Powersoft®

)pen Tools from Sybase, Inc.

ɂowerBuilder

ɔ++ Class Builder

/ersion 6

Power Builder®

AB0878

October 1997

Contents

About This Book

Subject

This book describes how to use the PowerBuilder C++ Class Builder to quickly and easily develop C++ classes for use with PowerBuilder. With the Class Builder you create PowerBuilder user objects in the C++ language, compile them with the Watcom C++ compiler Version 11.0, and link them into compact and efficient Windows DLLs.

Audience

This book is for programmers who want to create their own compiled functions using C++ and then make them available to PowerBuilder applications. It assumes you are:

- An experienced PowerBuilder developer
- An experienced C or C++ programmer
- Familiar with Windows programming concepts, particularly DLLs and the Windows API

Software required

The C++ Class Builder is available in the User Object painter of the PowerBuilder Enterprise Edition, beginning with Version 4.0. (C++ user objects created with the Class Builder are not compatible with earlier versions of PowerBuilder.)

CHAPTER 1 Introduction

About this chapter

This chapter introduces the PowerBuilder C++ Class Builder and the Watcom Integrated Development Environment (IDE).

The chapter also discusses the usage and construction of Windows dynamic link libraries (DLLs).

Contents

Introducing the C++ Class Builder

The C++ Class Builder is a revolutionary set of enabling tools that dramatically simplifies the process of compiling PowerScript user object functions into a Windows DLL. With the C++ Class Builder, you create user object functions and variable definitions within PowerBuilder, generate the corresponding C++ classes and skeleton code, then write the actual function logic in C++. This strategy enables you to take full advantage of PowerBuilder's object-oriented capabilities and the power and efficiency of the C++ language.

What the Class Builder includes

The C++ Class Builder contains everything you need to create and debug a complete Windows DLL, including:

- ♦ The Watcom Integrated Development Environment (IDE), which acts as a workplace shell for your C++ coding and debugging activities
- ♦ A powerful source code editor that takes advantage of the Windows Graphical User Interface (GUI)
- ♦ The Watcom C/C++ optimizing compiler Version 11.0 for Windows DLLs
- ♦ A flexible, user-configurable object code linker
- ♦ A versatile, easy-to-use source code debugger
- ♦ The PowerBuilder C++ Interface Generator that creates skeleton C++ source code from your PowerScript user object functions

What the Class Builder does

The C++ Class Builder automates much of the C++ coding normally associated with a Windows DLL. This includes function prototypes and function and class declarations, as well as initialization and exit routines. This frees you to concentrate on the logic of your functions rather than the mechanics of constructing the DLL.

What is a DLL?

A **dynamic link library** (DLL), like a static code library, is a collection of functions.

When an application uses functions from a **static library**, the library functions referenced by the application become part of the executable module. This form of linking is called **static linking**.

When an application uses functions from a **dynamic link library**, the library functions referenced by the application are not included in the executable module. Instead, the executable module contains references to these functions that are resolved at execution time. This form of linking is called **dynamic linking**.

Advantages

These are some of the advantages of using dynamic link libraries over standard libraries:

- **Reduced time and disk space** Functions in dynamic link libraries are not linked into your program. Only references to the functions in dynamic link libraries are placed in the program module (these references are called import definitions). As a result, the linking time is reduced and disk space is saved. If many applications reference the same dynamic link library, the savings in disk space can be significant.
- **Updating without relinking** Since program modules only reference dynamic link libraries and do not contain the actual executable code, a dynamic link library can be updated without relinking your application. When your application is executed, it will use the updated version of the dynamic link library.
- **Sharing code and data** Dynamic link libraries also allow sharing of code and data between the applications that use them. If many applications that use the same dynamic link library are executing concurrently, the sharing of code and data segments improves memory utilization.

Using DLLs with PowerBuilder

A Windows DLL is a library of related functions that can be called from a PowerBuilder application. Because the DLL is separate from the PowerBuilder application that calls it, many such programs can share the same function code without each of them having to include that code in its executable.

Advantages

- **Separate maintenance** The DLL functions can be maintained separately from the compiled PowerBuilder applications that use them. Programs that rely heavily on DLLs for their functionality are cheaper to maintain, since changes to a particular function can be made in the DLL and distributed more economically than recompiling and redistributing the whole application.
- **Extended functionality** The functionality of programs can be extended and enhanced through the use of carefully designed DLLs. While PowerBuilder offers a rich palette of functions, you may want to call additional Windows API functions. By packaging the necessary API calls in a DLL, along with code to allocate and manipulate Windows resources, you can make powerful, *encapsulated* functions available to any PowerBuilder application.
- **Improved performance** The PowerScript language is fast and powerful enough for most Windows applications, but for programs that are calculation-intensive (real-time graphics display, matrix manipulation, and so on), compiled C++ may be a better choice of language. Functions written in C++ can be made into a DLL and called from your PowerBuilder application.

DLL components

DLLs created for Windows Version 3.0 consist of three required functions: Entry, Main, and Exit. Those created for Windows Version 3.1 and Windows 95 require only Entry and Main; Windows NT requires only DLLMain; the Exit procedure is optional but is almost always included. Watcom C/C++ provides a DLL Entry function (called **LibEntry**) and an exit procedure (called **WEP)** in its linker libraries. PowerBuilder provides **LibMain** as C++ source code when you build a C++ user object.

A DLL will also contain **programmer-supplied functions**, to do the actual work for which you've built the DLL.

Required functions

LibMain

The LibMain function performs initialization specific to a particular DLL, and is linked into your DLL automatically by the Make utility.

Since LibMain is only called once each time the DLL is loaded, it cannot perform functions in support of individual instances that use the DLL. The kind of initialization commonly done in LibMain includes:

- Initializing data structures that will be managed by the DLL
- Loading bitmaps, icons, and other resources
- Registering for use by the calling application, such as window classes that may be used to implement custom controls

WEP

WEP (Windows Exit Procedure) is the last function of a DLL that executes before Windows unloads the DLL from memory. If any resources are allocated to the DLL in the LibEntry or LibMain functions, they should be deallocated in the WEP.

