initialization module, run-time library, and OS/2 API import library). TLINK will link these along with your own modules MAIN.OBJ and SUB1.OBJ, and your own library MYLIB.LIB, producing file MAIN.EXE.

If you are producing multi-threaded executables, invoke BCC like this:

```
BCC -sm main.obj sub1.obj func3.obj
```

The **–sm** switch tells BCC to invoke TLINK with the multi-thread library C2MT.LIB.

To use BCC for linking a PM DLL, invoke BCC like this:

```
BCC -sd FUNC1.OBJ FUNC2.OBJ FUNC3.OBJ
```

The **–sd** switch tells BCC to invoke TLINK with the DLL initialization module CO2D.OBJ.

When BCC invokes TLINK, it uses the **/c** (case-sensitive link) option by default.

File-name extensions on the TLINK command line

If you don't specify an executable file name, TLINK derives the name of the executable by appending .EXE or .DLL to the first object file name listed.

If no map file name is given, TLINK adds a .MAP extension to the .EXE file name. If no libraries are included, none will be linked. If you don't specify a module definition (.DEF) file and you have used the **/Toe** or **/Tod** option, TLINK creates an application based on default settings.

TLINK assumes or appends these extensions to file names that have none:

- .OBJ for object files.
- .EXE for OS/2 executable files (when you use the **/Toe** option).
- .DLL for dynamic-link libraries (when you use the **/Tod** option).
- .MAP for map files.
- .LIB for library files.
- .DEF for module definition files.

All of the file names except object files are optional. So, for example,

```
TLINK myapp myapp2
```

links the files MYAPP.OBJ and MYAPP2.OBJ, creates an executable file called MYAPP.EXE, creates a map file called MYAPP.MAP, links no libraries, and uses no module definition file.

Using response files

TLINK accepts its parameters not only from the command line, but also from a response file (or any combination of the two).

A response file is a text file that contains the options and file names that you would usually type after the name TLINK on your command line. This saves you from having to type the full command line each time you link.

Unlike the command line, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line. When a plus occurs at the end of a line, but it immediately follows one of the TLINK options that uses + to enable the option (such as **/v+**), the + is not treated as a line-continuation character.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the command line

```
TLINK /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

with the following response file, FINRESP:

```
/c mainline wd+
   ln tx,fin
   mfin
   work\lib\comm work\lib\support
```

You would then enter your TLINK command as

```
TLINK @finresp
```

Note that you must precede the file name with the @ character to indicate that the next name is a response file.

Alternatively, you could break your link command into multiple response files. For example, you could break the previous command line into the following two response files:

File name	Contents

LISTOBJS	`mainline+`
	`wd+`
	`ln tx`

LISTLIBS	`lib\comm+` `lib\support`

You would then enter the TLINK command as

```
tlink /c @listobjs,fin,mfin,@listlibs
```

The TLINK configuration file

The command-line version of TLINK looks for a file called TLINK.CFG, first in the current directory, then in the directory from which it was loaded.

TLINK.CFG is a text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked.

For example, the following TLINK.CFG file

```
/Lc:\BCOS2\lib;c:\winapps\lib
/v /s
/Toe
```

tells TLINK to search the specified directories for libraries, include debug information, create a detailed segment map, and produce an OS/2 program.

TLINK options

TLINK options can occur anywhere on the command line. The options consist of a slash (/), a hyphen (–), followed by the option. Options are case-sensitive.

If you have more than one option, spaces are not significant (**/m/c** is the same as **/m /c**), and you can have them appear in different places on the command line. The following sections describe each of the options.

/a (application type)

The **/a** option lets you specify one of three types of EXE images:

- **/aa** targets windowing applications; that is, applications that run in and use the Graphical User Interface (GUI) environment.
- **/ai** targets full-screen, character-mode applications.
- **/ap** targets character-mode applications that can be run in a window.

/A (align pages)

The **/A** option specifies page alignment for code and data within the executable file. The syntax is

/A:*dd*

dd must be a decimal power of 2. For example, if you specify an alignment value of **/A:12**, the pages of code and data are stored on 4096-byte boundaries. OS/2 seeks pages for loading based on this alignment value. The default is **/A:9**, which means pages are aligned on 512-byte boundaries within the executable file. Larger alignment values result in larger executable files, but can improve demand load performance.

/B (base address)

The **/B** option specifies an image base address for an application. If this option is used, internal fixups are removed from the image, and the requested load address of the first object is set to the hexadecimal number given with the option. All successive objects are aligned on 64K linear address boundaries.

Using this option makes applications smaller on disk, and improves both load-time and run-time performance, since the operating system no longer has to apply internal fixups. Since OS/2 loads all .EXE images at 64K, you're advised to link all .EXEs with **/B:0x10000**.

DLLs are loaded at arbitrary addresses, and should not be linked with the **/B** option.

/c (case sensitivity)

The **/c** option forces the case to be significant in public and external symbols.

/E (maximum errors)

The **/E** option lets you specify the maximum number of errors the linker reports before terminating. **/E0** means report an infinite number of errors (that is, as many as occur), and is the default.

/Gm (Goodies)

The **/Gm** option puts mangled names in a mapfile.

/L (library search paths)

The **/L** option lets you specify a list of directories that TLINK searches for libraries if an explicit path is not specified. TLINK searches the current directory before those specified with the **/L** option. For example,

```
TLINK /Lc:\BCOS2\lib;c:\mylibs splash logo,,,utils .\logolib
```

With this command line, TLINK first searches the current directory for UTILS.LIB, then searches C:\BCOS2\LIB and C:\MYLIBS. Because .\ LOGOLIB explicitly names the current directory, TLINK does not search the libraries specified with the **/L** option to find LOGOLIB.LIB.

TLINK also searches for the C or C++ initialization module (C02.OBJ, or C02D.OBJ) on the specified library search path.

/m, /s, and /x (map options)

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error messages produced during the link. If you don't want to create a map, turn it off with the **/x** option.

If you want to create a more complete map, the **/m** option adds a list of public symbols to the map file, sorted alphabetically as well as in increasing address order. Many debuggers can use the list of public symbols, which let you refer to symbolic addresses when you are debugging.

The **/s** option creates a map file with segments, public symbols and the program start address just like the **/m** option does, but also adds a detailed segment map. For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Except for the ACBP field, the information in the detailed segment map is self-explanatory.

The ACBP field encodes the A (alignment), C (combination), and B (big) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (alignment)	00	An absolute segment.
	20	A byte-aligned segment.
	40	A word-aligned segment.
	60	A paragraph-aligned segment.
	80	A page-aligned segment.
	A0	An unnamed absolute portion of storage.
The C field (combination)	00	Cannot be combined.
	08	A public combining segment.
The B field (big)	00	Segment less than 64K.
	02	Segment exactly 64K.
The P field	00	Segment is USE16.
	01	Segment is USE32.

When you request a detailed map with the **/s** option, the list of public symbols (if it appears) has public symbols flagged with "idle" if there are no references to that symbol. For example, this fragment from the public symbol section of a map file indicates that symbols *Symbol1* and *Symbol3* are not referenced by the image being linked:

```
0002:00000874    Idle    Symbol1
0002:00000CE4            Symbol2
0002:000000E7    Idle    Symbol3
```

/Oc (chain fixups)

The **/Oc** option directs TLINK to use the fixup-chaining optimization. This optimization collapses multiple fix-up records for the same address into one fix-up record. Though this optimization causes slower link time, it shrinks executable size and speeds up loading.

/S (stack size)

The **/S** option lets you set the application stack size, in hexadecimal. The form is **/S**:*xxxxxx*, where *xxxxxx* is a hexadecimal string. Specifying stack size with **/S** overrides any stack size setting in a module definition file.

/T (output file type)

The **/T** option specifies whether you are producing a program or DLL. **/Toe** tells the linker to produce a .EXE image, and **/Tod** tells the linker to produce a DLL image.

/v (debugging information)

The **/v** option directs TLINK to include debugging information in the executable file. If this option is found anywhere on the command line, debugging information will be included in the executable file for all object modules that contain debugging information. You can use the **/v+** and **/v-**

options to selectively enable or disable inclusion of debugging information on a module-by-module basis (but not on the same command line as **/v**). For example, the command

```
TLINK mod1 /v+ mod2 mod3 /v- mod4
```

includes debugging information for modules *mod2* and *mod3*, but not for *mod1* and *mod4*. When **/v-** is seen, debug information is turned off until **/v+** is seen.

/w (warning control)

The **/w** option lets you control specific warnings emitted by TLINK. With this switch you can either enable or disable select warnings. This switch can be used in the following ways:

- **/w***xxx* or **/w+***xxx* enables warnings.
- **/w-***xxx* disables warnings.
- **/w!** enables all warnings.

The value *xxx* can be one of the following:

- **ent** controls the "No entry point" warning.
- **dup** controls the "Duplicate symbol" warning for .OBJs.
- **stk** controls the "No stack" warning.
- **def** controls the "No .DEF file; using defaults" warning.
- **imt** controls the "Import does not match previous definition" warning.
- **msk** controls the "Multiple stack segments found" warning.
- **bdl** controls the "Using based linking for DLLs may cause the DLL to malfunction" warning.
- **srf** controls the "Self-relative fixup overflowed" warning.
- **dpl** controls the "Duplicate symbol" warning for libraries.

By default ent, dup, stk, srf, and bdl are enabled, and the rest are disabled.

Module definition reference

This section describes each statement in a module definition file. The module definition file provides information to the linker about the contents and system requirements of a PM application. More specifically, it

- Names the application or dynamic-link library (DLL).
- Identifies the type of application.
- Lists imported functions and exported functions.

- Describes the code and data segment attributes, and lets you specify attributes for additional code and data segments.
- Specifies the size of the stack.
- Provides for the inclusion of a stub program.

Note that the IMPDEF utility creates module definition files, and the IMPLIB utility creates import libraries out of module definition files. See Chapter 4, "Import library tools," for more information on these tools.

CODE statement

The CODE statement defines the default attributes of code segments. Code segments can have any name, but must belong to segment classes whose name ends in CODE. For example, valid segment class names are CODE or MYCODE. The syntax is

```
CODE [PRELOAD | LOADONCALL]
     [EXECUTEONLY | EXECUTEREAD]
     [CONFORMING | NONCONFORMING]
```

PRELOAD means code is loaded when the calling program is loaded. LOADONCALL (the default) means the code is loaded when called.

EXECUTEONLY means a code segment can be executed only. EXECUTEREAD (the default) means the code segment can be read and executed.

CONFORMING means a code segment can be called from Ring 2 or Ring 3. (This is also known as 80286 conforming.) NONCONFORMING (the default) means the code segment is not 80286 conforming.

DATA statement

The DATA statement defines attributes of data segments.

The syntax of the DATA statement is

```
DATA [NONE | SINGLE | MULTIPLE]
     [READONLY | READWRITE]
     [PRELOAD | LOADONCALL]
     [SHARED | NONSHARED]
```

NONE means no data segment is created. (This option is available only for libraries.) SINGLE means a single data segment is created and shared by all processes, and is the default for DLLs. MULTIPLE means a data segment is created for each process, and is the default for programs.

READONLY means the data segment can be read only. READWRITE (the default) means the data segment can be read and written to.

PRELOAD means the data segment is loaded when a module that uses it is first loaded. LOADONCALL (the default) means the data segment is loaded when it is first accessed.

SHARED (the default for DLLs) means one copy of the data segment is shared among all processes. NONSHARED (the default for programs) means a copy of the data segment is loaded for each process needing to use the data segment.

DESCRIPTION statement

The DESCRIPTION statement, which is optional, inserts text into the application module. The DESCRIPTION statement is typically used to embed author, date, or copyright information. The syntax is

```
DESCRIPTION 'Text'
```

Text specifies an ASCII string delimited with single quotes.

EXETYPE statement

The EXETYPE statement specifies the default executable file (.EXE) header type. The syntax is

```
EXETYPE  [WINDOWAPI] | [WINDOWCOMPAT] | [NOTWINDOWCOMPAT]
```

WINDOWAPI specifies a PM executable, and is equivalent to the TLINK option **/aa**.

WINDOWCOMPAT specifies a PM-compatible character-mode executable, and is equivalent to the TLINK option **/ap**.

NOTWINDOWCOMPAT specifies a character-mode application that won't run under PM. It is equivalent to the TLINK option **/ai**.

EXPORTS statement

The EXPORTS statement defines the names and attributes of functions to be exported. The EXPORTS keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line. The syntax is

```
EXPORTS
   ExportName [Ordinal] [RESIDENTNAME] [Parameter]
```

ExportName specifies an ASCII string that defines the symbol to be exported. It has the following form:

```
EntryName [=InternalName]
```

InternalName is the name used within the application to refer to this entry. *EntryName* is the name listed in the executable file's entry table and is externally visible.

Ordinal defines the function's ordinal value. It has the following form:

```
@ordinal
```

where *ordinal* is an integer value that specifies the function's ordinal value.

When an application module or DLL module calls a function exported from a DLL, the calling module can refer to the function by name or by ordinal value. In terms of speed, referring to the function by ordinal is faster because string comparisons are not required to locate the function. In terms of memory allocation, exporting a function by ordinal (from the point of view of that function's DLL) and importing/calling a function by ordinal (from the point of view of the calling module) is more efficient. When a function is exported by ordinal, the name resides in the nonresident name table. When a function is exported by name, the name resides in the resident name table. The resident name table for a module is resident in memory whenever the module is loaded; the nonresident name table isn't.

The RESIDENTNAME option lets you specify that the function's name must be resident at all times. This is useful only when exporting by ordinal (when the name wouldn't be resident by default).

Parameter is an optional integer value that specifies the number of words the function expects to be passed as parameters.

IMPORTS statement

The IMPORTS statement defines the names and attributes of functions to be imported from DLLs. Instead of listing imported DLL functions in the IMPORTS statement, you can either specify an import library for the DLL in the TLINK command line, or—in the IDE—include the import library for the DLL in the project.

The IMPORTS keyword marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line. The syntax is

```
IMPORTS
   [InternalName=]ModuleName.Entry
```

InternalName is an ASCII string that specifies the unique name that the application will use to call the function.

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module that contains the function. The module name must match the name of the executable file. For example, the file SAMPLE.DLL has the module name SAMPLE.

Entry specifies the function to be imported. It can be either an ASCII string that names the function, or an integer that gives the function's ordinal value.

LIBRARY statement

The LIBRARY statement defines the name of a DLL module. A module definition file can contain either a NAME statement to indicate a program or a LIBRARY statement to indicate a DLL, but not both.

Like a program's module name, a library's module name must match the name of the executable file. For example, the library MYLIB.DLL has the module name MYLIB. The syntax is

```
LIBRARY [LibraryName] [INITGLOBAL | INITINSTANCE]
```

LibraryName specifies an ASCII string that defines the name of the library module. *LibraryName* is optional. If the parameter is not included, TLINK uses the file-name part of the executable file (that is, the name with the extension removed). If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

INITGLOBAL means that the library-initialization routine is called only when the library module is first loaded into memory. INITINSTANCE means the library-initialization routine is called each time a new process makes use of the library.

NAME statement

The NAME statement defines the name of the application's executable module. The module name identifies the module when exporting functions. The syntax is

```
NAME ModuleName
```

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name SAMPLE.

The *ModuleName* parameter is optional. If the parameter is not included, TLINK assumes that the module name matches the file name of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, TLINK assumes that the module name is MYAPP.

If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

SEGMENTS statement

The SEGMENTS statement defines the segment attributes of additional code and data segments. The syntax is

```
SEGMENTS
   SegmentName [CLASS 'ClassName'] [MinAlloc]
   [PRELOAD | LOADONCALL]
   [SHARED | NONSHARED]
```

SegmentName specifies a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.

ClassName is an optional string that specifies the class name of the specified segment. If no class name is specified, TLINK uses the class name CODE by default.

MinAlloc is an optional integer value that specifies the minimum allocation size for the segment. Currently, TLINK ignores this value.

PRELOAD means the segment is loaded immediately; LOADONCALL means the segment is loaded when it is accessed or called. The Resource Compiler might override the LOADONCALL option and preload segments instead.

SHARED (the default for DLLs) means one copy of the data segment is shared among all processes. NONSHARED (the default for programs) means a copy of the data segment is loaded for each process needing to use the data segment.

Default attributes for additional segments are the same as described previously for CODE and DATA segments (depending on the type of additional segment).

STACKSIZE statement

The STACKSIZE statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it makes function calls. Do not use the STACKSIZE statement for dynamic-link libraries. The syntax is

```
STACKSIZE bytes
```

bytes is an integer value that specifies the stack size in bytes.

STUB statement

The STUB statement appends a DOS executable file specified by *FileName* to the beginning of the module. The executable stub should display a warning message and terminate if the user attempts to run the executable stub in the wrong environment (running a PM application under DOS, for example).

Borland C++ adds a built-in stub to the beginning of a PM application unless a different stub is specified with the STUB statement. Therefore, you should not use the STUB statement merely to include OS2STUB.EXE because the linker will do this for you automatically.

The syntax is

```
STUB "FileName"
```

FileName specifies the name of the DOS executable file that will be appended to the module. The name must have the DOS file name format.

If the file named by *FileName* is not in the current directory, TLINK searches for the file in the directories specified by the PATH environment variable.

C H A P T E R 2

Make: The program manager

MAKE.EXE is a command-line project-manager utility that helps you quickly compile only those files in a project that have changed since the last compilation. (MAKER is a real-mode version of MAKE.)

This chapter covers the following topics:

- MAKE basics
- Makefile contents
- Using explicit and implicit rules
- Using MAKE macros
- Using MAKE directives

MAKE basics

MAKE uses rules from a text file (MAKEFILE or MAKEFILE.MAK by default) to determine which files to build and how to build them. For example, you can get MAKE to compile an .EXE file if the date-time stamps for the .CPP files that contain the code for the .EXE are more recent than the .EXE itself. MAKE is very useful when you build a program from more than one file because MAKE will recompile only the files that you modified since the last compile.

Two types of rules (explicit and implicit) tell MAKE what files depend on each other. MAKE then compares the date-time stamp of the files in a rule and determines if it should execute a command (the commands usually tell MAKE which files to recompile or link, but the commands can be nearly any operating system command).

MAKE accepts * and ? as wildcards.

The general syntax for MAKE is