Programmer-supplied functions

This is where you put the real functionality of the DLL. Some of these functions will have entry points that are available to a calling application or another DLL; these are known as **exported functions**. Others will have entry points that can only be called from functions within the same DLL; these are known as **internal functions**.

Exported functions

You should not have to deal directly with exported functions, since PowerBuilder shields you from the complexities of the DLL internals. However, for the sake of completeness, here is a brief explanation.

Exported functions are the handles by which the rest of the world gets at the functionality of your DLL. Also known as entry points, exported functions can be called from a PowerBuilder application. Exported functions generally offer high-level services to the applications and DLLs that call them. They do this by calling internal functions that perform the operations required to support those services.

Exported functions for use by PowerBuilder must be declared as **__far** and must use the **__PASCAL** calling convention. This avoids passing parameters in the CPU registers. The registers are used by the prolog code of an exported function, and passed parameters would be overwritten. The __far and __PASCAL conventions are ensured by including the file **pbdll.h** in your source code. PowerBuilder does this for you when it builds the C++ skeleton file from your user object. The file pbdll.h contains a macro definition for PB_EXPORT, which is added to your exported function declarations.

Internal functions

Internal functions are the workhorses of a DLL. They are called to perform operations needed by other DLL functions—operations such as initialization and cleanup in LibMain and WEP, and function logic that you code.

Because internal functions are not known to the outside world (their entry points are not exported), they can only be called by other functions within the same DLL. They cannot be called from PowerBuilder. Much like subroutines in an application program, internal functions support the processing done by the program's main routines.

Finally, as in application programming, the use of internal functions simplifies and modularizes the coding of a DLL.

Memory models

Memory models limit the size of your program and data by describing the number of bytes used to address data and call functions in your program. The memory model is comprised of a **code model** and a **data model**.

Code models

There are two code models:

- **Small code model** In this model all calls to functions are made with **near calls**. Hence, in a small code model, all code comprising your program, including library functions, must be less than 64K.
- **Big code model** In this model all calls to functions are made with **far calls**. This model allows the size of the code comprising your program to exceed 64K.

Data models

There are three data models:

- **Small data model** In this model all references to data are made with **near pointers**. Hence, in a small data model, all data comprising your program must be less than 64K.
- **Big data model** In this model all references to data are made with **far pointers**. This removes the 64K limitation on data size imposed by the small data model. When a far pointer is incremented, only the offset is adjusted. Watcom C/C++ assumes that the offset portion of a far pointer will not be incremented beyond 64K. The compiler will assign an object to a new segment if the grouping of data in a segment will cause the object to cross a segment boundary. Implicit in this is the requirement that no individual object exceed 64K bytes. For example, an array containing 40,000 integers does not fit into the big data model. An object such as this should be described as huge.

- **Huge data model** In this model all references to data are made with **far pointers**, as in the big data model. However, in the huge data model, incrementing a far pointer will adjust the offset and the segment if necessary. The limit on the size of an object pointed to by a far pointer imposed by the big data model is removed in the huge data model.

In Class Builder

The PowerBuilder C++ Class Builder makes DLLs based on the large memory model only:

Memory model	Code model	Data model	Default mode pointer	Default data pointer
Tiny	Small	Small	Near	Near
Small	Small	Small	Near	Near
Medium	Big	Small	Far	Near
Compact	Small	Big	Near	Far
Large	*Big*	*Big*	*Far*	*Far*
Huge	Big	Huge	Far	Huge

CHAPTER 2

Creating the C++ Source Code

About this chapter

This chapter describes how to build a PowerBuilder C++ user object and convert it into skeleton source code to be compiled.

The chapter includes discussions of instance versus shared variables and access levels of public, private, and protected as used by PowerBuilder.

Contents

Creating C++ source code

To create C++ source code you need to perform these basic steps:

1. Create the user object
2. Declare user object functions
3. Declare instance and shared variables
4. Convert from PowerBuilder to C++ source code
5. Troubleshoot the conversion

Each of the steps is explained below.

Step 1: creating the user object

The code that handles event processing is placed in event scripts. Supporting routines, business logic, and so on are usually put in functions and subroutines. These functions may be associated with objects in PowerBuilder, such as windows, menus, and user objects.

Many useful tasks can be handled by user objects, from performing complex calculations and processing forms or dialog boxes to encapsulating business rules. The Class Builder gives you the advantage of coding these functions as fast, compact DLLs and still enables you to call these functions as if they resided in a PowerBuilder user object.

For those familiar with C++ programming, the user object is analogous to a C++ class. It consists of **data members** (declared as instance or shared variables) and **methods** (functions) that operate on that data. In fact, the definition of your user object, as specified in the User Object painter, goes on to become the C++ class definition in your DLL source code. This definition resides in the header file with the file extension hpp.

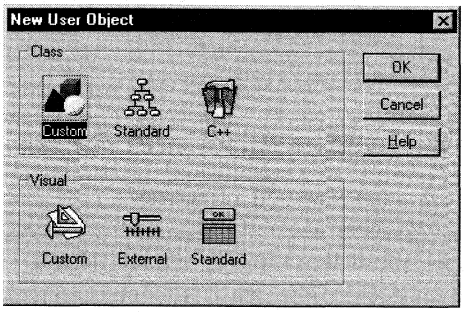
You create a C++ class from within PowerBuilder by first creating a special type of class user object called a **C++ user object**. This serves as a repository for the exported function and data declarations that will make up this particular C++ class in your DLL.

The example

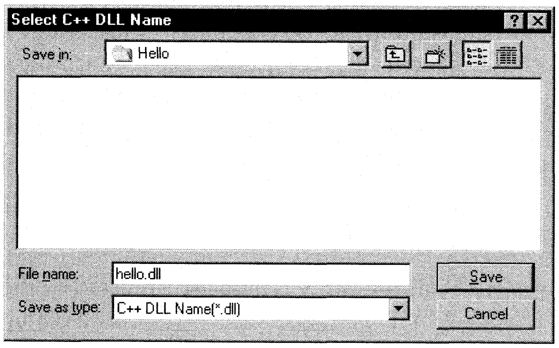
The example in the steps in this section begins building a very simple application called Hello World. The work is done by two functions declared in one user object. The functions are coded in C++ to create a DLL. We've named the DLL *hello.dll.*

❖ **To create a new C++ user object:**

1 In the User Object painter, select New and then double-click the C++ icon.



The Select C++ DLL Name dialog box appears:



Save in: Hello

File name: hello.dll

Save as type: C++ DLL Name(*.dll)

2 Specify a destination path and filename for the DLL you will create. If you want, you can click Cancel at this point and name the target DLL later, before entering the IDE.

In the Hello World program included on your distribution disk, the C++ user object contains a fully qualified DLL name:

```
string LibraryName="c:\Pwrs\Watcnt\samples\hello\
   hello.dll"
```

This is the path specified in the Select C++ DLL Name dialog box and where the application would try to find the Hello DLL. To simplify your installation, we have changed the line specifying the DLL path to:

```
string LibraryName=".\hello.dll"
```

The program will find the DLL as long as it is in the current directory or its directory is on the DOS path.

Notice that a new button is added to the User Object PainterBar. You will use it to enter the IDE, where you will write the C++ code for your user object.

Step 2: creating PowerBuilder functions for the user object

Once you have created a C++ user object, the next step is to define the user object functions it should contain. The PowerBuilder C++ User Object painter will use these definitions to build a skeleton DLL for you, containing one function declaration for each user object function you define. In the DLL, these are known as exported functions.

About functions

PowerBuilder is a powerful tool for creating visual frontends for a variety of databases. With PowerBuilder you can design a graphical environment through which users interact with the underlying program logic. Much of this logic does not need to be directly accessible to the user, but must do the work under the covers. Usually, for the sake of efficiency and modularity, it's a good idea to break up code into manageable sections, or **functions**.

In general, PowerBuilder processing is divided into two types: **event scripts** and **functions.**

- **Event scripts** These are triggered by the occurrence of a particular Windows event, such as clicking a button, pressing a key, opening or closing windows, editing the contents of a DataWindow, to name a few. The code you write in scripts should be more or less limited to handling those events. Additional processing that is not directly concerned with the event itself (even though it may occur in response to the event) should be placed in a function.
- **Functions** These are further divided into two types, **functions** and **subroutines**. The only difference to the programmer is that functions return a value and subroutines do not.

Naming conventions

You might want to adopt a naming convention to distinguish global functions from object-level functions. Doing this makes it easier for you to identify which are which, and which functions are meant to be compiled as DLLs.

Here is one suggestion:

Type of function	Name prefix
Global	f_
Window level	wf_
Menu level	mf_
User object level	uf_
C++ user object level	cf_
Application object level	af_

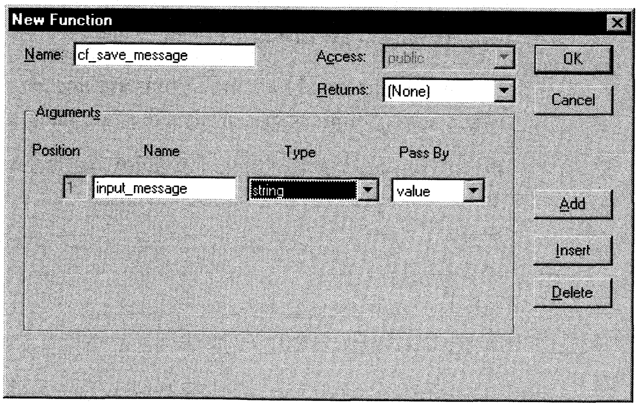
❖ **To create the functions that will be exported to the new C++ class:**

1. Select Declare>User Object Function from the menu bar.
   or
   Click the right mouse button and select User Object Functions from the popup menu.

   The Select Function In User Object window appears.

2. Select New.

   The New Function dialog box appears:



Name: cf_save_message
Access: public
Returns: (None)
Arguments

Position	Name	Type	Pass By
1	input_message	string	value

3. Enter the function definitions listed below. Enter the function names carefully; once you have saved a function definition, the only way it can be renamed is by deleting and redefining it.

Function name	Return type	Parameter type	Parameter name
cf_save_message	none	string	input_message
cf_display_message	none	none	none

Notice that when you click OK after entering the definition for each function, PowerBuilder returns you to the User Object painter. This is unlike the definition for a non-C++ user object, where you would go to the User Object Function painter.

C++ user objects
You don't write program logic for a C++ user object function within PowerBuilder—you only declare the function and its data. You will code the logic for this function later, within the Watcom IDE.

Step 3: declaring variables

Like functions and their parameters, all data members of a C++ class must be explicitly declared before the class can be instantiated. In order to have PowerBuilder create the data member declarations for you in C++, you must first declare them in the User Object painter.

Shared variables

If you want the value of a data member to be shared among all instances of its class, declare it as **shared**. A shared variable will have the same value to all scripts within all instances of the class.

Shared variables always have **private** access. This means they can only be accessed by objects of exactly the same class as that in which they are defined.

For example, say you have a window called w_main in which you define a shared integer variable called *mainvar*:

```
// In window w_win1 of type w_main...
shared:
integer mainvar
mainvar = 20
```

Only instances of the window type w_main can access it:

```
// In window w_win2 of type w_other...
   integer what
   what = mainvar
```

```
// The value of 'what' is undefined because the
// value of mainvar is only known to windows of the
// w_main class.
```

and they will all see the same variable:

```
// In window w_win3 of type w_main...
   integer what
   what = mainvar
// the value of 'what' is 20, as assigned in w_win1.
```

If the value of mainvar is changed in one instance of w_main, all other instances will see the changed value. Shared variables are also **static**. That means the value of a shared variable is maintained even when all instances of its parent object are closed. In the above example, if an instance of class w_main set mainvar = 20, and then all instances of w_main were closed, the next time w_main was instantiated, mainvar would still be 20.

Instance variables

If you want each instance of the class to keep its own copy of a data member, each with potentially different values, you should declare it as an **instance** variable.