```
MAKE [options...] [targets[s]]
```

To get command-line help for MAKE, type `MAKE -?` or `MAKE -h`.

where `options` are MAKE options that control how MAKE works, and `targets` are the names of the files in a makefile that you want MAKE to build. Options are separated from `MAKE` by a single space. Options and targets are also separated by spaces.

If you type MAKE at the command prompt, MAKE performs the following default tasks:

To place MAKE instructions in a file other than MAKEFILE, see the section titled "MAKE options."

1. MAKE looks in the current directory for a file called BUILTINS.MAK (this file contains rules MAKE always follows unless you use the **–r** option). If it can't find the file in the current directory, it looks in the directory where MAKE.EXE is stored. After loading BUILTINS.MAK, MAKE looks for a file called MAKEFILE or MAKEFILE.MAK. If MAKE can't find any of these files, it gives you an error message.
2. When MAKE finds a makefile, it tries to build *only* the first target file in the makefile (although the first target can force other targets to be built). MAKE checks the time and date of the dependent files for the first target. If the dependent files are more recent than the target file, MAKE executes the target commands, which update the target. See the section called "Using makefiles" for more information on instructions in makefiles.
3. If a dependent file for the first target appears as a target elsewhere in the makefile, MAKE checks its dependencies and builds it before building the first target. This chain reaction is called linked dependency.
4. If the MAKE build process fails, MAKE deletes the target file it was building. To get MAKE to keep a target when a build fails, see the **.precious** directive on page 36.

You can stop MAKE by using *Ctrl+Break* or *Ctrl+C*.

BUILTINS.MAK

BUILTINS.MAK contains standard rules and macros that MAKE uses before it uses a makefile (you can use the **–r** option to tell MAKE to ignore BUILTINS.MAK). Use BUILTINS.MAK for instructions or macros you want executed each time you use MAKE. Here's the default text of BUILTINS.MAK:

```
#
# Borland C++ - (C) Copyright 1992 by Borland International
#
CC = BCC
AS = TASM
RC = RC
.asm.obj:
      $(AS) $(AFLAGS) $&.asm
.c.exe:
      $(CC) $(CFLAGS) $&.c
.c.obj:
      $(CC) $(CFLAGS) /c $&.c
.cpp.obj:
      $(CC) $(CPPFLAGS) /c $&.cpp
```

```
.rc.res:
        $(RC) $(RFLAGS) /r $&
.SUFFIXES: .exe .obj .asm .c .res .rc
```

Using TOUCH.EXE

Sometimes you'll want to force a target file to be recompiled or rebuilt even though you haven't changed it. One way to do this is to use the TOUCH utility. TOUCH changes the date and time of one or more files to the current date and time, making it "newer" than the files that depend on it.

You can force MAKE to rebuild a target file by *touching* one of the files that target depends on. To touch a file (or files), type the following at the command prompt:

```
touch filename [filename...]
```

You can use wildcards * and ? with TOUCH.

TOUCH updates the file's creation date and time.

Important! Before you use TOUCH, make sure your system's internal clock is set correctly. If it isn't, TOUCH and MAKE won't work properly.

MAKE options

Command-line options control MAKE behavior. Options are case-sensitive. Type options with either a preceding – or /. For example, to use a file called PROJECTA.MAK as the makefile, type `MAKE -fPROJECTA.MAK` (a space after **–f** is optional). Many of the command-line options have equivalent directives that are used in the makefile (see page 32 for more information on directives).

Table 2.1: MAKE options

Option	Description
–h or –?	Displays MAKE options and shows defaults with a trailing plus sign.
–D*macro*	Defines *macro* as a single character, causing an expression **!ifdef** *macro* written in the makefile to return true.
[-D]*macro*=[*string*]	Defines *macro* as *string*. If *string* contains any spaces or tabs, enclose *string* in quotation marks. The **–D** is optional.
–I*directory*	Searches for include files in the current directory first, then in *directory*.
–K	Keeps temporary files that MAKE creates (MAKE usually deletes them). See also KEEP on page 22.
–N	Executes MAKE like Microsoft's NMAKE (see the section following this table for more information).
–U*macro*	Undefines previous definitions of *macro*.
–W	Writes the current specified non-string options to MAKE.EXE making them defaults.
–f*filename*	Uses *filename* or *filename*.MAK instead of MAKEFILE (space after **–f** is optional).

Table 2.1: MAKE options (continued)

-a	Checks dependencies of include files and nested include files associated with .OBJ files and updates the .OBJ if the .H file changed. See also **-c**.
-c	Caches autodependency information, which can improve MAKE's speed. Use with -a; don't use if MAKE changes include files (such as using TOUCH from a makefile or creating header or include files during the MAKE process).
-e	Ignores a macro if its name is the same as an environment variable (MAKE uses the environment variable instead of the macro).
-i	Ignores the exit status of all programs run from MAKE and continues the build process.
-m	Displays the date and time stamp of each file as MAKE processes it.
-n	Prints the commands but doesn't actually perform them, which is helpful for debugging a makefile.
-p	Displays all macro definitions and implicit rules before executing the makefile.
-q	Returns 0 if the target is up-to-date and nonzero if is is not (for use with batch files).
-r	Ignores any rules defined in BUILTINS.MAK.
-s	Suppresses onscreen command display.

Setting options on as defaults

The **-W** option lets you set some MAKE options on as defaults so that each time you use MAKE, those options are used. To set MAKE options, type

```
make -option[-] [-option][-] . . . -W
```

For example, you could type `MAKE -m -W` to always view file dates and times. Type `MAKE -m- -W` to turn off the default option. When MAKE asks you to write changes to MAKE.EXE, type `Y`.

Caution! The **-W** option doesn't work when the DOS Share program is running. The message `Fatal: unable to open file MAKE.EXE` is displayed. The **-W** option doesn't work with the following MAKE options:

- **-D***macro*
- **-D***macro=string*
- **-U***symbol*
- **-f***filename*
- **-?** or **-h**
- **-I***directory*

Compatibility with Microsoft's NMAKE

Use the **-N** option if you want to use makefiles that were originally created for Microsoft's NMAKE. The following changes occur when you use **-N**:

- MAKE interprets the **<<** operator like the **&&** operator: temporary files are used as response files, then deleted. To keep a file, either use the **-K** command-line option or use KEEP in the makefile.

  MAKE usually deletes temporary files it creates.

```
<<TEMPFILE.TXT!
text
```

```
:
!KEEP
```

If you don't want to keep a temporary file, type NOKEEP or type only the temporary file name. If you use NOKEEP with a temporary file, then use the **–K** option with MAKE, MAKE deletes the temporary file.

- The **$d** macro is treated differently. Use **!ifdef** or **!ifndef** instead.
- Macros that return paths won't return the last \. For example, if `$(<D)` normally returns `C:\CPP\`, the **–N** option makes it return `C:\CPP`.
- Unless there's a matching **.suffixes** directive, MAKE searches rules from bottom to top of the makefile.
- The **$*** macro always expands to the target name instead of the dependent in an implicit rule.

Using makefiles

A makefile is an ASCII file of instructions for MAKE.EXE. MAKE assumes your makefile is called MAKEFILE or MAKEFILE.MAK unless you use the **–f** option (see page 21).

MAKE either builds targets you specify at the MAKE command line or it builds *only* the first target it finds in the makefile (to build more than one target, see the section "Symbolic targets.") Makefiles can contain:

- Comments
- Explicit rules
- Implicit rules
- Macros
- Directives

Symbolic targets

A symbolic target forces MAKE to build multiple targets in a makefile (you don't need to rely on linked dependencies). The dependency line lists all the targets you want to build. You don't type any commands for a symbolic target.

In the following makefile, the symbolic target *allFiles* builds both FILE1.EXE and FILE2.EXE.

```
allFiles: file1.exe file2.exe     #Note this target has no commands.
file1.exe: file1.obj
   bcc file1.obj
file2.exe: file2.obj
   bcc file2.obj
```

Rules for symbolic targets

Observe the following rules with symbolic targets:

- Symbolic targets don't need a command line.
- Give your symbolic target a unique name; it can't be the name of a file in your current directory.
- Name symbolic targets according to the operating system rules for naming files.

Explicit and implicit rules

The explicit and implicit rules that instruct MAKE are generally defined as follows:

- Explicit rules give MAKE instructions for specific files.
- Implicit rules give general instructions that MAKE follows when it can't find an explicit rule.

Rules follow this general format:

```
Dependency line
  Commands
  :
```

The dependency line is different for explicit and implicit rules, but the commands are the same (for information on linked dependencies see page 20).

MAKE supports multiple rules for one target. You can add dependent files after the first explicit rule, but only one should contain a command line. For example,

```
Target1: dependent1 dep2 dep3 dep4 dep5
Target1: dep6 dep7 dep8
  bcc -c $**
```

Explicit rule syntax

Explicit rules are instructions to MAKE that specify exact file names. The explicit rule names one or more targets followed by one or two colons. One colon means one rule is written for the target; two colons mean that two or more rules are written for the target.

Explicit rules follow this syntax:

Braces must be included if you use the paths parameter.

```
target [target...]:[:][{path}] [dependent[s]...]
   [commands]
   :
```

- target — The name and extension of the file to be updated (*target* must be at the start of the line—no spaces or tabs are allowed). One or more targets must be separated by spaces or tabs. Don't use a target's name more than once in the target position of an explicit rule in a makefile.
- path — A list of directories, separated by semicolons and enclosed in braces, that points to the dependent files.
- dependent — The file (or files) whose date and time MAKE checks to see if it is newer than `target` (dependent *must* be preceded by a space). If a dependent file also appears in the makefile as a target, MAKE updates or creates the target file before using it as a dependent for another target.
- commands — Any operating system command. Multiple commands are allowed in a rule. Commands must be indented by at least one space or tab (see the section on commands on page 27).

If the dependency or command continues on to the next line, use the backslash (\) at the end of the line after a target or a dependent file name. For example:

```
MYSOURCE.EXE: FILE1.OBJ\
              FILE2.OBJ\
              FILE3.OBJ
     bcc file1.obj file2.obj file3.obj
```

Single targets with multiple rules

A single target can have more than one explicit rule. You must use the double colon :: after the target name to tell MAKE to expect multiple explicit rules. The following example shows how one target can have multiple rules and commands.

```
.cpp.obj:
   bcc -c -ncobj $<

.asm.obj:
   tasm  /mx $<, asmobj\\

mylib.lib :: f1.obj f2.obj
   echo Adding C files
   tlib mylib -+cobj\f1 -+cobj\f2
```

```
mylib.lib :: f3.obj f4.obj
   echo Adding ASM files
   tlib mylib -+asmobj\f3 -+asmobj\f4
```

Implicit rule syntax

An implicit rule starts with either a path or a period and *implies* a target-dependent file relationship. Its main components are file extensions separated by periods. The first extension belongs to the dependent, the second to the target.

If implicit dependents are out-of-date with respect to the target or if they don't exist, MAKE executes the commands associated with the rule. MAKE updates explicit dependents before it updates implicit dependents.

Implicit rules follow this basic syntax:

```
[{source_dirs}].source_ext[{target_dirs}].target_ext:
    [commands]
```

- {source_dirs} The directory of the dependent files. Separate multiple directories with a semicolon.
- .source_ext The dependent file-name extension.
- {target_dirs} The directory of the target (executable) files. Separate multiple directories with a semicolon.
- .target_ext The target file-name extension. Macros are allowed here.
- : Marks the end of the dependency line.
- commands Any operating system command. Multiple commands are allowed. Commands must be indented by one space or tab (see the section on commands on page 27).

If two implicit rules match a target extension but no dependent exists, MAKE uses the implicit rule whose dependent's extension appears first in the .SUFFIXES list. See the ".suffixes" section on page 36.

Explicit rules with implicit commands

A target in an explicit rule can get its command line from an implicit rule. The following example shows an implicit rule and an explicit rule without a command line.

```
.c.obj:
    bcc -c $<   #This command uses a macro $< described later.

myprog.obj:     #This explicit rule uses the command: bcc -c myprog.c
```

See page 31 for information on default macros.

The implicit rule command tells MAKE to compile MYPROG.C (the macro **$<** replaces the name `myprog.obj` with `myprog.c`).

Commands syntax

Commands can be any operating system command, but they can also include MAKE macros, directives, and special operators that operating systems can't recognize (note that | can't be used in commands). Here are some sample commands:

```
cd..
bcc -c mysource.c
COPY *.OBJ C:\PROJECTA
bcc -c $(SOURCE)      #Macros are explained later in the chapter.
```

Commands follow this general syntax:

```
[prefix...] commands
```

Command prefixes

Commands in both implicit and explicit rules can have prefixes that modify how MAKE treats the commands. Table 2.2 lists the prefixes you can use in makefiles; each prefix is explained in more detail following the table.

Table 2.2
Command prefixes

Option	Description
@	Don't display *command* while it's being executed.
–num	Stop processing commands in the makefile when the exit code returned from *command* exceeds *num*. Normally, MAKE aborts if the exit code is nonzero. No white space is allowed between – and *num*.
–	Continue processing commands in the makefile, regardless of the exit code returned by them.
&	Expand either the macro **$****, which represents all dependent files, or the macro **$?**, which represents all dependent files stamped later than the target. Execute the command once for each dependent file in the expanded macro.

Using @

The following command uses the modifier @, which prevents the command from displaying onscreen when MAKE executes it.

```
diff.exe : diff.obj
   @bcc diff.obj
```

Using –num and –

The **–num** and **–** modifiers control MAKE processing under error conditions. You can choose to continue with the MAKE process if an error occurs or only if the errors exceed a given number.

In the following example, MAKE continues processing if BCC isn't run successfully:

```
target.exe : target.obj
target.obj : target.cpp
  bcc -c target.cpp
```

Using &

The **&** modifier issues a command once for each dependent file. It is especially useful for commands that don't take a list of files as parameters. For example,

```
copyall : file1.cpp file2.cpp
   &copy $** c:\temp
```

results in COPY being invoked twice as follows:

```
copy file1.cpp c:\temp
copy file2.cpp c:\temp
```

Without the **&** modifier, COPY would be called only once.

Command operators

You can use any operating system command in a MAKE commands section. MAKE uses the normal operators (such as +, –, and so on), but it also has other operators you can use.

Table 2.3 Command operators

Operator	Description
<	Take the input for use by *command* from *file* rather than from standard input.
>	Send the output from *command* to *file*.
>>	Append the output from *command* to *file*.
<<	Create a temporary, inline file and use its contents as standard input to *command*.
&&	Create a temporary file and insert its name in the makefile.
delimiter	Any character other than # and \ used with **<<** and **&&** as a starting and ending delimiter for a temporary file. Any characters on the same line and immediately following the starting delimiter are ignored. The closing *delimiter* must be written on a line by itself.

Debugging with temporary files

Temporary files can help you debug a command set by placing the actual commands MAKE executes into the temporary file. Temporary file names start at MAKE0000.@@@, where the 0000 increments for each temporary file you keep. You must place delimiters after **&&** and at the end of what you want sent to the temporary file (! is a good delimiter).

The following example shows **&&** instructing MAKE to create a file of the input to TLINK.