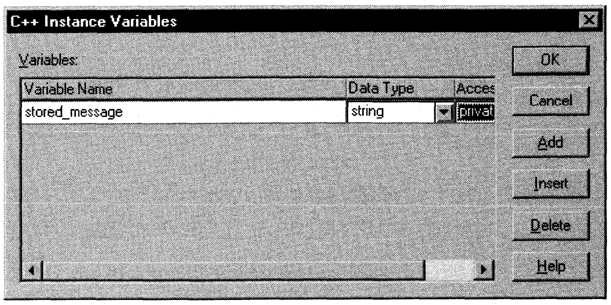
Instance variables declared in the user object as private or protected will reside in the data space of the C++ executable. They can be referenced directly by the C++ program, but not by any PowerBuilder scripts or functions. Private data members are known only to the C++ class in which they are defined. Protected data members can be referenced within the class in which they are defined, as well as any classes inherited from it.

❖ To declare the data member for the Hello World example:

1 In the User Object painter, select Declare>Instance Variables from the menu bar.
or
Right-click and select Instance variables from the popup menu.

The C++ Instance Variables dialog box appears.

2 Enter the following data declaration:



Variable Name	Data Type	Acces
stored_message	string	privat

3 Click OK.

4 Select File>Save or Save As from the menu bar.

5 Give the new user object the name uo_greeting.

6 Click OK to save.

You have now entered all the information necessary to produce a skeleton C++ source file from the uo_greeting user object.

Step 4: converting from PowerBuilder to C++ source code

The process of converting your PowerBuilder C++ user object to C++ source code is as simple as pushing a button. When you invoke the IDE, PowerBuilder recreates the user object you have just defined, adding external function calls that will speak to the functions in your DLL. In addition, PowerBuilder generates four files of C++ skeleton source code. You will edit this code to add the logic for your exported DLL functions.

❖ **To create the C++ skeleton files from your C++ user object:**

1 Start PowerBuilder with the Hello World application active.

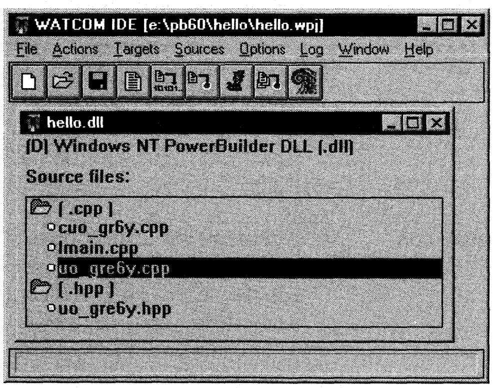
2 In the User Object painter, select uo_greeting if it is not already active.

3 Do one of the following to start the IDE:

Click the IDE button on the User Object PainterBar.
or
Select Design>Invoke C++ Editor from the menu bar.
or
Right-click anywhere in the User Object painter and select Invoke C++ Editor from the popup menu.

Tip
Most menu items on the Design and Declare menus are available by right-clicking in the User Object painter.

Once you are in the IDE, your screen should look like this:



WATCOM IDE [e:\pb60\hello\hello.wpj]

File Actions Targets Sources Options Log Window Help

hello.dll

[D] Windows NT PowerBuilder DLL [.dll]

Source files:

[.cpp]
- cuo_gr6y.cpp
- lmain.cpp
- uo_gre6y.cpp

[.hpp]
- uo_gre6y.hpp

Step 5: troubleshooting the conversion

On the rare occasion that the conversion from user object to C++ source code may fail, you will receive an error message. Such a failure is usually caused by modifying the C++ class *outside* PowerBuilder, changing code inside the cover function file, changing code between the \\$PB comment lines in the skeleton files, and so on.

FOR INFO For a list of error messages and explanations, see Appendix A, "Conversion Error Messages".

About the C++ files

When you build a C++ user object, PowerBuilder creates four source files for you. The examples in this chapter build a user object called uo_greeting and turn it into an executable called hello.dll. The corresponding files created by PowerBuilder are listed below:

Source filename	Description
uo_gre6Y.cpp	Contains the user object function declarations. This is the skeleton source code of your C++ user object functions *This is the file you modify by adding your own function code*
cuo_gr6Y.cpp	Contains cover function declarations that are called from the PowerBuilder user object and that correspond to the two user object functions cf_save_message and cf_display_message. PowerBuilder cannot directly call the C++ class functions you create in an external DLL. The functions in cuo_gr6Y.cpp act as a buffer between the user object and the DLL *This is a utility file used by PowerBuilder and should not be modified*
uo_gre6Y.hpp	Contains the instance variable declarations from the user object, as well as prototypes for each of its functions. This is the skeleton source code of your C++ user object variable declarations. It also includes the function prototypes (forward declarations) required by C++. This file is #INCLUDEd in uo_gre6Y.cpp when it is compiled, making these declarations available to your C++ code *You may modify this file if you need to add internal functions or data declarations to your DLL*
lmain.cpp	Contains the LibMain and WEP functions that are compiled into every C++ user object. It is created by the PowerBuilder C++ interface generator and consists of simple source code for LibMain and WEP *You would not normally change this file*

These files are compiled and linked under the control of the Make utility in the IDE. The result is a Windows DLL that can be called from a PowerBuilder application.

FOR INFO For the complete C++ source code files, see Appendix B, "The C++ Source Code".

Caution on modifying code

The C++ class definition (uo_gre6Y.hpp in our example) and skeleton file (uo_gre6Y.cpp) must be modified carefully. Any changes made on or between the two lines starting with //PB could corrupt these files and cause the code generation process to fail. If you return to PowerBuilder to make changes to your user object and enter the IDE again, all code enclosed within the //PB comments gets regenerated, destroying any changes you have made.

Any function code added to the C++ skeleton outside the //PB comments will be maintained, even if the user object is changed in PowerBuilder and resaved. This allows changes to your user object without losing the C++ code you've already written.

The C++ cover function file, cuo_gr6Y.cpp, should not be changed. It contains interface code only, and no user-serviceable parts. It is regenerated every time you save the PowerBuilder user object, so any changes you make here will be destroyed. Changes made here can cause unpredictable errors when you attempt to call the user object DLL from PowerBuilder.

C/C++ programming notes

This section deals with issues of compatibility between the C++ language and PowerScript. The topics covered include object-oriented principles such as inheritance and function overloading, as well as specific information about reserved words, naming conventions, and data type equivalency.

Inheritance

In PowerBuilder, objects such as windows, menus, and user objects can all be inherited. Likewise in C++, any class you define can use inheritance. Furthermore, the attributes of a C++ class can be inherited from several ancestors through a technique called **multiple inheritance**.

Multiple inheritance
Unless you exercise extreme care in your design, this technique can quickly add complexity and confusion to your program.

PowerBuilder imposes a limit of one direct ancestor per descendant, or **single inheritance**. This limit applies to C++ classes created from user objects.

How inheritance works

The C++ Class Builder implements class inheritance in two steps:

1. You create an inherited C++ user object. This is done by clicking the Inherit button in the New User Object dialog box and choosing an existing C++ user object to inherit from. You proceed as usual to populate this user object with function and data member definitions.
2. When PowerBuilder generates the skeleton C++ files for the inherited user object, it creates a class that is based on the ancestor class and #INCLUDEs the header file (hpp) for that ancestor within the new class. In this way, all functions and data members with access level of public or protected are now available to the descendent class.

Tip
All members of a particular class hierarchy (that is, all classes that are descended from a particular ancestor) must reside in the same DLL.

Function overloading

This is an object-oriented programming technique that enables you to call many functions by the same name. The function that will be executed depends on the object calling it and the type of data you are passing to it.

Example

Function prototype	Description
void print (char * my_string)	Calls a print function dedicated to printing strings (arrays of type char)
void print (double pay_info)	Calls a function that prints formatted decimal numbers
bool print (ref struct_def my_struct)	Calls another print function that takes a pointer to a structure as input and returns a boolean flag

You can overload function names in PowerBuilder, however, there is no way for PowerBuilder to create overloaded names in the DLL. If you were to create such overloaded functions in the DLL source code yourself, PowerBuilder would have no way to call them.

It is possible, however, to use function overloading within your DLL for nonexported functions. In this case, you must define the functions manually, either in the C++ skeleton file or in another source file that will be linked with it. Since you do not declare nonexported functions in PowerBuilder anyway, this should not pose a problem. But be aware: you should not overload function names that are exported to PowerBuilder.

Of course, once you are in an internal function, all the power of Windows DLLs is available to you. From here you can call any function in the Windows API, other DLLs you have written, or routines in any one of thousands of commercially available third-party DLLs and custom controls.

Reserved words

Avoid the following C/C++ keywords and symbols in the names of data members and arguments to PowerBuilder functions. These names will be copied into the DLL skeleton, and will likely confuse the compiler.

Standard C/C++ keywords and symbols

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Additional WATCOM C/C++ keywords

based	far	huge	near	saveregs	segname
cdecl	far16	interrupt	Packed	Seg16	self
export	fortran	loadds	pascal	segment	syscall

Reserved symbols

!	(	|	:	/	>>	*=	&=
#	)	~	“	->	<=	/=	^=
$	-	[	<	++	>=	%=	|=
%	+	]	>	--	==	+=	::
^	=	\	?	.*	!=	-=	##
&	{	;	,	->*	&&	<<=	
*	}	‘	.	<<	||	>>=	

In addition, you may not use names beginning with the following combinations of characters:

__	Double underscore
_C	Underscore followed by any uppercase character

How PowerBuilder builds C++ user object names

You may have noticed that the filenames of your C++ skeleton source code look a little strange. With respect to filenames, the C++ interface generator must do two things:

- Ensure that from each user object name it creates a unique, eight-character DOS filename.
- Prefix the cover function file with the character c.

Here is how PowerBuilder names the cpp and hpp files it creates from your user object. If the original user object name is six characters or shorter in length, it is used, as is, for the C++ filenames. Otherwise, the C++ skeleton source filename starts with the first six characters of the original user object name. Characters seven and eight are an encryption of the user object name and are appended to the first six characters. Finally, the extension cpp is added.

For example, the two-character encryption of uo_greeting is 6y. This is appended to the first six characters to form the eight-character name uo_gre6Y.cpp.

The name for the user object header file is derived in the same manner, except that it gets an extension of hpp.

The C++ cover function file starts with the letter c followed by the first five characters from the original user object name. If the original name is more than five characters long, the last two characters are an encryption of the entire original name, as above. The extension cpp is added. For example, uo_greeting becomes cuo_gr6Y.cpp.

More about naming conventions
While it is wise to prefix user object names with *uo* or *uo_* to make them easy to spot, these characters take up valuable real estate in the DOS filename. Use them where you can, but recognize they add no meaning when trying to identify a particular user object.

Equivalent PowerBuilder and C++ data types

C++ type	PowerBuilder type	Description
BOOL	BOOLEAN	2-byte signed integer

C++ type	PowerBuilder type	Description
WORD	UINT	2-byte unsigned integer
DWORD	ULONG	4-byte unsigned integer
HANDLE	UINT	2-byte unsigned integer
HWND	UINT	2-byte unsigned integer
LPINT	STRING	4-byte FAR pointer
LPWORD	STRING	4-byte FAR pointer
LPLONG	STRING	4-byte FAR pointer
LPDWORD	STRING	4-byte FAR pointer
LPVOID	STRING	4-byte FAR pointer
LPVOID	CHAR	4-byte HUGE pointer
BYTE	CHAR	1-byte
CHAR	CHAR	1-byte
CHAR [10]	CHAR [10]	array of 10-bytes
INT	INT	2-byte signed integer
UNSIGNED INT	UINT	2-byte unsigned integer
LONG	LONG	4-byte signed integer
UNSIGNED LONG	ULONG	4-byte unsigned integer
DOUBLE	DOUBLE	8-byte double precision floating point number
DOUBLE	DECIMAL	8-byte double precision floating point number
FLOAT	REAL	4-byte single precision floating point number
N/A	TIME	Date & time structure
N/A	DATE	Date & time structure
N/A	DATETIME	Date & time structure

CHAPTER 3

Compiling and Linking the C++ Class

About this chapter

This chapter describes the tools and techniques you use to produce a finished DLL from your PowerBuilder C++ source code. The tools are part of the Watcom IDE and include the editor, the Make utility, and the debugger.