```
prog.exe: A.obj B.obj
    TLINK /c &&!
    c0s.obj $**
    prog.exe
    prog.map
    maths.lib cs.lib
    !
```

The response file created by **&&** contains these instructions:

```
c0s.obj a.obj b.obj
prog.exe
prog.map
maths.lib cs.lib
```

Using MAKE macros

Macros are case-sensitive: MACRO1 is different from Macro1.

A MAKE macro is a string that is expanded (used) wherever the macro is called in a makefile. Macros let you create template makefiles that you can change to suit different projects. For example, to define a macro called LIBNAME that represents the string "mylib.lib," type `LIBNAME = mylib.lib`. When MAKE encounters the macro `$(LIBNAME)`, it uses the string `mylib.lib`.

If MAKE finds an undefined macro in a makefile, it looks for an operating-system environment variable of that name (usually defined with SET) and uses its definition as the expansion text. For example, if you wrote `$(path)` in a makefile and never defined *path*, MAKE would use the text you defined for PATH in your AUTOEXEC.BAT. (See the manuals for your operating system for information on defining environment variables.)

Defining macros

The general syntax for defining a macro in a makefile is `MacroName = expansion_text`.

- *MacroName* is case-sensitive and is limited to 512 characters.
- *expansion_text* is limited to 4096 characters consisting of alphanumeric characters, punctuation, and white space.

Each macro must be on a separate line in a makefile. Macros are usually put at the top of the makefile. If MAKE finds more than one definition for a *macroName*, the new definition replaces the old one.

Macros can also be defined using the command-line option **–D** (see page 21). More than one macro can be defined by separating them with spaces. The following examples show macros defined at the command line:

```
make -Dsourcedir=c:\projecta
make command="bcc -c"
make command=bcc option=-c
```

The following differences in syntax exist between macros entered on the command line and macros written in a makefile.

Table 2.4 Command line vs. makefile macros

Syntax	Makefile	Command line
Spaces allowed before and after =	Yes	No
Space allowed before *macroName*	No	Yes

Using a macro

To use a macro in a makefile, type `$(MacroName)` where MacroName is the name of a defined macro. You can use braces {} and parentheses () to enclose the MacroName.

MAKE expands macros at various times depending on where they appear in the makefile:

- Nested macros are expanded when the outer macro is invoked.
- Macros in rules and directives are expanded when MAKE first looks at the makefile.
- Macros in commands are expanded when the command is executed.

String substitutions in macros

MAKE lets you temporarily substitute characters in a previously defined macro. For example, if you defined a macro called SOURCE as `SOURCE = f1.cpp f2.cpp f3.cpp`, you could substitute the characters .OBJ for the characters .CPP by using `$(SOURCE:.CPP=.OBJ)`. The substitution doesn't redefine the macro.

Rules for macro substitution:

- Syntax: `$(MacroName:original_text=new_text)`.
- No whitespace before or after the colon.
- Characters in *original_text* must exactly match the characters in the macro definition; this text is case-sensitive.

MAKE now lets you use macros within substitution macros. For example,

```
MYEXT=.C
SOURCE=f1.cpp f2.cpp f3.cpp
$(SOURCE:.cpp=$(MYEXT))          #Changes f1.cpp to f1.C, etc.
```

Default MAKE macros

MAKE contains several default macros you can use in your makefiles. Table 2.5 lists the macro definition and what it expands to in explicit and implicit rules.

Table 2.5: Default macros

Macro	Expands in implicit:	Expands in explicit:	Example
$*	path\dependent file	path\target file	C:\PROJECTA\MYTARGET
$<	path\dependent file+ext	path\target file+ext	C:\PROJECTA\MYTARGET.OBJ
$:	path for dependents	path for target	C:\PROJECTA
$.	dependent file+ext	target file + ext	MYSOURCE.C
$&	dependent file	target file	MYSOURCE
$@	path\target file+ext	path\target file+ext	C:\PROJECTA\MYSOURCE.C
$**	path\dependent file+ext	all dependents file+ext	FILE1.CPP FILE2.CPP FILE3.CPP
$?	path\dependent file+ext	old dependents	FILE1.CPP

Table 2.6
Other default macros

Macro	Expands to:	Comment
__MSDOS__	1	If running under DOS.
__MAKE__	0x0370	MAKE's hex version number.
MAKE	make	MAKE's executable file name.
MAKEFLAGS	options	The options typed at the command line.
MAKEDIR	directory	Directory where MAKE.EXE is located.

Modifying default macros

When the default macros listed in Table 2.5 don't give you the exact string you want, macro modifiers let you extract parts of the string to suit your purpose.

To modify a default macro, use this syntax:

```
$(MacroName [modifier])
```

Table 2.7 lists macro modifiers and provides examples of their use.

Table 2.7
File-name macro modifiers

Modifier	Part of file name expanded	Example	Result
D	Drive and directory	$(<D)	C:\PROJECTA\
F	Base and extension	$(<F)	MYSOURCE.C
B	Base only	$(<B)	MYSOURCE
R	Drive, directory, and base	$(<R)	C:\PROJECTA\MYSOURCE

Using MAKE directives

MAKE directives resemble directives in languages such as C and Pascal, and perform various control functions, such as displaying commands onscreen before executing them. MAKE directives begin either with an exclamation point or a period. Table 2.8 lists MAKE directives and their corresponding command-line options (directives override command-line options). Each directive is described in more detail following the table.

Table 2.8
MAKE directives

Directive	Option	Description
.autodepend	–a	Turns on autodependency checking.
!elif		Acts like a C **else if**.
!else		Acts like a C **else**.
!endif		Ends an **!if**, **!ifdef**, or **!ifndef** statement.
!error		Stops MAKE and prints an error message.
!if		Begins a conditional statement.
!ifdef		If defined that acts like a C **ifdef**, but with macros rather than **#define** directives.
!ifndef		If not defined.
.ignore	–i	MAKE ignores the return value of a command.
!include		Specifies a file to include in the makefile.
!message		Lets you print a message from a makefile.
.noautodepend	–a-	Turns off autodependency checking.
.noIgnore	–i-	Turns off **.Ignore**.
.nosilent	–s-	Displays commands before MAKE executes them.
.path.ext		Tells MAKE to search for files with the extension .ext in *path* directories.

Table 2.8: MAKE directives (continued)

.precious		Saves the target or targets even if the build fails.
.silent	–s	Executes without printing the commands.
.suffixes		Determines the implicit rule for ambiguous dependencies.
!undef		Clears the definition of a macro.

.autodepend

Autodependencies occur in .OBJ files that have corresponding .CPP, .C, or .ASM files. With **.autodepend** on, MAKE compares the dates and times of all the files used to build the .OBJ. If the dates and times of the files used to build the .OBJ are different from the date-time stamp of the.OBJ file, the .OBJ file is recompiled. You can use **.autodepend** or **–a** in place of linked dependencies (see page 20 for information on linked dependencies).

!error

This is the syntax of the **!error** directive:

```
!error message
```

MAKE stops processing and prints the following string when it encounters this directive:

```
Fatal makefile exit code: Error directive: message
```

Embed **!error** in conditional statements to abort processing and print an error message, as shown in the following example:

```
!if !$d(MYMACRO)
#if MYMACRO isn't defined
!error MYMACRO isn't defined
!endif
```

If MYMACRO in the example isn't defined, MAKE prints the following message:

```
Fatal makefile 4: Error directive: MYMACRO isn't defined
```

Summing up error-checking controls

Four different controls turn off error checking:

- The **.ignore** directive turns off error checking for a selected portion of the makefile.
- The **–i** command-line option turns off error checking for the entire makefile.
- The ***–num*** command operator, which is entered as part of a rule, turns off error checking for the related command if the exit code exceeds the specified number.

- The – command operator turns off error checking for the related command regardless of the exit code.

!if and other conditional directives

The **!if** directive works like C **if** statements (see the *Programmer's Guide* if you don't understand **if** statements). As shown here, the syntax of **!if** and the other conditional directives resembles compiler conditionals:

```
!if condition      !if condition    !if condition      !ifdef macro
   :                  :                :                  :
!endif             !else            !elif condition    !endif
                      :                :
                   !endif           !endif
```

The following expressions are equivalent:

`!ifdef macro` and `!if $d(macro)`
`!ifndef macro` and `!if !$d(macro)`

These rules apply to conditional directives:

- One **!else** directive is allowed between **!if**, **!ifdef**, or **!ifndef** and **!endif** directives.
- Multiple **!elif** directives are allowed between **!if**, **!ifdef**, or **!ifndef** and **!else** directives and **!endif**.
- You can't split rules across conditional directives.
- You can nest conditional directives.
- **!if**, **!ifdef**, and **!ifndef** must have matching **!endif** directives within the same source file.

The following information can be included between **!if** and **!endif** directives:

- Macro definition
- Explicit rule
- Implicit rule
- **!include** directive
- **!error** directive
- **!undef** directive

Condition in **if** statements represents a conditional expression consisting of decimal, octal, or hexadecimal constants and the operators shown in Table 2.9.

Table 2.9 Conditional operators

Operator	Description	Operator	Description
–	Negation	?:	Conditional expression
~	Bit complement	!	Logical NOT
+	Addition	>>	Right shift

Table 2.9: Conditional operators (continued)

-	Subtraction	<<	Left shift
*	Multiplication	&	Bitwise AND
/	Division	|	Bitwise OR
%	Remainder	^	Bitwise XOR
&&	Logical AND	>=	Greater than or equal*
||	Logical OR	<=	Less than or equal*
>	Greater than	==	Equality*
<	Less than	!=	Inequality*

*Operator also works with string expressions.

MAKE evaluates a conditional expression as either a simple 32-bit signed integer or as a character string.

!include

This directive is like the **#include** preprocessor directive for the C or C++ language—it lets you include the text of another file in the makefile:

```
!include filename
```

You can enclose *filename* in quotation marks ("") or angle brackets (<>) and nest directives to unlimited depth, but writing duplicate **!include** directives in a makefile isn't permitted—you'll get the error message *cycle in the include file.*

Rules, commands, or directives must be complete within a single source file; you can't start a command in an **!include** file, then finish it in the makefile.

MAKE searches for **!include** files in the current directory unless you've specified another directory with the **-I** option.

!message

The **!message** directive lets you send messages to the screen from a makefile. You can use these messages to help debug a makefile that isn't working the way you'd like it to. For example, if you're having trouble with a macro definition, you could put this line in your makefile:

```
!message The macro is defined here as: $(MacroName)
```

When MAKE interprets this line, it will print onscreen `The macro is defined here as: .CPP`, if the macro expands to .CPP at that line. Using a series of **!message** directives, you can debug your makefiles.

.path.ext

The **.path.ext** directive tells MAKE where to look for files with a certain extension. The following example tells MAKE to look for files with the .c extension in C:\SOURCE or C:\CFILES and to look for files with the .obj extension in C:\OBJS.

```
.path.c = C:\CSOURCE;C:\CFILES
.path.obj = C:\OBJS
```

.precious

If a MAKE build fails, MAKE deletes the target file. The **.precious** directive prevents the file deletion, which is desired for certain kinds of targets such as libraries. When a build fails to add a module to a library, you don't want the library to be deleted.

The syntax for **.precious** is:

```
.precious: target [target] . . . [target]
```

.suffixes

The **.suffixes** directive tells MAKE the order (by file extensions) for building implicit rules.

The syntax of the **.suffixes** directive is:

```
.suffixes: .ext [.ext]  [.ext] . . . [.ext]
```

.ext represents the dependent file extension in implicit rules. For example, you could include the line `.suffixes: .asm .c .cpp` to tell MAKE to interpret implicit rules beginning with the ones dependent on .ASM files, then .C files, then .CPP files, regardless of what order they appear in the makefile.

The following example shows a makefile containing a **.suffixes** directive that tells MAKE to look for a source file (MYPROG.EXE) first with an .ASM extension, next with a .C extension, and finally with a .CPP extension. If MAKE finds MYPROG.ASM, it builds MYPROG.OBJ from the assembler file by calling TASM. MAKE then calls TLINK; otherwise, MAKE searches for MYPROG.C to build the .OBJ file, and so on.

```
.suffixes: .asm .c .cpp

myprog.exe: myprog.obj
tlink myprog.obj

.cpp.obj:
  bcc -P $<
.asm.obj:
  tasm /mx $<
.c.obj:
  bcc -P- $<
```

!undef

The syntax of the **!undef** directive is:

```
!undef MacroName
```

!undef (undefine) clears the given macro, *MacroName*, causing an **!ifdef** *MacroName* test to fail.

Using macros in directives

The macro **$d** is used with the **!if** conditional directive to perform some processing if a specific macro is defined. The **$d** is followed by a macro name, enclosed in parentheses or braces, as shown in the following example.

```
!if $d(DEBUG)              #If DEBUG is defined,
bcc -v f1.cpp f2.cpp       #compile with debug information;
!else                      #otherwise (else)
bcc -v- f1.cpp f2.cpp      #don't include debug information.
!endif
```

Caution! Don't use the **$d** macro when MAKE is invoked with the **–N** option.

Null macros

An undefined macro causes an **!ifdef** *MacroName* test to return false; a null macro returns true. A null macro is a macro defined with either spaces to the right of the equal sign (=) or no characters to the right of the equal sign. For example, the following line defines a null macro in a makefile:

```
NULLMACRO =
```

One of the following lines can define a null macro on the MAKE command line:

```
NULLMACRO=""
```

or

```
-DNULLMACRO
```

C H A P T E R 3

TLIB: The Turbo librarian

TLIB is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension.

The libraries included with Borland C++ were built with TLIB. You can use TLIB to build your own libraries, or to modify the Borland C++ libraries, your own libraries, libraries furnished by other programmers, or commercial libraries you've purchased. You can use TLIB to

- Create a new library from a group of object modules.
- Add object modules or other libraries to an existing library.
- Remove object modules from an existing library.
- Replace object modules from an existing library.
- Extract object modules from an existing library.
- List the contents of a new or existing library.

Although TLIB is not essential for creating executable programs with Borland C++, it is a useful programming productivity tool. You will find TLIB indispensable for large development projects. If you work with object module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

Why use object module libraries?

When you program in C and C++, you often create a collection of useful functions and classes. Because of C and C++'s modularity, you are likely to split those functions into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. On the other hand, if you always include all the source files, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of functions and classes. When you link your program with a library, the

linker scans the library and automatically selects only those modules needed for the current program.

The TLIB command line

To get a summary of TLIB's usage, just type `TLIB` and press *Enter*.

The TLIB command line takes the following general form, where items listed in square brackets ([*like this*]) are optional:

```
tlib [/C] [/E] [/Psize] libname [operations] [, listfile]
```

Table 3.1 TLIB options

Option	Description
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a *libname*. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both BCC and BC's project-make facility require the .LIB extension in order to recognize library files. **Note:** If the named library does not exist and there are *add* operations, TLIB creates the library.
/C	The case-sensitive flag. This option is not normally used; see page 43 for a detailed explanation.
/P*size*	Sets the library page size to *size*; see page 42 for a detailed explanation.
operations	The list of operations TLIB performs. Operations can appear in any order. If you only want to examine the contents of the library, don't give any operations.
listfile	The name of the file listing library contents. The *listfile* name (if given) must be preceded by a comma. No listing is produced if you don't give a file name. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the *listfile* is .LST. You can direct the listing to the screen by using the *listfile* name CON, or to the printer by using the name PRN.

For TLIB examples, refer to the "Examples" section on page 43.

This section summarizes each of these command-line components; the following sections provide details about using TLIB.

The operation list

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character action symbol followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to the OS/2 CMD.EXE-imposed line-length limit of 256 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

You can replace a module by first removing it, then adding the replacement module.

File and module names

TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you need to supply only the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB operations

TLIB recognizes three action symbols (–, +, *), which you can use singly or combined in pairs for a total of five distinct operations. The order of the characters is not important for operations that use a pair of characters. The action symbols and what they do are listed here:

Table 3.2
TLIB action symbols

Action symbol	Name	Description
+	**Add**	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	**Remove**	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation needs only a module name. TLIB lets you enter a full path name with drive and extension included, but ignores everything except the module name.
*	**Extract**	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.

To create a library, add modules to a library that does not yet exist.

Table 3.2: TLIB action symbols (continued)

-* *-	**Extract & Remove**	TLIB copies the named module to the corresponding file name and then removes it from the library. This is just shorthand for an *extract* followed by a *remove* operation.
-+ +-	**Replace**	TLIB replaces the named module with the corresponding file. This is just shorthand for a *remove* followed by an *add* operation.

You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.

Using response files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using response files. A response file is simply an ASCII text file (which can be created with the Borland C++ editor) that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one OS/2 command line.

See "Examples" on page 43 for a sample response file and a TLIB command line incorporating it.

To use a response file path name, specify `@pathname` at any position on the TLIB command line.

- More than one line of text can make up a response file; you use the "and" character (**&**) at the end of a line to indicate that another line follows.
- You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest on the command line itself.
- You can use more than one response file in a single TLIB command line.

Setting the page size: The /P option

Every DOS library file contains a dictionary, which appears at the end of the .LIB file, following all of the object modules. For each module in the library, this dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library—it cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, the page size must be increased using the **/P** option; the page size must be a power of 2, and it cannot be smaller than 16, or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), an average of 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use **/P** with the next available higher page size.

Advanced operation: The /C option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module to the library that would cause a duplicate symbol, TLIB displays a message and won't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates. Since C and C++ treat uppercase and lowercase letters as distinct, use the **/C** option to add a module to a library that includes a symbol differing only in case from one already in the library. The **/C** option tells TLIB to accept a module with symbols in it that differ only in case from symbols already in the library.

If you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should **not** use the **/C** option.

It may seem odd that, without the **/C** option, TLIB rejects symbols that differ only in case, especially since C and C++ are case-sensitive languages. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case. Such linkers treat *stars*, *Stars*, and *STARS* as the same identifier. TLINK, on the other hand, has no problem distinguishing uppercase and lowercase symbols, and it properly accepts a library containing symbols that differ only in case. In this example, then, Borland C++ would treat *stars*, *Stars*, and *STARS* as three separate identifiers. As long as you use the library only with TLINK, you can use the TLIB **/C** option without any problems.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB:

- To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

```
tlib mylib +x +y +z
```

- To create the same library as in #1 and get a listing in MYLIB.LST too, type

```
tlib mylib +x +y +z, mylib.lst
```

- To get a listing in CS.LST of an existing library CS.LIB, type

```
tlib cs, cs.lst
```

- To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

```
tlib mylib -+x +a -z
```

- To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type

```
tlib mylib *y, mylib.lst
```

- To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:
  - First create a text file, ALPHA.RSP, with

```
+a.obj +b.obj +c.obj &
       +d.obj +e.obj +f.obj &
       +g.obj
```

  - Then use the TLIB command, which produces a listing file named ALPHA.LST:

```
tlib alpha @alpha.rsp, alpha.lst
```

C H A P T E R 4

Import library tools

This chapter describes IMPLIB and IMPDEF. IMPLIB creates import libraries, and IMPDEF creates module definition files (.DEF files). Import libraries and module definition files provide information to the linker about functions imported from dynamic-link libraries (DLLs).

Dynamic-link libraries are an important part of OS/2 programming. Using DLL functions in your application is called *importing* DLL functions. There are two ways to import a DLL function: you can use import libraries, or you can use the IMPORTS section of module definition files.

When linking an OS/2 application, TLINK uses import libraries and module definition files to know when a function is defined in and imported from a DLL. Import libraries generally replace the IMPORTS section of module definition files.

IMPLIB creates an import library for a DLL. IMPDEF creates a module definition file that has an EXPORTS statement containing the name of each exported function in a DLL. See the *Programmer's Guide*, Chapter 9, for an explanation of module definition files.

IMPLIB: The import librarian

The IMPLIB utility creates import libraries. IMPLIB takes as input DLLs, module definition files, or both, and produces an import library as output.

Import libraries contain records. Each record contains the name of a DLL, and specifies where in the DLL the imported functions reside. These records are bound to the application by TLINK or the IDE linker, and provide OS/2 with the information necessary to resolve DLL function calls. An import library can be substituted for part or all of the IMPORTS section of a module definition file.

If you've created an OS/2 application, you've already used at least one import library, OS2.LIB, the import library for the standard OS/2 DLLs. These DLLs contain the OS/2 system calls. (OS2.LIB is linked automatically when you build an OS/2 application in the IDE and when using BCC to

link. You have to explicitly link with OS2.LIB only if you're using TLINK to link separately.)

See page 47 for information on using IMPDEF and IMPLIB to customize an import library for a specific application.

An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).

To create an import library for a DLL, type