Contents

IDE overview

The IDE acts as a *launching pad* for the various tools used to create and test a DLL. These include an editor, a compiler, a linker, and a debugger.

When you install the C++ Class Builder, a new button is added to the User Object PainterBar. Clicking it opens the IDE.

IDE project
Everything you do in the IDE is done within a project. The project acts as a logical container for all your source files, resource files, object files, Make utility files, executables, and so on.

Executable entities such as DLLs that are built in the IDE are called **targets** and together they make up a project. A project file is essentially a file list used by the IDE to keep track of the source code in the current project. Project files have the file extension of wpj. Target files are always of type DLL, and are made from source code files with the extensions cpp and hpp.

The editor

The editor included in the C++ Class Builder is the Windows-based Watcom Editor, Version 11.0. Most developers have their favorite C++ editor; but the C++ Class Builder includes one just in case you don't.

The editor is configured to fit into the Windows environment. It contains a toolbar and menu items. It can use proportional fonts. It also contains drag and drop toolbars or palettes where you make choices and then simply drag them to the elements to which you want to apply them.

For detailed instructions on using the editor, see the editor's online Help. Select Actions>Edit Text from the IDE main menu. When the editor starts, select Help.

The Make utility

The Make utility converts your source code into an executable DLL file. It does this by first compiling your source code into object files using the Watcom C/C++ 11.0 optimizing compiler. Next, one or more object files within your project are linked to form an executable DLL, using the Watcom linker.

The debugger

The debugger is a powerful debugging tool that helps you analyze your programs and find out why they are not behaving as you expect. It allows you to single step through your code, set breakpoints based on complex conditions, modify variables and memory, expand structures and classes, and much more.

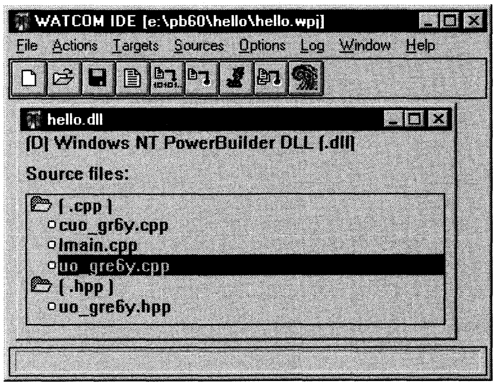
FOR INFO For detailed instructions on using the debugger, see the debugger's online Help. To start the debugger, select Target>Debug from the IDE main menu. When the debugger starts, select Help.

Using the IDE

❖ **To start the IDE from PowerBuilder:**

- Click the IDE button in the User Object PainterBar.
  or
  Select Design>Invoke C++ Editor.
  or
  Right click anywhere in the User Object painter and select Invoke C++ Editor from the popup menu.

  Here is a picture similar to what you should see when the IDE starts:


WATCOM IDE [e:\pb60\hello\hello.wpj]
File Actions Targets Sources Options Log Window Help
hello.dll

[D] Windows NT PowerBuilder DLL [.dll]
Source files:
[.cpp]
cuo_gr6y.cpp
lmain.cpp
uo_gre6y.cpp
[.hpp]
uo_gre6y.hpp

Toolbar buttons

These are the available toolbar buttons:

Button	Name	What you do
	Create project	Set up a new project file with no targets in it. When beginning work on a new PowerBuilder DLL, you can do this step yourself or have PowerBuilder create a new project file for you
	Open project	Opens a previously saved project so you can work on the source code or re-make the target DLL
	Save project	Saves the current project. The source files defined within a project must be saved separately, from the editor

Button	Name	What you do
	Edit	When you open a project, all the source files associated with it display in a listbox. To edit one of them, select one and click this button. You can also double-click on the filename within the target window
	Build the selected source	Compile and link a single source file The result is an object (obj) file being created for that source, and a DLL being produced from that object by the linker
	Build the current target	Make a PowerBuilder DLL If your target has more than one source, you click here to compile and link all source files, and create a DLL from them
	Debugger	Start the Watcom character-mode debugger This is not the PowerBuilder debugger, but a tool for controlling execution of your compiled DLL. To use it, you set breakpoints in your C++ source code and run your PowerBuilder application. When a DLL breakpoint is reached, execution is transferred to the debugger, where you can single-step or loop through the DLL as required
	Build all targets in the current project	Allows you to write source code for more DLLs, compile them all in one step, and link them into their respective executables. This is useful if, for example, several DLLs share the same source code modules or header files
	Exit	Returns you from the IDE to PowerBuilder

Editing your C++ source code

When you enter the IDE from PowerBuilder, you see an open project listing the C++ source files corresponding to your PowerBuilder user objects.

This example
These steps edit the source files for the Hello World example.

❖ **To edit the source files:**

1. Select the C++ skeleton file (uo_gre6Y.cpp) and click the Edit button.
   or
   Double-click the filename.

   The skeleton of your C++ source code displays in the editor window.

2. In the function cf_save_message, move to the space labeled:

   ```
   /*
   * Put your code here
   */
   ```

3. Type:

   stored_message = input_message;

   This transfers the message string into the instance variable you declared earlier, where the function cf_display_message can find it and display it.

4. In the function cf_display_message go to the space labeled:

   ```
   /*
   * Put your code here
   */
   ```

5. Type this as a single line:

   MessageBox (NULL, stored_message, "C++ User Object Message", MB_OK);

   This entry point uses the Windows MessageBox function to display the contents of the instance variable, stored_message, in a modal window.

   The completed uo_gre6Y.cpp source code should look like this (the lines you have added in this tutorial are in *bold*).

   ```
   /* Watcom Interface Generator Version 1.0 */
   /* This file contains code generated by PowerBuilder.
   * Do not modify code delimited by comments of the
   form:
   * // $PB$ -- begin generated code for object <>. Do
   not modify this code
   * // $PB$ -- end generated code for object <>.
   * This file contains the bodies the functions for
   your user object.
   */
   ```

```
#include <pbdll.h>

#include "uo_gre6Y.hpp"

// $PB$ -- begin generated code for object
//<uo_greeting>. Do not modify this code
#if 1
void uo_greeting::cf_display_message( ) {
// $PB$ -- end generated code for object
//<uo_greeting>.
//===================================

   stored_message = input_message;
}
#endif // PowerBuilder code, do not remove

// $PB$ -- begin generated code for object
//<uo_greeting>. Do not modify this code
#if 1
void uo_greeting::cf_save_message( ){
// $PB$ -- end generated code for object
//<uo_greeting>.
//===================================

   MessageBox (NULL, stored_message, "C++ User Object
   Message", MB_OK);
}
#endif // PowerBuilder code, do not remove
```

6 Select File>Save to save your work.

7 Double-click the editor control box.
or
Select File>Exit to quit the Editor.

8 Choose Yes when prompted to save the changes you have just made.

The MessageBox function

The first DLL entry point makes a call to the Windows API MessageBox function. This is a very useful function that places a complete dialog box, with a custom message and limited button response, on the window you choose. Here is the basic syntax of the MessageBox function call:

MessageBox (parent window, message, titlebar text, buttons) where:

- *parent window* In our example, the current window is the parent, so we can set the first parm to NULL. Notice this is entered in uppercase. This is required, because it must match exactly the definition of the word NULL in the file WINDOWS.H. This header file is #INCLUDEd at the top of this source file, and contains definitions of many constants and function prototypes for Windows API calls, including the MessageBox call.
- *message* Stored_message is the name of the string variable that holds the message you want to display.
- *titlebar text* The quoted string *C++ User Object Message* is any string you want to appear in the title bar of the message box.
- *buttons* MB_OK is another constant from WINDOWS.H, and causes a single button labeled OK to appear in the message box, below your message.

Compiling and linking the DLL

In this example, we will use the Make utility to compile and link the Hello World applet.

On your distribution disk is a PowerBuilder application called Hello. It consists of hello.pbl, a PowerBuilder library containing:

- An application object called hello
- w_main, a window with:
    - A button
    - A single line edit control
- A user object called uo_greeting that displays text in a Windows message box

What follows is an abridged version of the exported code for the application, window, and user object.

Application object

Open script

```
on open;
/* Open a main window */
```

```
Open (w_main)
end on
```

w_main

Variable declarations

```
shared variables
     uo_greeting my_greeting
end variables
```

Open script

```
on open;
/* Instantiate the PowerBuilder user object */
/* my_greeting is declared as a SHARED VARIABLE */
   my_greeting = CREATE uo_greeting
/**************************************************/
/* Two DLL functions are called from the CLICKED */
/* event of cb_OK: */
/* cf_save_message fetches the contents of */
/* sle_input on w_main and cf_display_message */
/* displays it in a Windows message box. */
/**************************************************/
end on
```

Close script

```
on close;
/* Destroy the user object instance */
   DESTROY my_greeting
end on
```

Button cb_1

Clicked script

```
on clicked;
     /* Get greeting string from w_main.sle_1; */
     /* call DLL function to store it */
     my_greeting.cf_save_message (
          w_main.sle_input.text)
          /* Call another function in the DLL to */
          /* display the message */
          my_greeting.cf_display_message ( )
     end on
```

uo_greeting

```
     $PBExportHeader$uo_greeting.sru
          global type uo_greeting from cplusplus
     end type
```

DLL name

```
     global type uo_greeting from cplusplus
          string LibraryName=".\hello.dll"
     end type
     global uo_greeting uo_greeting /* instantiate it */
```

Instance variable definition

```
     type variables
          private string stored_message
     end variables
```

Function definitions

```
     forward prototypes
          public subroutine cf_display_message ( )
          public subroutine cf_save_message ( string
     input_message )
     end prototypes
```

DLL Name
Notice the string variable LibraryName declared at the label DLL name above. It is initialized to the value ".\hello.dll"—the name you supplied in the DLL Name dialog box earlier. The characters ".\" indicate that the DLL should be found in the current directory, the one containing the PowerBuilder PBL. If you move the DLL out of this directory, the functions in uo_greeting may not be able to find it.

❖ **To set the value of LibraryName to the directory where the DLL resides:**

1. Export the user object to an sru file.
2. Change the value of LibraryName to the new pathname.
3. Save the sru file and import it using the Library painter.

❖ **To see the complete generated code for each of the PowerBuilder objects described above:**

1. Open the Library painter.
2. Select the object you want to export.
3. Click the Export button on the PainterBar.
4. Use the PowerBuilder File Editor or another text editor such as DOS Edit or Windows Notepad to browse the resulting file.

The application object, window, and user object will be found in files with the extensions sra, srw, and sru respectively.

You are now ready to compile and link the Hello World application.

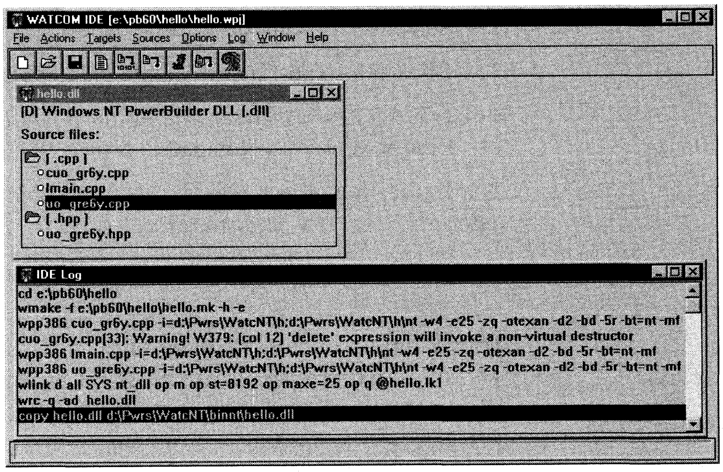
This example uses the Make utility
The steps in this example use the Make utility to compile and link the Hello World applet.