```
IMPLIB Options LibName [ DefFiles... | DLLs... ]
```

Note that a DLL can also have an extension of .EXE or .DRV, not just .DLL.

where *Options* is an optional list of one or more IMPLIB options (see Table 4.1), *LibName* (required) is the name for the new import library, *DefFiles* is a list of one or more existing module definition files for one or more DLLs, and *DLLs* is a list of one or more existing DLLs. You must specify at least one DLL or module definition file.

Table 4.1 IMPLIB options

You can use either a hyphen or a slash to precede IMPLIB's options, but the options must be lowercase.

Option	What it does
-t	Terse warnings.
-v	Verbose warnings.
-w	No warnings.

IMPDEF: The module definitions file manager

An import library provides access to the functions in a Windows DLL. See page 45 for more details.

IMPDEF takes as input a DLL name, and produces as output a module definition file with an export section containing the names of functions exported by the DLL. The syntax is

```
IMPDEF DestName.DEF SourceName.DLL
```

This creates a module definition file named *DestName*.DEF from the file *SourceName*.DLL. The module definition file would look something like this:

```
LIBRARY     FileName
DESCRIPTION 'Description'
EXPORTS
            ExportFuncName          @Ordinal
               ⋮
            ExportFuncName          @Ordinal
```

where *FileName* is the DLL's root file name, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement,

ExportFuncName names an exported function, and *Ordinal* is that function's ordinal value (an integer).

Classes in a DLL

IMPDEF is useful for a DLL that uses C++ classes. If you use the **_export** keyword when defining a class, all of the non-inline member functions and static data members for that class are exported. It's easier to let IMPDEF make a module definition file for you because it lists all the exported functions, automatically including the member functions and static data members.

Since the names of these functions are mangled, it would be tedious to list them all in the EXPORTS section of a module definition file simply to create an import library from the module definition file. If you use IMPDEF to create the module definition file, it will include the ordinal value for each exported function. If the exported name is mangled, IMPDEF will also include that function's unmangled, original name as a comment following the function entry. So, for instance, the module definition file created by IMPDEF for a DLL that used C++ classes would look something like this:

```
LIBRARY     FileName
DESCRIPTION 'Description'
EXPORTS
            MangledExportFuncName  @Ordinal ; ExportFuncName
               ⋮
            MangledExportFuncName  @Ordinal ; ExportFuncName
```

where *FileName* is the DLL's root file name, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement, *MangledExportFuncName* provides the mangled name, *Ordinal* is that function's ordinal value (an integer), and *ExportFuncName* gives the function's original name.

Functions in a DLL

IMPDEF creates an editable source file that lists all the exported functions in the DLL. You can edit this .DEF file to contain only those functions that you want to make available to a particular application, then run IMPLIB on the edited .DEF file. This results in an import library that contains import information for a specific subset of a DLL's export functions.

Suppose you're distributing a DLL that provides functions to be used by several applications. Every export function in the DLL is defined with **_export**. Now, if all the applications used all the DLL's exports, then you could simply use IMPLIB to make one import library for the DLL. You could deliver that import library with the DLL, and it would provide

import information for all of the DLL's exports. The import library could be linked to any application, thus eliminating the need for the particular application to list every DLL function it uses in the IMPORTS section of its module definition file.

But let's say you want to give only a few of the DLL's exports to a particular application. Ideally, you want a customized import library to be linked to that application—an import library that provides import information only for the subset of functions that the application will use. All of the other export functions in the DLL will be hidden to that client application.

To create an import library that satisfies these conditions, run IMPDEF on the compiled and linked DLL. IMPDEF produces a module definition file that contains an EXPORT section listing all of the DLL's export functions. You can edit that module definition file, removing EXPORTS section entries for those functions you don't want in the customized import library. Once you've removed the exports you don't want, run IMPLIB on the module definition file. The result will be an import library that contains import information for only those export functions listed in the EXPORTS section of the module definition file.

CHAPTER 5

Resource tools

This chapter describes the Borland resource tools.

- BRCC.EXE is the Borland resource compiler. It compiles resource script files (.RC files) and produces the binary .RES file.
- RC.EXE is the OS/2 resource compiler. It compiles .RC files, and links .RES files to an executable.

Presentation Manager (PM) programs are easy to learn and use because they provide a familiar and standard user interface for all applications. The components of the user interface are known as resources. Resources are defined external to your source code, then attached to the executable that uses them.

Resources describe PM user-interface devices such as

For more information on resources, see the Resource Workshop *User's Guide.*

- Menus
- Dialog boxes
- Pointers
- Icons
- Bitmaps
- Strings
- Accelerator keys

These resources are joined with an executable file before the application is run, but the application calls resources into memory only when needed, minimizing memory usage.

Resource script (.RC) files are text files that describe the resources a particular application will use. BRCC or RC uses the .RC file to compile the resources into a binary format resource (.RES) file. RC then attaches the .RES file, containing your resources, to your executable. Attaching .RES files to the .EXE file is commonly called resource linking.

BRCC.EXE: The resource compiler

BRCC is a command-line version of Resource Workshop's resource compiler. It accepts a resource script file (.RC) as input and produces a resource object file (.RES) as output. BRCC uses the following command-line syntax:

```
BRCC [options] <filename>.RC
```

Table 5.1 lists all BRCC switches.

Table 5.1
BRCC (Borland resource compiler)

Switch	Description
–d<name>[=<string>]	Defines a preprocessor symbol.
–fo<filename>	Renames the output .RES file. (By default, BRCC creates the output .RES file with the same name as the input .RC file.)
–i<path>	Adds one or more directories (separated by semicolons) to the include search path.
–k<value>	Sets codepage to value.
–r	This switch is ignored. It is included for compatibility with other resource compilers.
–v	Prints progress messages (verbose).
–x	Deletes the current include path.
–? or –h	Displays switch help.

Like Resource Workshop's resource compiler, BRCC predefines common resource-related PM constants such as WS_VISIBLE and BS_PUSHBUTTON. Also, two special compiler-related symbols are defined: RC_INVOKED and WORKSHOP_INVOKED. These symbols can be used in the source text in conjunction with conditional preprocessor statements to control compilation. For example, the following construct can greatly speed up compilation:

```
#ifndef WORKSHOP_INVOKED
#include "os2.h"
#endif
```

Examples

The following example adds two directories to the include path and produces a .RES file with the same name as the input .RC file.

```
brcc -i<dir1>;<dir2> <filename>.RC
```

This example enables you to produce an output .RES file with a name different from the input .RC file's.

```
brcc -fo<filename>.RES <filename>.RC
```

RC.EXE: The OS/2 resource compiler

See Table 5.2 for a description of the Resource Compiler options.

The PM resource compiler (RC.EXE) will both compile and link your resources. Here is the syntax for invoking RC.EXE from the command line:

```
RC [options] ResourceFile [ModuleFile]
```

For example, to compile MYAPP.RC file and add it to MYAPP.EXE, you would give this command line:

```
rc myapp
```

This simplest form only works if the resource file and the executable file share the same name. If MYAPP.RC was instead named MYAPPRS.RC, you would type

```
rc myapprs myapp
```

To compile only the MYAPP.RC resource file (and not add the resulting MYAPP.RES to MYAPP.EXE), use the **–R** option, like this:

```
rc -r myapp
```

You would then have a MYAPP.RES file. To add MYAPP.RES to MYAPP.EXE, type

```
rc myapp.res
```

To mark a module as PM-compatible, but not add any resources to it, simply invoke the Resource Compiler with the module name (note that the file name must have one of these extensions: .EXE, .DLL, or .DRV). For example,

```
rc myapp.exe
```

The following table describes the Resource Compiler options. Note that Resource Compiler options are not case sensitive (**-e** is the same as **–E**). Also, options that take no arguments can be combined (for instance, **–kpr** is legal).

Table 5.2: Resource Compiler options

Option	What it does
–?	Lists help on Resource Compiler options (also **–H**).
–d *Symbol*	Defines *Symbol* for the preprocessor.
–h	Lists help on Resource Compiler options (also **–?**).
–i *Path*	After searching the current directory for include files and resource files, RC searches the directory named in *Path*. The **–i** option can be repeated if you want to specify more than one search path. Also see the description for the **–x** option.
–p	Packs resources; resources will not cross 64K boundaries.
–r	Compiles the .RC file into a .RES file, but does not add it to an .EXE.

APPENDIX A

Error messages

This appendix describes the error messages that can be generated by Borland C++. The error messages in this appendix include messages that can be generated by the compiler, the MAKE utility, the librarian (TLIB), and the linker (TLINK). This appendix also lists the errors that you can receive when you run your program (run-time errors).

Messages are listed in ASCII alphabetic order. Messages beginning with symbols come first, then messages beginning with numbers, and then messages beginning with letters of the alphabet. Messages that begin with symbols are alphabetized by the first word in the message that follows the symbols. For example, you might receive the following error message if you incorrectly declared your function *my_func*:

```
my_func must be declared with no parameters
```

To find this error message, look under the alphabetized listing of "must."

Message classes

Messages fall into three categories: fatal errors, errors, and warnings.

Fatal errors

Fatal errors can be generated by the compiler, the linker, and the MAKE utility. Fatal errors cause the compilation to stop immediately; you must take appropriate action to fix the error before you can resume compiling.

If the compiler or MAKE utility issues a fatal error, no .EXE file is created. If the linker issues a fatal error, any .EXE file that might have been created by the linker is deleted before the linker returns.

Errors

Errors can be generated by the compiler, the linker, the MAKE utility, and the librarian. In addition, errors can be generated by your program at run time.

Errors generated by the compiler indicate program syntax errors, command-line errors, and disk or memory access errors. Compiler errors

don't cause the compilation to stop—the compiler completes the current phase of the compilation and then stops and reports the errors encountered. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing, and code-generating).

Errors generated by the linker don't cause the linker to delete the .EXE or .MAP files. However, you shouldn't execute any .EXE file that was linked with errors. Linker errors are treated like fatal errors if you're compiling from the Integrated Development Environment (IDE).

The MAKE utility generates errors when there is a syntax or semantic error in the source makefile. You must edit the makefile to fix these types of errors.

Run-time errors are usually caused by logic errors in your program code. If you receive a run-time error, you must fix the error in your source code and recompile the program for the fix to take effect.

Warnings

Warnings can be issued by the compiler, the linker, and the librarian. Warnings do not prevent the compilation from finishing. However, they do indicate conditions that are suspicious, even if the condition that caused the warning is legitimate within the language. The compiler also produces warnings if you use machine-dependent constructs in your source files.

Message listings

Messages are written with the message class first, followed by the source file name and line number where the error was detected, and finally with the text of the message itself.

Some messages include a symbol (such as a variable, file name, or module) that is taken from your program. Symbols in the message explanations are shown in *italics* to indicate that they're variable in nature.

Be aware that the compiler generates messages as they are detected. Because C and C++ don't force any restrictions on placing statements on a line of text, the true cause of the error might be one or more lines before or after the line number mentioned in the error message.

Message explanations

')' missing in macro invocation *MAKE error*
A left parenthesis is required to invoke a macro.

(expected *Compiler error*
A left parenthesis was expected before a parameter list.

) expected *Compiler error*
A right parenthesis was expected at the end of a parameter list.

, expected *Compiler error*
A comma was expected in a list of declarations, initializations, or parameters.

: expected after private/protected/public *Compiler error*
When used to begin a **private/protected/public** section of a C++ class, these reserved words must be followed by a colon.

< expected *Compiler error*
The keyword **template** was not followed by a left angle bracket (<). Every template declaration must include the template formal parameters enclosed within angle brackets (< >), immediately following the **template** keyword.

> expected *Compiler error*
A new-style cast (for example, dynamic_cast) is missing a closing ">".

@ seen, expected a response-file name *Librarian error*
The response file is not given immediately after @.

{ expected *Compiler error*
A left brace ({) was expected at the start of a block or initialization.

} expected *Compiler error*
A right brace (}) was expected at the end of a block or initialization.

16-bit segments not supported in module *module* *Linker error*
16-bit segments aren't supported in 32-bit applications. Check to make sure that you haven't inadvertently compiled your 32-bit application using the 16-bit compiler.

286/287 instructions not enabled *Compiler error*
Use the **–2** command-line compiler option or the 80286 options from the Options|Compiler|Code Generation|Advanced Code Generation dialog box to enable 286/287 opcodes. The resulting code cannot be run on 8086- and 8088-based machines.

32-bit record encountered *Linker error*
An object file that contains 80386 32-bit records was encountered, and the **/3** option had not been used.

Abnormal program termination *Run-time error*
The program called *abort* because there wasn't enough memory to execute. This can happen as a result of memory overwrites.

Access can only be changed to public or protected *Compiler error*
A C++ derived class can modify the access rights of a base class member, but only to **public** or **protected**. A base class member cannot be made **private**.

Added file *filename* does not begin correctly, ignored *Librarian warning*

The librarian has decided that the file being added is not an object module, so it will not try to add it to the library. The library is created anyway.

Address of overloaded function *function* doesn't match *type* *Compiler error*

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.

Alias *alias* defined in module 'module' is redefined *Linker warning*

An ALIAS definition record specified a public symbol substitute that was already defined by another ALIAS. The second public symbol substitute will be used. ALIAS records are generated by the assembler when the ALIAS directive is used.

***module* already in LIB, not changed!** *Librarian warning*

An attempt to use the + action on the library has been made, but there is already an object with the same name in the library. If an update of the module is desired, the action should be +-. The library has not been modified.

Ambiguity between *function1* and *function2* *Compiler error*

Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.

Ambiguous member name *name* *Compiler error*

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.

Ambiguous operators need parentheses *Compiler warning*

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.

Ambiguous override of virtual base member *base_function*: *derived_function* *Compiler error*

This error message is issued when a virtual function that is defined in a virtual base class is overridden with different functions having two derived classes in the same inheritance hierarchy. For example,

```
struct VB
{
   virtual f();
};

struct A:virtual VB
{
   virtual f();
};

struct B:virtual VB
   virtual f();
};

struct D:A,B
{
}      //errors here
```

The above code will be flagged with the following errors:

```
Error: Ambiguous override of virtual base member VB::f():A::f()
Error: Ambiguous override of virtual base member VB::f():B::f()
```

Array allocated using 'new' may not have an initializer *Compiler error*

When initializing a vector (array) of classes, you must use the *default constructor* (the constructor that has no arguments).

Array bounds missing] *Compiler error*

Your source file declared an array in which the array bounds were not terminated by a right bracket.

Array must have at least one element *Compiler error*

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.

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 */
```

Array of references is not allowed *Compiler error*

It is illegal to have an array of references because pointers to references are not allowed and array names are coerced into pointers.

Array size for 'delete' ignored *Compiler warning*

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, Borland C++ ignores this construct and issues this warning.

Assembler stack overflow *Compiler error*

The assembler ran out of memory during compilation. Review the portion of code flagged by the error message to ensure that it uses memory correctly.

Assembler statement too long *Compiler error*

Inline assembly statements cannot be longer than 480 bytes.

Assigning *type* to *enumeration* *Compiler warning*

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.

Assignment to 'this' not allowed, use X::operator new instead *Compiler error*

In early versions of C++, the only way to control allocation of a class of objects was to use the **this** parameter inside a constructor. This practice is no longer allowed because a better, safer, and more general technique is to instead define a member function **operator new**.

Attempt to export non-public symbol *symbol* *Linker error*

This error usually occurs when a .DEF file specifies an EXPORT for a symbol that you either forgot to define or misspelled.

Attempt to grant or reduce access to *identifier* *Compiler error*

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.

Attempting to return a reference to a local object *Compiler error*

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). Because this object will disappear when the function returns, the reference will then be illegal.

Attempting to return a reference to local variable *identifier* *Compiler error*

This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal because the variable referred to disappears when the function exits. You can return a reference to any static or global variable, or you can change the function to return a value instead.

Bad call of intrinsic function *Compiler error*

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.

Bad character in parameters –> *char* *Linker error*

One of the following characters (or any control character other than horizontal tab, linefeed, carriage return, or *Ctrl+Z*) was encountered in the command line or in a response file:

" * < = > ? [ ] |

Bad define directive syntax *Compiler error*

A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.

Bad field list in debug information in module *module* *Linker error*

This is typically caused by bad debug information in the OBJ file. Borland Technical Support should be informed.

Bad file name *filename* *Linker error*

An invalid file name was passed to the linker.

Bad file name format in include directive *Compiler error*

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, it is not surrounded by < > or " ".

Bad filename format in include statement *MAKE error*

Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.

Bad file name format in line directive *Compiler error*

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, it is not surrounded by quote marks.

Bad header in input LIB *Librarian error*

When adding object modules to an existing library, the librarian has determined that it has a bad library header. Rebuild the library.

Bad ifdef directive syntax *Compiler error*

An **#ifdef** directive must contain a single identifier (and nothing else) as the body of the directive.

Bad LF_POINTER in module *module* *Linker error*

This is typically caused by bad debug information in the OBJ file. Borland Technical Support should be informed.

Bad macro output translator *MAKE error*

Invalid syntax for substitution within macros.

Bad object file *filename* near file offset *offset* *Linker error*

The linker has found a bad OBJ file. This is usually caused by a translator error.

Bad object file record *Linker error*

Bad object file *file* near file offset *offset* *Linker error*

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+Break* was pressed.

Bad OMF record type *type* encountered in module *module* *Librarian error*

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 re-created.

Bad syntax for pure function definition *Compiler error*

Pure virtual functions are specified by appending "= 0" to the declaration. You wrote something similar, but not quite the same.

Bad undef directive syntax *Compiler error*

An **#undef** directive must contain a single identifier (and nothing else) as the body of the directive.

Bad undef statement syntax *MAKE error*

An **!undef** statement must contain a single identifier and nothing else as the body of the statement.

Bad version number in parameter block *Linker error*

This error indicates an internal inconsistency in the IDE. If it occurs, exit and restart the IDE. This error does not occur in the standalone version.

Base class *class* contains dynamically dispatchable functions *Compiler error*

Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance. This error occurs because a class that contains DDVT functions attempted to inherit DDVT functions from multiple parent classes.

Base class *class* is also a base class of *class* *Compiler warning*

A class inherits from the same base class both directly and indirectly. It is best to avoid this non-portable construct in your program code.

Base class *class* is included more than once *Compiler error*

A C++ class can be derived from any number of base classes, but can be directly derived from a given class only once.

Base class *class* is initialized more than once *Compiler error*

In a C++ class constructor, the list of initializations following the constructor header includes base class *class* more than once.

Base initialization without a class name is now obsolete *Compiler error*

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 that you 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) { ... }
```

Bit field cannot be static *Compiler error*

Only ordinary C++ class data members can be declared **static**, not bit fields.

Bit field too large *Compiler error*

This error occurs when you supply a bit field with more than 32 bits.

Bit fields must be signed or unsigned int *Compiler error*

In ANSI C, bit fields can only be **signed** or **unsigned int** (not **char** or **long**, for example).

Bit fields must be signed or unsigned int *Compiler warning*

In ANSI C, bit fields cannot be of type **signed char** or **unsigned char**. When not compiling in strict ANSI mode, the compiler allows such constructs, but flags them with this warning.

Bit fields must contain at least one bit *Compiler error*

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 Word Alignment). In C++, bit fields must have an integral type; this includes enumerations.

Bit fields must have integral type *Compiler error*

In C++, bit fields must have an integral type; this includes enumerations.

Body has already been defined for function *function* *Compiler error*

A function with this name and type was previously supplied a function body. A function body can be supplied only once.

Both return and return with a value used *Compiler warning*

The current function has **return** statements with and without values. This is legal in C, but is almost always an error. Possibly a **return** statement was omitted from the end of the function.

Call of nonfunction *Compiler error*

The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or by misspelling the function name.

Call to function *function* with no prototype *Compiler warning*

The "Prototypes required" warning was enabled and you called function *function* without first giving a prototype for it.

Call to undefined function *function* *Compiler error*

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. **int** functions are exempt in C because in old versions of C there was no **void** type to indicate functions that return nothing.

Cannot access an inactive scope *IDE debugger error*

You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

***virtual* can only be used with member functions** *Compiler error*

A data member has been declared with the virtual specifier. Only member functions can be declared virtual.

Can't grow LE/LIDATA record buffer *Librarian error*

Command-line error. See the **out of memory reading LE/LIDATA record from object module** message.

Can't inherit non-RTTI class from RTTI base *class* *Compiler error*
Can't inherit RTTI class from non-RTTI base *class* *Compiler error*

When virtual functions are present, the RTTI attribute of all base classes must match that of the derived class.

Cannot access an inactive scope *Compiler error*

You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

Cannot add or subtract relocatable symbols *Compiler error*

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.

Cannot allocate a reference *Compiler error*

An attempt to create a reference using the **new** operator has been made; this is illegal because references are not objects and cannot be created through **new**.

***identifier* cannot be declared in an anonymous union** *Compiler error*

The compiler found a declaration for a member function or static member in an anonymous union. Such unions can contain data members only.

function1* cannot be distinguished from *function2 *Compiler error*

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.

Cannot generate *function* from template function *template* *Compiler error*

A call to a template function was found, but a matching template function cannot be generated from the function template.

Cannot call *main* from within the program *Compiler error*

C++ does not allow recursive calls of the function *main*.

Cannot call near class member function with a pointer of type *type* *Compiler error*

Member functions of near classes (classes are near by default) cannot be called using far or huge member pointers. This also applies to calls using pointers to members. Either change the pointer to be **__near**, or declare the class as **__far**.

Cannot cast from *type1* to *type2* *Compiler error*

A cast from type *type1* to type *type2* is not allowed. In C, a pointer can be cast to an integral type or to another pointer. An integral type can be cast to any integral, floating, or pointer type. A floating type can be cast to an integral or floating type. Structures and arrays cannot 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 can be cast to a member pointer. A member pointer can 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.

Cannot convert *type1* to *type2* *Compiler error*

An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal.

Cannot create instance of abstract class *class* *Compiler error*

Abstract classes—those with pure virtual functions—cannot be used directly, only derived from.

Cannot create pre-compiled header: code in header *Compiler warning*

One of the headers contained a non-inline function body.

Cannot create pre-compiled header: initialized data in header *Compiler warning*

One of the headers contained a global variable definition. In a C header, this message indicates that a global variable was initialized. In C++ header, this message indicates that a variable was not declared "extern."

Cannot create pre-compiled header: header incomplete *Compiler warning*

The pre-compiled header ended in the middle of a declaration. This often happens when there is a missing "}" in a class definition that is located in a header file.

Cannot create pre-compiled header: write failed *Compiler warning*

The compiler could not write to the pre-compiled header file. This is usually due to a full disk or a disk that is write protected.

Cannot define a pointer or reference to a reference *Compiler error*

It is illegal to have a pointer to a reference or a reference to a reference.

Cannot find *class*::*class*(*class* &) to copy a vector *Compiler error*

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, called a *copy constructor*, takes just one parameter (a reference to its class).

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.

Cannot find *class*::operator=(*class*&) to copy a vector *Compiler error*

When a C++ class *class1* contains a vector (array) of class *class2*, and you want 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.

Cannot find default constructor to initialize array element of type *class* *Compiler error*

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.

Cannot find default constructor to initialize base class *class* *Compiler error*

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.

Cannot find default *constructor* to initialize member *identifier* *Compiler error*

When a C++ class *class1* contains a member of class *class2*, and you want 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 supplies a default constructor

automatically unless you've defined a constructor for class *class2*. If you have, the compiler won't supply the default constructor automatically—you must supply one.

Cannot find MAKE.EXE *MAKE error*

The MAKE command-line tool cannot be found. Be sure that MAKE.EXE is in either the current directory or in a directory contained in your directory path.

Cannot generate *function* from template function *template* *Compiler error*

A call to a template function was found, but a matching template function cannot be generated from the function template.

Cannot have a non-inline function in a local class *Compiler error*
Cannot have a static data member in a local class *Compiler error*

All members of classes declared local to a function must be entirely defined in the class definition. This means that such local classes cannot contain any static data members, and all of their member functions must have bodies defined within the class definition.

Cannot initialize a class member here *Compiler error*

Individual members of **struct**s, **union**s, and C++ **class**es cannot have initializers. A **struct** or **union** can be initialized as a whole using initializers inside braces. A C++ **class** can be initialized only by the use of a constructor.

Cannot initialize *type1* with *type2* *Compiler error*

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.

Cannot modify a const object *Compiler error*

This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.

Cannot overload 'main' *Compiler error*

main is the only function that cannot be overloaded.

***function* cannot return a value** *Compiler error*

A function with a return type **void** contains a **return** statement that returns a value; for example, an **int**.

***identifier* cannot start a parameter declaration** *Compiler error*

An undefined 'identifier' was found at the start of an argument in a function declarator. This error usually occurs because the wrong header file was used. If that isn't the cause, check to see if the type name is misspelled or if the type declaration is missing.

Cannot take address of *main* *Compiler error*

In C++ it is illegal to take the address of the main function.

Cannot throw *type* — ambiguous base class *base* *Compiler error*

It is not legal to throw a class that contains more than one copy of a (nonvirtual) base class.

Cannot write a string option *MAKE error*

the **–W** MAKE option writes a character option to MAKE.EXE. If there's any string option (for example, **–D**xxxx="My_foo" or **–U**xxxxx), this error message is generated.

Case bypasses initialization of a local variable *Compiler error*

In C++ it is illegal to bypass the initialization of a local variable in any way. In this instance, there is a **case** label that can transfer control past this local variable.

Case outside of switch *Compiler error*

The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched braces.

Case statement missing : *Compiler error*

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.

***catch* expected** *Compiler error*

In a C++ program, a *try* block must be followed by at least one *catch* block.

Character constant must be one or two characters long *Compiler error*

Character constants can be only one or two characters long.

Character constant too long *MAKE error*

A char constant in an expression is too long.

Circular dependency exists in makefile *MAKE error*

The makefile indicates that a file needs to be up-to-date *before* it can be built. Take, for example, the explicit rules:

```
filea: fileb
fileb: filec
filec: filea
```

This implies that file*a* depends on file*b*, which depends on file*c*, and file*c* depends on file*a*. This is illegal because a file cannot depend on itself, indirectly or directly.

Class *class* may not contain pure functions *Compiler error*

The class being declared cannot be abstract; it therefore cannot contain any pure functions.

Class member *member* declared outside its class *Compiler error*

C++ class member functions can be declared inside the class declaration only. Unlike nonmember functions, they cannot be declared multiple times or at other locations.

Code has no effect *Compiler warning*

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.

Colon expected *MAKE error*

Your implicit rule is missing a colon at the end.

```
.c.obj:     # Correct
.c.obj      # Incorrect
```

Command arguments too long *MAKE error*

The arguments to a command were more than the 255-character limit imposed by the operating system.

Command syntax error *MAKE error*

This message occurs if

- The first rule line of the makefile contained any leading whitespace.
- An implicit rule did not consist of *.ext.ext:*.
- An explicit rule did not contain a name before the **:** character.
- A macro definition did not contain a name before the **=** character.

Command too long *MAKE error*

This message is issued when the command line length exceeds MAKE's internal buffer (4096 bytes). The length of a command has exceeded 512 characters. You might want to use a response file.

Compiler could not generate copy constructor for class *class* *Compiler error*

The compiler cannot generate a needed copy constructor due to language rules.

Compiler could not generate default constructor for class *class* *Compiler error*

The compiler cannot generate a needed default constructor due to language rules.

Compiler could not generate operator= for class *class* *Compiler error*

The compiler cannot generate a needed assignment operator due to language rules.

Compiler stack overflow *Compiler error*

The compiler ran out of memory during compilation. Review the portion of code flagged by the error message to ensure that it uses memory correctly.

Compiler table limit exceeded *Compiler error*

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.

Compound statement missing } *Compiler error*

The compiler reached the end of the source file and found no closing brace. This is often caused by mismatched braces.

Condition is always false *Compiler warning*
Condition is always true *Compiler warning*

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' */
   }
}
```

Conflicting type modifiers *Compiler error*

This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier can be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) can be given for a function.

Constant expression required *Compiler error*

Arrays must be declared with constant size. This error is commonly caused by misspelling a **#define** constant.

Constant is long *Compiler warning*

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**.

Constant member *member* in class without constructors *Compiler error*

A class that contains constant members must have at least one user-defined constructor; otherwise, there would be no way to initialize such members.

Constant member *member* is not initialized *Compiler warning*

This C++ class contains a constant member *member*, which does not have an initialization. Constant members can only be initialized; they cannot be assigned to.

Constant out of range in comparison *Compiler warning*

Your source file includes a comparison involving a constant subexpression that was outside the range allowed by the other subexpression's type. For example, comparing an **unsigned** quantity to –1 makes no sense. To get an **unsigned** constant greater than 2147483647 (in decimal), you should either cast the constant to **unsigned** or append a letter *u* or *U* to the constant.

When this message is issued, the compiler still generates 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 still performs the test.

Constant variable *variable* must be initialized *Compiler error*

This C++ object is declared **const**, but is not initialized. Since no value can be assigned to it, it must be initialized at the point of declaration.

Constructor cannot be declared const or volatile *Compiler error*

A constructor has been declared as **const** and/or **volatile**, and this is not allowed.

Constructor cannot have a return type specification *Compiler error*

C++ constructors have an implicit return type used by the compiler, but you cannot declare a return type or return a value.

Conversion may lose significant digits *Compiler warning*

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. Because **int** type and **long** type variables don't have the same size, this kind of conversion can alter the behavior of a program.

Conversion operator cannot have a return type specification *Compiler error*

This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function **operator** cannot specify any return type.

Conversion to *type* will fail for members of virtual base *class* *Compiler error*

This warning can occur when a member pointer (whose class contains virtual bases) is cast to another member-pointer type and you use the **–Vv** option. This error indicates 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.

Could not allocate memory for per module data *Librarian error*

The librarian has run out of memory.

Could not create list file *filename* *Librarian error*

The librarian could not create a list file for the library. This could be due to lack of disk space.

Could not find a match for *argument(s)* *Compiler error*

No C++ function could be found with parameters matching the supplied arguments.

Could not find file *filename* *Compiler error*

The compiler is unable to find the file supplied on the command line.

Could not get procedure address from DLL *filename* *Linker error*

The linker was not able to get a procedure from the specified DLL. Check to make sure that you have the correct DLL version.

Could not load DLL *filename* *Linker error*

The linker was not able to load the specified DLL. Check to make sure the DLL is on your path.

Could not write output *Librarian error*

The librarian could not write the output file.

***filename* couldn't be created, original won't be changed** *Librarian warning*
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.

Couldn't get LE/LIDATA record buffer *Librarian error*
Command-line error. See the **Out of memory reading LE/LIDATA record from object module** message.

Couldn't get procedure address from dll *dll* *Linker error*
The linker wasn't able to get a procedure from the specified DLL. Check to make sure you have the correct version of the DLL.

Couldn't load dll *dll* *Linker error*
The linker wasn't able to load the specified DLL. Check to make sure the DLL is on your path.

Cycle in include files: *filename* *MAKE error*
This error message is issued if a makefile includes itself in the make script.

Debug information enabled, but no debug information found in OBJs *Linker warning*
No part of the application was compiled with debug information, but you requested that debug information be turned on in the link. The image will be written without any debug information section.

Debug information in module *module* will be ignored *Linker warning*
Object files compiled with debug information now have a version record. The major version of this record is higher than what TLINK currently supports and TLINK did not generate debug information for the module in question.

Declaration does not specify a tag or an identifier *Compiler error*
This declaration doesn't declare anything. This might be a **struct** or **union** without a tag or a variable in the declaration. C++ requires that something be declared.

Declaration is not allowed here *Compiler error*
Declarations cannot be used as the control statement for **while**, **for**, **do**, **if**, or **switch** statements.

Declaration missing ; *Compiler error*
Your source file contained a declaration that was not followed by a semicolon.

Declaration syntax error *Compiler error*
Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.

Declaration terminated incorrectly *Compiler error*
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.

Declaration was expected *Compiler error*
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.

Declare operator delete (void*) or (void*, size_t) *Compiler error*
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 be declared using the single-parameter form only.

Declare operator delete[] (void*) or (void*, size_t) *Compiler error*

Declare the operator delete with one of the following:

- A single void* parameter
- 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 be declared using the single-parameter form only.

Declare type *type* prior to use in prototype *Compiler warning*

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 { /* ... */ };
```

Because there is no **struct** *s* in scope at the prototype for *func*, the type of parameter *ps* is a pointer to undefined **struct** *s*, and is not the same as the **struct** *s* that is later declared. This results in warning and error messages about incompatible types, which would be mysterious without this warning message. To fix the problem, you can move the declaration for **struct** *s* ahead of any prototype that references it, or add the incomplete type declaration `struct s;` ahead of any prototype that 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.

Default argument value redeclared *Compiler error*

When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.

Default argument value redeclared for parameter *parameter* *Compiler error*

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.

Default expression may not use local variables *Compiler error*

A default argument expression is not allowed to use any local variables or other parameters.

Default outside of switch *Compiler error*

The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched braces.

Default value missing *Compiler error*

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.

Default value missing following parameter *parameter* *Compiler error*

All parameters following the first parameter with a default value must also have defaults specified.

Define directive needs an identifier *Compiler error*

The first non-whitespace character after a **#define** must be an identifier. The compiler found some other character.

symbol* defined in module *module* is duplicated in module *module *Linker error or warning*

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 TLINK doesn't know which is valid. A warning results if TLINK finds one of the duplicated symbols in a library and finds the other in an .OBJ file; in this case, TLINK uses the one in the .OBJ file.

Delete array size missing] *Compiler error*

The array specifier in an operator is missing a right bracket.

Destructor cannot be declared const or volatile *Compiler error*

A destructor has been declared as **const** and/or **volatile**, and this is not allowed.

Destructor cannot have a return type specification *Compiler error*

It is illegal to specify the return type for a destructor.

Destructor for *class* is not accessible *Compiler error*

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.

Destructor for *class* required in conditional expression *Compiler error*

If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor, because the variable might or might 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.

Destructor name must match the class name *Compiler error*

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.

Divide error *Run-time error*

You've tried to divide an integer by zero. You can trap this error with the **signal** function. Otherwise, Borland C++ calls **abort** and your program terminates.

Division by zero *Compiler error*

Your source file contained a division or remainder operator in a constant expression with a zero divisor.

Division by zero *Compiler warning*

A division or remainder operator expression had a literal zero as a divisor.

Division by zero *MAKE error*

A division or remainder operator in an **!if** statement has a zero divisor.

do statement must have while *Compiler error*

Your source file contained a **do** statement that was missing the closing **while** keyword.

do-while statement missing (*Compiler error*

In a **do** statement, the compiler found no left parenthesis after the **while** keyword.

do-while statement missing) *Compiler error*

In a **do** statement, the compiler found no right parenthesis after the test expression.

do-while statement missing ; *Compiler error*

In a **do** statement test expression, the compiler found no semicolon after the right parenthesis.

***filename* does not exist – don't know how to make it** *MAKE error*

There is a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.

DOS error, ax = *number* *Linker error*

This error occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes in which this error could occur are read, write, seek, and close.

DOSSEG directive ignored in *module* *Linker warning*

This warning indicates that the DOSSEG directive is no longer supported by the linker.

Duplicate case *Compiler error*

Each **case** of a **switch** statement must have a unique constant expression value.

***filename* (*linenum*): Duplicate external name in exports** *Linker warning*

Two export functions listed in the EXPORTS section of a module definition file defined the same external name. For example:

```
EXPORTS
    AnyProc=MyProc1
    AnyProc=MyProc2
```

Duplicate file *file* in list, not added! *Librarian error*

When building a library module, you specified an object file more that once.

Duplicate handler for *type1*, already had *type2* *Compiler error*

It's illegal to specify two handlers for the same type.

***filename* (*linenum*): Duplicate internal name in exports** *Linker warning*

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
```

***filename* (*linenum*): Duplicate internal name in imports** *Linker warning*

Two import functions listed in the IMPORTS section of the module definition file defined the same internal name. For example:

```
IMPORTS
    AnyProc=MyMod1.MyProc1
    AnyProc=MyMod2.MyProc2
```

Duplicate ordinal for exports: *string* (*ordval1*) and *string* (*ordval2*) *Linker error*

Two exports have been found for the same symbol, but with differing ordinal values. You must use the same ordinal value or remove one of the exports.

Empty LEDATA record in module *module* *Linker warning*

This warning can happen if the translator emits a data record containing data. If this should happen, report the occurrence to the translator vendor; there should be no bad side effects from the record.

Enum syntax error *Compiler error*

An **enum** declaration did not contain a properly formed list of identifiers.

Error changing file buffer size *Librarian error*

The librarian is attempting to adjust the size of a buffer used while reading or writing a file, but there is not enough memory. It is likely that quite a bit of system memory will have to be freed up to resolve this error.

Error directive: *message* *Compiler error*

The text of the **#error** directive being processed in the source file is displayed.

Error directive: *message* *MAKE error*

MAKE has processed an **#error** directive in the source file, and the text of the directive is displayed in the message.

Error opening *filename* *Librarian error*

librarian cannot open the specified file for some reason.

Error opening *filename* for output *Librarian error*

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. This error occurs when the target file exists but is marked as a read-only file.

Error renaming *filename* to *filename* *Librarian error*

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 is posted.

Error writing output file *Compiler error*

An operating system error that prevents Borland C++ from writing an .OBJ, .EXE, or temporary file. Check the output directory and make sure that this is a valid directory. Also check that there is enough free disk space.

__*except* or __*finally* expected following __*try* *Compiler error*

In C, a __*try* block must be followed by an __*except* or __*finally* handler block.

Exception handling not enabled *Compiler error*

A 'try' block was found with the exception handling disabled.

Exception handling variable may not be used here *Compiler error*

An attempt has been made to use one of the exception handling values that are restricted to particular exception handling constructs, such as GetExceptionCode().

Exception specification not allowed here *Compiler error*

Function pointer type declarations are not allowed to contain exception specifications.

Export *symbol* is already imported *Linker error*

You have attempted to export a symbol that you are also importing.

Export *symbol* is duplicated *Linker warning*

This warning occurs if two different symbols with the same name are exported by the use of **_export**. The linker cannot determine which definition it should export, and therefore uses the first symbol.

Expression expected *Compiler error*

An expression was expected here, but the current symbol cannot begin an expression. This message can 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.

Expression of scalar type expected *Compiler error*

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.

Expression syntax *Compiler error*

This is a catchall error message when the compiler parses an expression and encounters a serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

Expression syntax error in !if statement *MAKE error*

The expression in an **!if** statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

***reason* – extended dictionary not created** *Librarian warning*

The librarian could not produce the extended dictionary because of the *reason* given in the warning message.

Extended dictionary not found in library *library*, extended dictionaries ignored *Linker warning*

The **/E** option for TLINK requires that all libraries in the link have extended dictionaries. When a library without an extended dictionary is encountered during a link operation in which the **/E** option is specified, the linker abandons extended dictionary processing and proceeds to link with a default link.

Extern variable cannot be initialized *Compiler error*

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.

Extra argument in template class name *template* *Compiler error*

A template class name specified too many actual values for its formal parameters.

Extra parameter in call *Compiler error*

A call to a function, via a pointer defined with a prototype, had too many arguments given.

Extra parameter in call to *function* *Compiler error*

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

Failed read from *filename* *Linker error*

An OS/2 error occurred while TLINK read the module definition file. This usually means that a premature end-of-file occurred.

Failed write to *filename* *Linker error*

The Linker was unable to write to the file.

__far16 may only be used with __pascal or __cdecl *Compiler error*

When you use **__far16** to make calls to functions or reference data in a 16-bit DLL, such functions and data can be modified only by **__pascal** or **__cdecl**.}]

Far COMDEFs are not supported *Linker error*

TLINK does not support far COMDEFs. Compile without the **–Fc** option, or remove comdef usage from the 16-bit code you are using.

File must contain at least one external declaration *Compiler error*

This compilation unit was logically empty, containing no external declarations. ANSI C and C++ require that something be declared in the compilation unit.

Filename too long *Compiler error*

The file name given in an **#include** directive was too long for the compiler to process. Path names must be no longer than 260 characters.

File name too long *MAKE error*

The path name in an **!include** directive overflowed MAKE's internal buffer (512 bytes).

***filename* file not found** *Librarian error*

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 TLIB generates this 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. Rebuilding your project should resolve the problem or indicate where the errors have occurred.

Fixup to zero length segment in module *module* *Linker error*

A reference has been made past the end of an image segment. This reference would end up accessing an invalid address, and has been flagged as an error.

Fixup overflow at *address*, target = *address* *Linker warning*

These messages indicate an incorrect data or code reference in an object file that TLINK 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. These errors 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 in a high-level language other than Borland C++, there might be other possible causes for this message. Even in Borland C++, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Fixup to zero length segment in module *module* *Linker error*

This error usually occurs if you make a reference to a segment that doesn't contain any data. If the segment isn't grouped with other segments, the result is a zero-length physical segment, which cannot exist. The linker therefore cannot make a reference to it.

Floating point error: Divide by 0. *Run-time error*
Floating point error: Domain. *Run-time error*
Floating point error: Overflow. *Run-time error*

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).
- "Overflow" means the result is +INF (infinity) or –INF with complete loss of precision, such as assigning 1e200*1e200 to a **double**.

Floating point error: Partial loss of precision. *Run-time error*
Floating point error: Underflow. *Run-time error*

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*.

Floating point error: Stack fault. *Run-time error*

The floating-point stack has been overrun. This error does not normally occur and might be due to assembly code using too many registers or 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*. See the functions *_control87* and *signal* for details.

for statement missing (*Compiler error*

In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

for statement missing) *Compiler error*

In a **for** statement, the compiler found no right parenthesis after the control expressions.

for statement missing ; *Compiler error*

In a **for** statement, the compiler found no semicolon after one of the expressions.

Friends must be functions or classes *Compiler error*

A **friend** of a C++ class must be a function or another class.

Function call missing) *Compiler error*

The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

Function calls not supported *Compiler error*

In integrated debugger expression evaluation, calls to functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

Function *function* cannot be static *Compiler error*

Only ordinary member functions and the operators **new** and **delete** can be declared static. Constructors, destructors, and other operators must not be static.

Function defined inline after use as extern *Compiler error*

Functions cannot become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely.

Function definition cannot be a typedef'ed declaration *Compiler error*

In ANSI C a function body cannot be defined using a typedef with a function Type.

Function *function* should have a prototype *Compiler error*

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.

Function should return a value *Compiler warning*

This function was declared (perhaps 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**.

Functions *function1* and *function2* both use the same dispatch number *Compiler error*

This error is the result of a dynamically dispatched virtual table (DDVT) problem. When you override a dynamically dispatchable function in a derived class, use the same dispatch index. Each function within the same class hierarchy must use a different dispatch index.

Functions cannot return arrays or functions *Compiler error*

A function cannot return an array or a function. Only pointers or references to arrays or functions can be returned.

Functions containing *reserved word* are not expanded inline *Compiler warning*

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.

Functions containing local destructors are not expanded inline in function *function* *Compiler warning*

You've created an inline function for which Borland C++ turns off inlining. You can ignore this warning if you like; the function will be generated out of line.

Functions may not be part of a struct or union *Compiler error*

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++.

Functions with exception specifications are not expanded inline *Compiler warning*

Check your inline code for lines containing exception specifications.

Functions with taking class-by-value argument(s) are not expanded inline *Compiler warning*
When exception handling is enabled, functions that take class arguments by value cannot be expanded inline (Note that functions taking class parameters by reference are not subject to this restriction.)

General error *Linker error*
General error in library file *filename* in module *module* near module file offset 0xyyyyyyyy. *Linker error*
General error in module *module* near module file offset 0xyyyyyyyy *Linker error*
The linker gives as much information as possible about what processing was happening at the time of the unknown fatal error. Call Borland Technical Support with information about .OBJ or .LIB files.

General error
General error in module *module_name*
The linker encountered an internal error. The circumstances of this error should be reported to Borland Technical Support.

Global anonymous union not static *Compiler error*
In C++, a global anonymous union at the file level must be static.

Goto bypasses initialization of a local variable *Compiler error*
In C++ it is illegal to bypass the initialization of a local variable in any way. You'll get this error when there is a **goto** that tries to transfer control past this local variable.

Goto into an exception handler is not allowed *Compiler error*
It's illegal to jump into a try block or an exception handler that's attached to a try block.

Goto statement missing label *Compiler error*
The **goto** keyword must be followed by an identifier.

Handler for *type1* hidden by previous handler for *type2* *Compiler warning*
This warning is issued when a handler for a type *D* that is derived from type *B* is specified after a handler for *B*, since the handler for *D* will never be invoked.

***specifier* has already been included** *Compiler error*
This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.

Hexadecimal value contains more than 3 digits *Compiler warning*
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.

function1* hides virtual function *function2 *Compiler warning*
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.

Identifier expected *Compiler error*
An identifier was expected here, but not found. In C, this error occurs 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.

Identifier *identifier* cannot have a type qualifier *Compiler error*
A C++ qualifier *class::identifier* cannot 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 (which is redundant but legal).

If statement missing (*Compiler error*

In an **if** statement, the compiler found no left parenthesis after the **if** keyword.

If statement missing) *Compiler error*

In an **if** statement, the compiler found no right parenthesis after the test expression.

If statement too long *MAKE error*
Ifdef statement too long *MAKE error*
Ifndef statement too long *MAKE error*

An **If**, **Ifdef**, or **Ifndef** statement has exceeded 4096 characters.

Ignored *module*, path is too long *Librarian warning*

The path to a specified .OBJ or .LIB file is greater than 252 characters. The max path to a file for librarian is 252 characters.

Ill-formed pragma *Compiler warning*

A pragma does not match one of the pragmas expected by the Borland C++ compiler.

Illegal ACBP byte in SEGDEF *Linker error*

This is usually a translator error.

Illegal character *character* (0x*value*) *Compiler error*

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.

Illegal character in constant expression *expression* *MAKE error*

MAKE encountered a character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.

Illegal component to GRPDEF *Linker error*

This is usually a translator error.

Illegal initialization *Compiler error*

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.

Illegal local public in *module* *Linker warning*

The message occurs when the linker sees an LPUBDEF record with an offset of zero for a VIRDEF that resides in an overlay segment. This can happen if you are trying to use structured exception support in an application that uses overlays.

Illegal octal digit *Compiler error*

An octal constant containing a digit of 8 or 9 was found.

Illegal parameter to __emit__ *Compiler error*

You supplied an argument to *__emit__* that is not a constant or an address.

Illegal pointer subtraction *Compiler error*

This is caused by attempting to subtract a pointer from a nonpointer.

Illegal structure operation *Compiler error*

In C or C++, structures can be used with dot (.), address-of (&), or assignment (=) operators, or can 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.

Illegal to take address of bit field *Compiler error*

It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.

Illegal type of entry point *Linker error*
Only entry points that target a segment index are supported.

Illegal use of floating point *Compiler error*
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.

Illegal use of member pointer *Compiler error*
Pointers to class members can be used only 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.

Illegal use of pointer *Compiler error*
Pointers can be used only with addition, subtraction, assignment, comparison, indirection (*) or arrow (**->**) operators. Your source file used a pointer with some other operator.

Image base address must be a multiple of 0x10000 *Linker error*
Based images, the base address must be a multiple of 0x10000.

Implicit conversion of *type1* to *type2* not allowed *Compiler error*
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.

Import *record* does not match previous definition *Linker warning*
This warning usually occurs if an IMPDEF record appears in an import library at the same time as the import in question is imported from a .DEF file. If the description of the imports differ in internal name or ordinal, this warning appears, and the first definition is used. This warning can be controlled with the **–w** switch.

Import *symbol* in module *module* clashes with prior module *Linker error*
An import symbol can appear only once in a library file. A module that is being added to the library contains an import that is already in a module of the library and it cannot be added again.

Improper use of typedef *identifier* *Compiler error*
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.

Include files nested too deep *Compiler error*
When the compiler detects that header files are nested more than 1,000 levels deep, it assumes that the header file is recursive, and stops compilation with this (fatal) error.

***filename* (*linenum*): Incompatible attribute**
TLINK encountered incompatible segment attributes in a CODE or DATA statement. For instance, both PRELOAD and LOADONCALL can't be attributes for the same segment.

Incompatible type conversion *Compiler error*
The cast requested can't be done. Check the types.

Incorrect command-line argument: *argument* *MAKE error*
You've used incorrect command-line arguments.

Incorrect command-line option: *option* *Compiler error*
The compiler did not recognize the command-line parameter as legal.

Incorrect configuration file option: *option* *Compiler error*
The compiler did not recognize the configuration file parameter as legal; check for a preceding hyphen (–).

Incorrect number format *Compiler error*

The compiler encountered a decimal point in a hexadecimal number.

Incorrect use of default *Compiler error*

The compiler found no colon after the **default** keyword in a **case** statement.

Initialization is only partially bracketed *Compiler warning*

When structures are initialized, nested pairs of braces can be used to mark the initialization of each member of the structure. Bracketing the members ensures that your idea and the compiler's idea of the initializations are the same. The compiler issues this warning when the brackets are not equally matched.

Initializing enumeration with *type* *Compiler warning*

You're trying to initialize an **enum** variable to a different type. For example,

```
enum count { zero, one, two } x = 2;
```

results 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.

Inline assembly not allowed *Compiler error*

Your source file contains inline assembly-language statements and you're trying to compile it from within the integrated environment. You must use BCC to compile source files that contain inline assembly.

Inline assembly not allowed in inline and template functions *Compiler error*

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.

int and string types compared *MAKE error*

You have tried to compare an integer operand with a string operand in an **!if** or **!elif** expression.

***name* in the (non)resident name table is too long** *Linker error*

OS/2 limits the length of names in the resident and non-resident name tables to 127 characters. Names of this length may not be exported.

Internal linker error *errorcode* *Linker error*

An error occurred in the internal logic of TLINK. 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 Technical Support.

Invalid combination of opcode and operands *Compiler error*

The built-in assembler does not accept this combination of operands. Possible causes are the following:

- There are too many or too few operands for this assembler opcode.
- The number of operands is correct, but their types or order do not match the opcode; for example **DEC 1**, **MOV AX**, or **MOV 1,AX**. Try prefacing the operands with type overrides; for example **MOV AX, WORD PTR foo**.

Invalid exe filename: *filename* *Linker error*

The exe filename had an incorrect extension, such as .OBJ, .MAP, .LIB, .DEF, or .RES.

Invalid extended dictionary in library *library*: extended dictionaries ignored *Linker warning*

The extended dictionary in the library is invalid. Run TLIB /E on the library.

Invalid indirection *Compiler error*

The indirection operator (*) requires a non-**void** pointer as the operand.

Invalid macro argument separator *Compiler error*

In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.

Invalid map filename: *filename* *Linker error*

The map filename had an incorrect extension, such as .OBJ, .EXE, .DLL, .LIB, .DEF, or .RES.

Invalid page size value ignored *Librarian warning*

Invalid page size is given. The page size must be a power of 2, and it cannot be smaller than 16 or larger than 32,768.

Invalid pointer addition *Compiler error*

Your source file attempted to add two pointers together.

Invalid register combination (e.g. [BP+BX]) *Compiler error*

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[EBX]** (where **Local** is a local variable) appears valid, it is not, because the final operand would become **Local[BP+EBX]**.

Invalid target->/T*target* *Linker error*

The command-line linker found an invalid target. Valid targets are 'w' and 'd'.

Invalid template argument list *Compiler error*

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.

Invalid template qualified name *template::name* *Compiler error*

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(){}
```

Invalid use of dot *Compiler error*

An identifier must immediately follow a period operator (**.**).

Invalid use of template *template* *Compiler error*

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**.

Irreducible expression tree *Compiler error*

This is a sign of some form of compiler error. An expression on the indicated line of the source file has caused the code generator to be unable to generate code. The offending expression should be avoided. Notify Borland Technical Support if the compiler encounters this error.

base* is an indirect virtual base class of *class *Compiler error*

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).

***identifier* is assigned a value that is never used** *Compiler warning*

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.

***identifier* is declared as both external and static** *Compiler warning*

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.

***identifier* is declared but never used** *Compiler warning*

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.

identifier* is not a member of *struct *Compiler error*

You are trying to reference *identifier* as a member of ***struct***, but it is not a member. Check your declarations.

***identifier* is not a non-static data member and can't be initialized here** *Compiler error*

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.

***identifier* is not a parameter** *Compiler error*

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.

***type* is not a polymorphic class type** *Compiler error*

A dynamic_cast was used with a pointer to a class that was compiled with the –RT compiler option disabled.

base* is not a public base class of *class *Compiler error*

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.

***filename* is not a valid library** *Linker warning*

This error occurs if a file that wasn't a valid library module was passed to the linker in the library section.

***member* is not a valid template type member** *Compiler error*

A member of a template with some actual arguments that depend on the formal arguments of an enclosing template was found not to be a member of the specified template in a particular instance.

***member* is not accessible** *Compiler error*

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.

function is obsolete *Compiler warning*

The compiler generates this warning message when it encounters a function that is obsolete. Functions marked by this error message will be removed from the next version of the product.

Iterated data block too large in module _name_ *Linker error*

The linker encountered an iterated data block which was too large to expand. This can happen with assembler code where arrays of large static data structures are defined. To prevent this, declare the static block by using DB and DUP, using the size of the structure.

Last parameter of _operator_ must have type int *Compiler error*

When a postfix **operator++** or **operator--** is declared, the last parameter must be declared with the type **int**.

Library too large, please restart with library page size _size_ *Librarian error*

The library being created could not be built with the current library page size.

Linkage specification not allowed *Compiler error*

Linkage specifications such as **extern** "C" are allowed only at the file level. Move this function declaration out to the file level.

Linker name conflict for _function_ *Compiler error*

When the mangled name of a C++ inline function or a virtual table is too long and has to be truncated (this happens most often with templates), and the truncated name matches a previously generated function or virtual table, this error is issued by the compiler. The problem can be fixed by changing the name of the class or function, or by compiling with the **-Vs** option.

Linker stack overflow *Linker error*

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.

Lvalue required *Compiler error*

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.

Macro argument syntax error *Compiler error*

An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.

Macro expansion too long *Compiler error*

A macro cannot expand to more than 4,096 characters.

Macro expansion too long *MAKE error*

A macro cannot expand to more than 4,096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

Macro substitute text _string_ is too long *MAKE error*
Macro replace text _string_ is too long *MAKE error*

The macro substitution or replacement text *string* overflowed MAKE's internal buffer of 512 bytes.

main must have a return type of int *Compiler error*

In C++, function **main** has special requirements, one of which is that it cannot be declared with any return type other than **int**.

Matching base class function for _function_ has different dispatch number. *Compiler error*

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.

Matching base class function for _function_ is not dynamic *Compiler error*

If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.

Maximum precision used for member pointer type *type* *Compiler warning*

When you use the **–Vmd** option, the compiler has to use the most general (and the least efficient) representation for that member pointer type when it is declared and its class hasn't been fully defined. This can cause less efficient code to be generated (and make the member pointer type unnecessarily large), and can also cause problems with separate compilation.

Member *member* cannot be used without an object *Compiler error*

This means that the user has written `class::member` where *member* is an ordinary (nonstatic) 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`.

Member function must be called or its address taken *Compiler error*

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.

Member *member* has the same name as its class *Compiler error*

A static data member, enumerator, member of an anonymous union, or nested type cannot have the same name as its class. Only a member function or a nonstatic member can have a name that is identical to its class.

Member identifier expected *Compiler error*

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.

Member is ambiguous: *member1* and *member2* *Compiler error*

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.

Member *member* is initialized more than once *Compiler error*

In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

Member pointer required on right side of .* or –>* *Compiler error*

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.

Memory full listing truncated! *Librarian warning*

The librarian has run out of memory creating a library listing file. A list file will be created but is not complete.

Memory reference expected *Compiler error*

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]**.

Misplaced break *Compiler error*

The compiler encountered a **break** statement outside a **switch** or looping construct.

Misplaced continue *Compiler error*

The compiler encountered a **continue** statement outside a looping construct.

Misplaced decimal point *Compiler error*

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

Misplaced elif directive *Compiler error*

The compiler encountered an **#elif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

Misplaced elif statement *MAKE error*

An **!elif** directive is missing a matching **!if** directive.

Misplaced else *Compiler error*

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.

Misplaced else directive *Compiler error*

The compiler encountered an **#else** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

Misplaced else statement *MAKE error*

An **!else** directive does not have a matching **!if** directive.

Misplaced endif directive *Compiler error*

The compiler encountered an **#endif** directive without a matching **#if**, **#ifdef**, or **#ifndef** directive.

Misplaced endif statement *MAKE error*

An **!endif** directive does have a matching **!if** directive.

Mixed common types in module *module*. Cannot mix COMDEFs and VIRDEFs. *Linker error*

You cannot mix both COMDEFs and VIRDEFs. This should only occur if you are using assembler code, and are explicitly creating a common segment, along with virtual segments. Remove one or the other.

Mixing pointers to different 'char' types *Compiler warning*

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.)

Multiple base classes require explicit class names *Compiler error*

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.

Multiple declaration for *identifier* *Compiler error*

This identifier was improperly declared more than once. This might be caused by conflicting declarations such as `int a; double a;`, by a function declared two different ways, by a label repeated in the same function, or by some declaration repeated other than an **extern** function or a simple variable (in C).

Multiple entry points defined *Linker error*

More than one entry point was defined for the application. You can only have one entry point.

Multiple stack segments found. The most recent one will be used. *Linker warning*

This warning occurs when two stack segments of different names are defined in the object modules. The startup code defines a stack segment for the application. The warning can be controlled with the **–w** switch.

***identifier* must be a member function** *Compiler error*

Most C++ operator functions can 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.

***identifier* must be a member function or have a parameter of class type** *Compiler error*

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.

***identifier* must be a previously defined class or struct** *Compiler error*

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.

***identifier* must be a previously defined enumeration tag** *Compiler error*

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.

***identifier* must be declared with no parameters** *Compiler error*

This C++ operator function was incorrectly declared with parameters.

***operator* must be declared with one or no parameters** *Compiler error*

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).

***operator* must be declared with one or two parameters** *Compiler error*

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 (for the postfix version).

***identifier* must be declared with one parameter** *Compiler error*

This C++ operator function was incorrectly declared with more than one parameter.

***identifier* must be declared with two parameters** *Compiler error*

This C++ operator function was incorrectly declared with other than two parameters.

Must take address of a memory location *Compiler error*

Your source file used the address-of operator (**&**) with an expression that cannot be used that way; for example, a register variable (in C).

Need an identifier to declare *Compiler error*

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.

'new' and 'delete' not supported *IDE debugger error*

The integrated debugger does not support the evaluation of 'new' and 'delete'.

No : following the ? *Compiler error*

The question mark (**?**) and colon (**:**) operators do not match in this expression. The colon might have been omitted, or parentheses might be improperly nested or missing.

No automatic data segment *Linker warning*

No group named DGROUP was found. Because Borland's 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.

No base class to initialize *Compiler error*

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.

No closing quote *MAKE error*

There is no closing quote for a string expression in a **!if** or **!elif** expression.

No declaration for function *function* *Compiler warning*

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.

No .DEF file; using defaults

This warning occurs when you do not specify a .DEF file for the link. It can be controlled by the **-w** switch.

No file name ending *Compiler error*

The file name in an **#include** statement was missing the correct closing quote or angle bracket.

No filename ending *MAKE error*

The file name in an **!include** statement is missing the correct closing quote or angle bracket.

No file names given *Compiler error*

The command line of the Borland C++ command-line compiler (BCC) contained no file names. You must specify a source file name.

No internal name for IMPORT in .DEF file *Linker error*

The .DEF file has a semantic error. You probably forgot to put the internal name for an import before the module name. For example:

```
IMPORTS
   _foo.1
```

The proper syntax is:

```
IMPORTS
   _foo=mydll.1
```

No macro before = *MAKE error*

You must give a macro a name before you can assign it a value.

No match found for wildcard *expression* *MAKE error*

There are no files matching the wildcard *expression* for MAKE to expand. For example, if you write

```
prog.exe: *.obj
```

MAKE sends this error message if there are no files with the extension .OBJ in the current directory.

No module definition file specified; using defaults *Linker warning*

This warning occurs when you do not specify a .DEF file for the link.

No output file specified *Linker error*

No EXE or DLL file was specified. Because the linker defaults to the first .OBJ name, this error is usually caused because no object object files were included.

No program starting address defined *Linker warning*

This warning means that no module defined the initial starting address of the program. This is probably caused by forgetting to link in the initialization module C0*x*.OBJ.

No stack *Linker warning*

This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Borland C++, or for any application program that will be converted to a .COM file. Except for DLLs, this indicates an error.

If a Borland C++ program produces this message for any but the tiny memory model, make sure you are using the correct C0*x* startup object files.

Not enough memory for command-line buffer *Librarian warning*
This error occurs when TLIB runs out of memory.

Not setting protected DLL flag *Linker warning*
The linker found a protected DLL comment record in an OBJ, and the you asked to link an EXE rather than a DLL.

No terminator specified for in-line file operator *MAKE error*
The makefile contains either the **&&** or **<<** command-line operators to start an inline file, but the file is not terminated.

No type information *IDE debugger error*
The integrated debugger has no type information for this variable. Ensure that you've compiled the module with debug information.

Non-ANSI Keyword Used: *keyword* *Compiler warning*
A non-ANSI keyword (such as **__fastcall**) was used when strict ANSI conformance was requested via the **-A** option.

Non-const function *function* called for const object *Compiler warning*
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.

Non-virtual function *function* declared pure *Compiler error*
Only virtual functions can be declared pure, because derived classes must be able to override them.

Non-volatile function *function* called for volatile object *Compiler warning*
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 can be called for a volatile object.

Nonportable pointer comparison *Compiler warning*
Your source file compared a pointer to a nonpointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.

Nonportable pointer conversion *Compiler error*
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 might not make any sense, so be sure this is what you want to do.

Nonportable pointer conversion *Compiler warning*
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.

Nonresident Name Table is greater than 64K *Linker warning*
The maximum size of the Nonresident name table is 64K (in accordance with the industry-wide executable specification standard). The linker continues with the link but ignores any subsequent Nonresident names encountered during linking.

Nontype template argument must be of scalar type *Compiler error*
A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.

***filename* not a MAKE** *MAKE error*
The file you specified with the **-W** and the MAKE.EXE in the path is not a valid MAKE.EXE.

Not a valid expression format type *IDE debugger error*
You used an invalid format specifier following an expression in the integrated debugger. A valid format specifier is a repeat value (optional) followed by one of the following format specifiers: c, d, f[n], h, m, p, r, s, or x.

Not an allowed type *Compiler error*

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

Not enough memory *MAKE error*

All your working storage has been exhausted.

Not enough memory to run application *Linker error*

There is not enough memory to run TLINK. Try reducing the size of any RAM disk or disk cache currently active. If you're running real mode, try using the MAKE **–S** option, or removing TSRs and network drivers.

***module* not found in library** *Librarian warning*

An attempt to perform either a '_' or '*' on a library has been made and the indicated object does not exist in the library.

Numeric constant too large *Compiler error*

String and character escape sequences larger than hexadecimal \xFF or octal \377 cannot be generated. Two-byte character constants can 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`.

Object module *filename* is invalid *Librarian error*

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.

Objects of type *type* cannot be initialized with {} *Compiler error*

Ordinary C structures can be initialized with a set of values inside braces. C++ classes can be initialized with constructors only if the class has constructors, private members, functions, or base classes that are virtual.

Old debug information in module *module* will be ignored *Linker warning*

Debug information in the .OBJ file is incompatible with this linker, and it will be ignored.

Only <<KEEP or <<NOKEEP *MAKE error*

You have specified something besides KEEP or NOKEEP when closing a temporary inline file.

Only member functions may be 'const' or 'volatile' *Compiler error*

Something other than a class member function has been declared **const** and/or **volatile**.

Only one of a set of overloaded functions can be "C" *Compiler error*

C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you want 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.)

Operand of delete must be non-const pointer *Compiler error*

It is illegal to delete a constant pointer value using operator **delete**.

Operator [ ] missing] *Compiler error*

The C++ **operator[ ]** was declared as **operator [**. You must add the missing] or otherwise fix the declaration.

Operator –> must return a pointer or a class *Compiler error*

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.

Operator delete must return void *Compiler error*

This C++ overloaded operator **delete** was declared in some other way.

Operator delete[] must return void *Compiler error*

This C++ overloaded operator **delete** was declared in some other way. Declare the **delete** with one of the following:

- A single void* parameter
- 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 be declared using the single-parameter form only.

Operator must be declared as function *Compiler error*

An overloaded operator was declared with something other than function type.

Operator new must have an initial parameter of type size_t *Compiler error*

Operator **new** can be declared with an arbitrary number of parameters, but it must always have at least one parameter that specifies the amount of space to allocate.

Operator new[] must have an initial parameter of type size_t *Compiler error*

Operator **new** can be declared with an arbitrary number of parameters. It must always have at least one parameter that specifies the amount of space to allocate.

Operator new must return an object of type void * *Compiler error*

The C++ overloaded operator **new** was declared another way.

Operator new[] must return an object of type void * *Compiler error*

This C++ overloaded operator **new** was declared another way.

Operators may not have default argument values *Compiler error*

It is illegal for overloaded operators to have default argument values.

Out of memory *Compiler error*

The total working storage is exhausted. Compile the file on a machine with more memory.

Out of memory *Librarian error*

For any number of reasons, the librarian or Borland 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.

If you get this message because the public symbol tables have grown too large, you must free up memory. For the command line this could involve removing TSR's or device drivers using real mode memory. In the IDE, some additional memory can be gained by closing editors.

Out of memory *Linker error*

The linker has run out of dynamically allocated memory needed during the link process. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together.

Out of memory at library *library*: extended dictionaries ignored *Linker warning*

The linker ran out of memory allocating space to cache the extended dictionaries. The linker will ignore extended dictionaries and proceed with the link.

Out of memory creating extended dictionary *Librarian error*

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.

Out of memory for block *block* *Linker error*

This error should not occur. If it does, call Borland Technical Support and give them the text of the message, including the block name.

Out of memory reading LE/LIDATA record from object module *Librarian error*

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.

Out of space allocating per module debug struct *Librarian error*

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.

Output device is full *Librarian error*

Usually this error means that no space is left on the disk.

Overload is now unnecessary and obsolete *Compiler warning*

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.

Overloadable operator expected *Compiler error*

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 a semicolon (**;**) cannot be overloaded.

Overloaded *function name* ambiguous in this context *Compiler error*

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. This error was issued because an overloaded function name has been used in some other context.

Overloaded function resolution not supported *IDE debugger error*

The only time an overloaded function name can be used without actually calling the function is when a variable or parameter if a appropriate type is initialized or assigned. In this case, an overloaded function name has been used in some other context.

Overloaded prefix 'operator *operator*' used as a postfix operator *Compiler warning*

It is now possible to overload both the prefix and postfix versions of the **++** and **--** operators. To allow older code to compile, Borland C++ uses the prefix operator and issues this warning whenever only the prefix operator is overloaded, but is used in a postfix context.

Parameter names are used only with a function body *Compiler error*

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
```

Parameter *number* missing name *Compiler error*

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.)

Parameter *parameter* is never used *Compiler warning*

The named parameter, declared in the function, was never used in the body of the function. This might or might 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.

***path* – path is too long** *Librarian error*

This error occurs when the length of any of the library file or module file's *path* is greater than 64.

Physical and virtual device drivers not supported *Linker error*

Pointer to structure required on left side of –> or –>* *Compiler error*

Nothing but a pointer is allowed on the left side of the arrow (**->**) in C or C++. In C++ a **–>*** operator is allowed.

Possible reference to undefined extern *xxxx::i* in module *module* *Linker warning*

Static data member has been declared but not defined in your application.

Possible use of *identifier* before definition *Compiler warning*

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 might assign the value before the program uses it.

Possibly incorrect assignment *Compiler warning*

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (that is, as part of an **if**, **while** or **do-while** statement). More often than not, this is a typographical error for the equality operator. If you want 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) ...
```

Public *symbol* in module *module1* clashes with prior module *module2* *Librarian error*

A public symbol can appear only once in a library file. A module that is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added. The command-line message reports the *module2* name.

Public *symbol* in module *filename* clashes with prior module *Librarian error*

A public symbol can appear only once in a library file. A module that is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added.

Public symbol *sym* defined in both module *mod1* and *mod2* *Librarian warning*

This warning occurs when two .OBJ files in the .OBJ file list both define the same public symbol. The first public symbol will override the second public symbol. This warning can be controlled with the **–wdup** switch.

Record kind *num* found, expected theadr or lheadr in module *filename* *Librarian error*

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.

Record length *len* exceeds available buffer in module *module* *Librarian error*

This error occurs when the record length *len* exceeds the available buffer to load the buffer in module *module*. This occurs when librarian runs out of dynamic memory.

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*.

Redefinition of *macro* is not identical *Compiler warning*

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.

Redefinition of target *filename* *MAKE error*

The named file occurs on the left side of more than one explicit rule.

Reference initialized with *type1*, needs lvalue of type *type2* *Compiler error*

A reference variable or parameter that is not declared constant must be initialized with an lvalue of the appropriate type. This error was issued either because the initializer wasn't an lvalue or because its type didn't match the reference being initialized.

Reference member *member* in class without constructors *Compiler error*

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.

Reference member *member* initialized with a non-reference parameter *Compiler error*

An attempt has been made to bind a reference member to a parameter in a constructor. Because the parameter object ceases to exist the moment the constructor returns, the reference member is then left referring to an undefined object. (This is a common mistake that causes crashes and erratic program behavior.)

Reference member *member* is not initialized *Compiler error*

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, because it has no way of knowing how to initialize the references.

Reference member *member* needs a temporary for initialization *Compiler error*

You provided an initial value for a reference type that was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Because there is no obvious place to store this temporary, the initialization is illegal.

Reference variable *variable* must be initialized *Compiler error*

This C++ object is declared as a reference but is not initialized. All references must be initialized at their point of declaration.

Register allocation failure *Compiler error*

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.

Repeat count needs an lvalue *IDE debugger error*

The expression before the comma in the Watch or Evaluate window must be a manipulable region of storage. For example, expressions like these are not valid:

```
i++, 10d
x = y, 10m
```

Resident Name Table is greater than 64K *Linker warning*

The maximum size of the Resident name table is 64K (in accordance with the industry-wide executable specification standard). The linker continues with the link but ignores any subsequent Resident names encountered during linking.

Restarting compile using assembly *Compiler warning*

The compiler encountered an **ASM** with an accompanying **–B** command-line option or **#pragma inline** statement. The compile restarts using assembly language capabilities.

Results are safe in file *filename* *Librarian warning*

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).

Rule line too long *MAKE error*

An implicit or explicit rule was longer than 4,096 characters.

Segment *segment* is in two groups: *group1* and *group2* *Linker warning*

The linker found conflicting claims by the two named groups. Usually, this happens only in assembly language programs. It means that two modules assigned the segment to two different groups.

Self relative fixup overflowed in module *module* *Linker warning*

This message appears if a self-relative reference (usually a call) is made from one physical segment to another. It usually happens only when employing assembler code, but can occur if you use the segment-naming options in the compiler. If the reference is from one code segment to another, you are safe. If, however, the reference is from a code segment to a data segment, you have probably made a mistake in some assembler code.

Side effects are not allowed *IDE debugger error*

Side effects such as assignments, ++, or — are not allowed in the debugger watch window. A common error is to use "x = y" (not allowed) instead of "x == y" to test the equality of "x" and "y."

Size of *identifier* is unknown or zero *Compiler error*

This identifier was used in a context where its size was needed. For example, a **struct** tag might only be declared (with the **struct** not defined yet), or an **extern** array might be declared without a size. If so, it's illegal to have 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.

Size of the type is unknown or zero *Compiler error*

This type was used in a context where its size was needed. For example, a **struct** tag might only be declared (with the **struct** not defined yet). If so, it's illegal to have 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.

sizeof may not be applied to a bit field *Compiler error*

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

sizeof may not be applied to a function *Compiler error*

sizeof can be applied only to data objects, not functions. You can request the size of a pointer to a function.

***identifier* specifies multiple or duplicate access** *Compiler error*

A base class can be declared **public** or **private**, but not both. This access specifier can appear no more than once for a base class.

Stack overflow *Run-time error*

The default stack size for Borland C++ programs is 45056 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 get this message only 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. Change the stack size by using **–S:*xxxx*** TLINK option.

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. This is usually only a minor disadvantage.
2. The function can no longer be reentrant. 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 OS/2 programs this is not a problem.

Statement missing ; *Compiler error*

The compiler encountered an expression statement without a semicolon following it.

Storage class *class* is not allowed here *Compiler error*

The given storage class is not allowed here. Probably two storage classes were specified, and only one can be given.

String type not allowed with this operand *MAKE error*

You have tried to use an operand that is not allowed for comparing string types. Valid operands are ==, !=, <, >, <=, and >=.

Structure passed by value *Compiler warning*

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.

Structure required on left side of . or .* *Compiler error*

The left side of a dot (.) operator (or C++ dot-star operator) must evaluate to a structure type. In this case it did not.

Style of function definition is now obsolete *Compiler warning*

In C++, this old C style of function definition is illegal:

```
int func(p1, p2)
int p1, p2;
{...}
```

Subscripting missing] *Compiler error*

The compiler encountered a subscripting expression that was missing its closing bracket. This could be caused by a missing or extra operator, or by mismatched parentheses.

Superfluous & with function *Compiler warning*

An address-of operator (**&**) is not needed with function name; any such operators are discarded.

Suspicious pointer conversion *Compiler warning*

The compiler encountered some conversion of a pointer that caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.

Switch selection expression must be of integral type *Compiler error*

The selection expression in parentheses in a **switch** statement must evaluate to an integral type (**char**, **short**, **int**, **long**, **enum**). You might be able to use an explicit cast to satisfy this requirement.

Switch statement missing (*Compiler error*

In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.

Switch statement missing) *Compiler error*

In a **switch** statement, the compiler found no right parenthesis after the test expression.

Table limit exceeded *Linker error*

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.

Target index of FIXUP is 0 in module *module* *Linker error*

This is a translator error.

Template argument must be a constant expression *Compiler error*

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.

Template class nesting too deep: 'class' *Compiler error*

The compiler imposes a certain limit on the level of template class nesting; this limit is usually exceeded only through a recursive template class dependency. When this nesting limit is exceeded, the compiler issues 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 code 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
```

Template function argument *argument* not used in argument types *Compiler error*

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.

Template functions may only have type-arguments *Compiler error*

A function template was declared with a non-type argument. This is not allowed with a template function because there is no way to specify the value when calling it.

Templates can only be declared at file level *Compiler error*

Templates cannot be declared inside classes or functions; they are allowed only in the global scope (file level).

Templates must be classes or functions *Compiler error*

The declaration in a template declaration must specify either a class type or a function.

Temporary used to initialize *identifier* *Compiler warning*
Temporary used for parameter *number* in call to *function* *Compiler warning*
Temporary used for parameter *number* *Compiler warning*
Temporary used for parameter *parameter* *Compiler warning*

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.

In the following example, 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);
```

Terminated by user *Linker error*

You canceled the link or *Ctrl+Break* was pressed.

The '...' handler must be last *Compiler error*

In a list of catch handlers, if the '...' handler is present, it must be the last handler in the list (that is, it cannot be followed by any more catch handlers).

The combinations '+*' or '*+' are not allowed *Librarian error*

It is not legal to add and extract an object module from a library in one action. The action probably desired is a '+-'.

The constructor *constructor* is not allowed *Compiler error*

Constructors of the form **X::(X)** are not allowed. The correct way to write a copy constructor is **X::(const X&)**.

The value for *identifier* is not within the range of an int *Compiler error*

All enumerators must have values that can be represented as an integer. You attempted to assign a value that is out of the range of an integer. In C++ if you need a constant of this value, use a **const** integer.

'this' can be used only within a member function *Compiler error*

In C++, **this** is a reserved word that can be used only within class member functions.

This initialization is only partly bracketed *Compiler warning*

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 can be used. When some of the optional braces are omitted, the compiler issues this warning.

Too few arguments in template class name *template* *Compiler error*

A template class name was missing actual values for some of its formal parameters.

Too few parameters in call *Compiler error*

A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.

Too few parameters in call to *function* *Compiler error*
A call to the named function (declared using a prototype) had too few arguments.

Too many commas on command-line *Linker error*
An invalid entry in the command-line was found.

Too many decimal points *Compiler error*
The compiler encountered a floating-point constant with more than one decimal point.

Too many default cases *Compiler error*
The compiler encountered more than one **default** statement in a single **switch**.

Too many error or warning messages *Compiler error*
A maximum of 255 errors and warnings can be set before the compiler stops.

Too many errors *Linker error*
The linker encountered more errors than the **–E** switch will permit.

Too many exponents *Compiler error*
The compiler encountered more than one exponent in a floating-point constant.

Too many initializers *Compiler error*
The compiler encountered more initializers than were allowed by the declaration being initialized.

Too many LNAMEs *Linker error*
TLINK has a limit of 256 LNAMES in a single .OBJ file.

Too many rules for target *target* *MAKE error*
MAKE can't determine which rules to follow when building a target because you've created too many rules for the target. For example, the following makefile generates this error message:

```
abc.exe : a.obj
   bcc -c a.c

abc.exe : b.obj

abc.exe : c.obj
   bcc -c b.c c.c
```

Too many segments *Linker error*
TLINK has a limit of 256 SEGDEFs in a single .OBJ file.

Too many storage classes in declaration *Compiler error*
A declaration can never have more than one storage class.

Too many suffixes in .SUFFIXES list *MAKE error*
The limit of 255 allowable suffixes in the suffixes list has been exceeded.

Too many types in declaration *Compiler error*
A declaration can never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum**, or **typedef-***name*.

Too much global data defined in file *Compiler error*
The sum of the global data declarations exceeds 64K bytes. Check the declarations for any array that might be too large. Also consider reorganizing the program or using **far** variables if all the declarations are needed.

Two consecutive dots *Compiler error*

Because an ellipsis contains three dots (**...**), and a decimal point or member selection operator uses one dot (**.**), two consecutive dots cannot legally occur in a C program.

Two operands must evaluate to the same type *Compiler error*

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 might be an error or you might merely need to cast one side to the type of the other.

Type *type* is not a defined class with virtual functions *Compiler error*

A dynamic_cast was used with a pointer to a class type that is either undefined or doesn't have any virtual member functions.

Note on type-mismatch errors: When compiling C++ programs, the following type-mismatch error messages 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 could also be due to many other reasons.

Type *type* may not be defined here *Compiler error*

Classes and enumerations may not be defined in certain places. For example, the return type specification of a function. The class or enum definition must be moved into a separate type declaration.

Type mismatch in default argument value *Compiler error*
Type mismatch in default value for parameter *parameter* *Compiler error*

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. See the previous note on type-mismatch errors.

Type mismatch in parameter *number* *Compiler error*

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 errors.

Type mismatch in parameter *number* in call to *function* *Compiler error*

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 errors.

Type mismatch in parameter *parameter* *Compiler error*

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 errors.

Type mismatch in parameter *parameter* in call to *function* *Compiler error*

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*** and the previous note on type-mismatch errors.

Type mismatch in parameter *parameter* in template class name *template* *Compiler error*
Type mismatch in parameter *number* in template class name *template* *Compiler error*

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 errors.

Type mismatch in redeclaration of *identifier* *Compiler error*

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. See the previous note on type-mismatch errors.

Type name expected *Compiler error*

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.

Type qualifier *identifier* must be a struct or class name *Compiler error*

The C++ qualifier in the construction `qual::identifier` is not the name of a **struct** or **class**.

Unable to create output file *filename* *Compiler error*

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.

Unable to create turboc.$ln *Compiler error*

The compiler cannot create the temporary file TURBOC.$LN because it cannot access the disk or the disk is full.

Unable to execute command: *command* *MAKE error*

A command failed to execute; this might be because the command file could not be found or was misspelled, because there was no disk space left in the specified swap directory, because the swap directory does not exist, or (less likely) because the command itself exists but has been corrupted.

Unable to execute command *command* *Compiler error*

TLINK or TASM cannot be found, or possibly the disk is bad.

Unable to open file *filename* *MAKE error*
Unable to open *filename* *Linker error*

This occurs if the named file does not exist or is misspelled.

Unable to open include file *filename* *Compiler error*

The compiler could not find the named file. This error can 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.

Unable to open *filename* for output *Librarian error*

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.

Unable to open include file *filename* *MAKE error*

MAKE could not find the named file. This error can 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.

Unable to open input file *filename* *Compiler error*

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.

Unable to open makefile *MAKE error*

The current directory does not contain a file named MAKEFILE or MAKEFILE.MAK, or it does not contain the file you specified with **–f**.

Unable to process debug information, disable tasm /zi option *Linker error*

This happens when you compile .C or .CPP code with debug information, generating assembler output, and then run TASM on the result with the /zi option. Do not use the /zi option. The compiler already put out the appropriate debug information.

Unable to redirect input or output *MAKE error*

MAKE was unable to open the temporary files necessary to redirect input or output. If you are on a network, make sure you have rights to the current directory.

unable to rename *filename* to *filename* *Librarian error*

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.

Undefined alias symbol *symbol* *Linker error*

An ALIAS definition record was encountered which specified a substitute public symbol for an external reference. The public symbol was never found. ALIAS records are generated by the assembler when the ALIAS directive is used.

Undefined label *identifier* *Compiler error*

The named label has a **goto** in the function, but no label definition.

Undefined structure *identifier* *Compiler warning*

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.

Undefined structure *structure* *Compiler error*

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.

Undefined symbol *identifier* *Compiler error*

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.

Unexpected } *Compiler error*

An extra right brace was encountered where none was expected. Check for a missing **{**.

Unexpected char *X* in command line *Librarian error*

The librarian encountered a syntactical error while parsing the command line.

Unexpected end of file *MAKE error*

The end of the makefile was reached without a temporary inline file having been closed.

Unexpected end of file in comment started on *line number* *CompilerMAKE error*

The source file ended in the middle of a comment. This is normally caused by a missing close of comment (*/).

Unexpected end of file in conditional started on line *line number* *MAKE error*

The source file ended before the compiler (or MAKE) encountered an **!endif**. The **!endif** was either missing or misspelled.

Union cannot be a base type *Compiler error*

A union cannot be used as a base type for another class type.

Union cannot have a base type *Compiler error*

A union cannot be derived from any other class.

Union member *member* is of type class with constructor *Compiler error*
Union member *member* is of type class with destructor *Compiler error*
Union member *member* is of type class with operator= *Compiler error*
A union cannot contain members that are of type **class** with user-defined constructors, destructors, or operator=.

Unions cannot have virtual member functions *Compiler error*
A union cannot have virtual functions as its members.

Unknown assembler instruction *Compiler warning*
The compiler encountered an inline assembly statement.

Unknown command line switch *X* ignored *Librarian warning*
A forward slash character (/) was encountered on the command line or in a response file without being followed by one of the allowed options.

Unknown Goodie *Linker error*
An unsupported option was supplied to the command-line linker. See the documentation for currently supported Goodies (options).

Unknown language, must be C or C++ *Compiler error*
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 having the compiler rename it like this:

```
extern "C" int pascal func( /*...*/ );
```

A C++ (possibly overloaded) function can be declared Pascal and allow the usual compiler renaming (to allow overloading) like this:

```
extern int pascal func( /*...*/ );
```

Unknown option -> *option* *Linker error*
A forward slash character (/), hyphen (–), or DOS switch character was encountered on the command line or in a response file without being followed by one of the allowed options. You might have used the wrong case to specify an option.

Unknown preprocessor directive: *identifier* *Compiler error*
The compiler encountered a # character at the beginning of a line, and the name following was not a legal directive name or the rest of the directive was not well formed.

Unknown preprocessor statement *MAKE error*
A ! character was encountered at the beginning of a line, and the statement name following was not **error**, **undef**, **if**, **elif**, **include**, **else**, **endif**, **ifdef**, or **ifndef**.

Unreachable code *Compiler warning*
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 that cannot fall through.

Unresolved external *name* referenced from module *module* *Linker error*
This is the actual text of the message for "Undefined symbol *sym* in module *module*."

Unsupported option *string* *Linker error*
You have specified an invalid option to the linker.

Unterminated string or character constant *Compiler error*

The compiler found no terminating quote after the beginning of a string or character constant.

Use '> >' for nested templates instead of '>>' *Compiler warning*

Whitespace is required to separate the closing ">" in a nested template name, but since it is a common mistake to leave out the space, the compiler accepts a ">>" with this warning.

Use . or –> to call *function* *Compiler error*

You tried to call a member function without giving an object.

Use . or –> to call *member*, or & to take its address *Compiler error*

A reference to a nonstatic class member without an object was encountered. Such a member must be used with an object, or its address must be taken with the **&** operator.

Use :: to take the address of a member function *Compiler error*

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, rather than 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.)

Use /e with TLINK to obtain debug information from library *Librarian warning*

The library was built with an extended dictionary and also includes debugging information. TLINK will not extract debugging information if it links using an extended dictionary, so to obtain debugging information in an executable from this library, the linker must be told to ignore the extended dictionary using the **/e** switch. *Note*: The IDE linker does *not* support extended dictionaries; therefore no settings need to be altered in the IDE.

Use of : and :: dependents for target *target* *MAKE error*

You have tried to use the target in both single and multiple description blocks (using both the **:** and **::** operators). Examples:

```
filea: fileb
filea:: filec
```

Use qualified name to access nested type *type* *Compiler warning*

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 their class' scope. To allow older code to compile, whenever such a name is uniquely defined in one single class, Borland C++ allows its usage without **class::** and issues this warning.

Using based linking for DLLs may cause the DLL to malfunction

This warning occurs if you use the **/B** switch when linking a DLL. In almost every case, this is an error that will prevent the application from running. This warning can be controlled with the **–w** switch.

Value of type void is not allowed *Compiler error*

A value of type **void** is really not a value at all, and thus cannot 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.

Variable *variable* has been optimized. *IDE debugger error*

You have tried to inspect, watch, or otherwise access a variable that the optimizer removed. This variable is never assigned a value and has no stack location.

Variable *identifier* is initialized more than once *Compiler error*

This variable has more than one initialization. It is legal to declare a file level variable more than once, but it can have only one initialization (even if two are the same).

***Constant* variable *variable* must be initialized** *Compiler error*

This C++ object is declared **const**, but is not initialized. Because no value can be assigned to it, it must be initialized at the point of declaration.

VIRDEF name conflict for *function* *Compiler error*

The compiler must truncate mangled names to a certain length because of a name length limit that is imposed by the linker. This truncation might (in rare cases) cause two names to mangle to the same linker name. If these names happen to both be **VIRDEF** names, the compiler issues this error message. The simplest workaround for this problem is to change the name of *function* so that the conflict is avoided.

'virtual' can only be used with member functions *Compiler error*

A data member has been declared with the **virtual** specifier; only member functions can be declared **virtual**.

Virtual function *function1* conflicts with base class *base* *Compiler error*

The compiler encountered a virtual function that has the same argument types as a function in its base class, but the two functions have different return types. This is illegal.

Virtual specified more than once *Compiler error*

The C++ reserved word **virtual** can appear only once in a member function declaration.

void & is not a valid type *Compiler error*

A reference always refers to an object, but an object cannot have the type **void**. Thus the type **void** is not allowed.

Void functions may not return a value *Compiler warning*

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.

function* was previously declared with the language *language *Compiler error*

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.

While statement missing (*Compiler error*

In a **while** statement, the compiler found no left parenthesis after the **while** keyword.

While statement missing) *Compiler error*

In a **while** statement, the compiler found no right parenthesis after the test expression.

This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.

Write error on file *filename* *MAKE error*

MAKE couldn't open or write to the file specified in the makefile. Check to ensure that there's enough space left on your disk, and that you have write access to the disk.

Wrong number of arguments in call of macro *mac* *Compiler error*

Your source file called the named macro with an incorrect number of arguments.

Index

N

O

P

Q

R

S

T

U

V

W

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # BCP1415WW21773 • BOR 7003