❖ **To use the Make utility to compile and link the application:**

1 From the IDE toolbar, select the Make Current Target button.

The Make utility will use a preset script to control the compiler and linker to produce a Windows DLL from your code.

The IDE opens a log window while the Make utility runs, so you can see the compiler and linker directives that are being followed. When you see the message Execution complete, the Make utility has completed successfully.


WATCOM IDE [e:\pb60\hello\hello.wpj]
File Actions Targets Sources Options Log Window Help
hello.dll
[D] Windows NT PowerBuilder DLL [.dll]
Source files:
[.cpp]
cuo_gr6y.cpp
lmain.cpp
uo_gre6y.cpp
[.hpp]
uo_gre6y.hpp
IDE Log

```
cd e:\pb60\hello
wmake -f e:\pb60\hello\hello.mk -h -e
wpp386 cuo_gr6y.cpp -i=d:\Pwrs\WatcNT\h;d:\Pwrs\WatcNT\h\nt -w4 -e25 -zq -otexan -d2 -bd -5r -bt=nt -mf
cuo_gr6y.cpp(33): Warning! W379: (col 12) 'delete' expression will invoke a non-virtual destructor
wpp386 lmain.cpp -i=d:\Pwrs\WatcNT\h;d:\Pwrs\WatcNT\h\nt -w4 -e25 -zq -otexan -d2 -bd -5r -bt=nt -mf
wpp386 uo_gre6y.cpp -i=d:\Pwrs\WatcNT\h;d:\Pwrs\WatcNT\h\nt -w4 -e25 -zq -otexan -d2 -bd -5r -bt=nt -mf
wlink d all SYS nt_dll op m op st=8192 op maxe=25 op q @hello.lk1
wrc -q -ad  hello.dll
copy hello.dll d:\Pwrs\WatcNT\binnt\hello.dll
```

Many of the compiler and linker directives you see here can be altered from the IDE menu bar. For now, you will get the applet compiled and linked so you can see how it works. Later you will learn about a switch you can set that affects the amount of debugging information available.

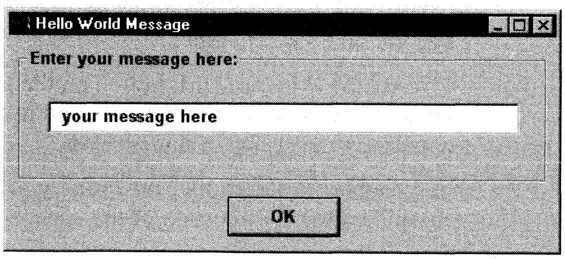
You will now have a file called hello.dll in the directory you specified when you created your C++ user object.

2 Click the Exit button to return to PowerBuilder.

The IDE project does not close.

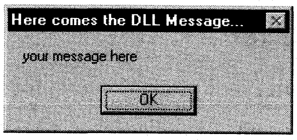
3 Run the PowerBuilder application.

You should see a window with a single line edit and a button, and a prompt for you to enter a short message.



4 When you do, click OK.

The clicked script for the button calls the two functions you have just coded in the DLL—the first to store the message you entered and the second to display it in a message box. Here is what you will see:



your message here

While this a very simple example, it shows you just how easy it is to build a DLL from a PowerBuilder user object.

The Watcom debugger

So far, you have done the following:

1 Created your PowerBuilder user object, including user object functions and variable declarations.

2 Clicked the Watcom DLL button on the User Object PainterBar. This created skeleton C++ source code and invoked the IDE, ready for you to edit your user object source code.

3 In the editor, selected the source file you want to work with and completed the C++ portion of the user object function coding.

4 Used the Make utility to compile and link your user object into a DLL.

When the last step is successful, you're ready to run and debug your application.

Features of the Watcom debugger

What follows is a brief overview of debugger features.

FOR INFO For detailed help using the debugger, select Help from the debugger menu bar.

Execution history

The debugger keeps an execution history as you debug your program. This history is accessible using the Undo menu. The effect of program statements as you single-step through your program are recorded. All interactions that allow you to modify the state of your program (including modifying variable values and changing memory and registers) are also recorded.

You can resume program execution at any previous point in the history. The program history has no size restrictions aside from the amount of memory available to the debugger, so theoretically you could single-step through your entire program and then execute it in reverse.

There are several practical problems that get in the way of this. When you single-step over a call or interrupt instruction, or let the program run normally, the debugger has no way of knowing what kind of operations occurred. No attempt is made to discover and record these operations, but the fact that you stepped over a call is recorded. If you try to resume program execution from a point before a side-effect, the debugger will give you the option to continue or back out of the operation. Use caution if you choose to continue. If an important operation is duplicated, your program could fail. Of course, reversing execution over functions with no side-effects is harmless, and can be a useful debugging technique. If you have accidentally stepped over a call that does have an important side-effect, you can use Replay to restore your program state.

Undo/Redo

Undo and Redo let you browse backward and forward through this execution history. As you use these menu items, all recorded effects are undone or redone, and each of the debugger's windows is updated accordingly.

Unwind/Rewind

Unwind and Rewind move the debugger's state up and down the call stack. Like Undo, all windows are updated as you browse up and down the stack, and you can resume execution from a point up the call stack. A warning is issued if you try resuming from a point up the call stack, since the debugger cannot completely undo the effects of the call. Unwind is particularly useful when your program crashes in a routine that does not contain debugging information.

Caution
If you modify the machine state in any way when you are browsing backward through the execution history, all forward information from that point is discarded. If you have browsed backward over a side-effect, the debugger will give you the option of canceling any such operation.

Replaying the call stack

The debugger also keeps a history of all interactions that affect the execution of your program, such as setting breakpoints and tracing. Replay allows you to restart the application and run it back to a previous point. This is particularly useful when you accidentally trace over a call. This replay information may be saved to a file in order to resume a debugging session at a later date.

You can navigate up and down the program's call stack to see where the currently executing routine was called from. As you do this, all other windows in the debugger update automatically. Local variables in the calling routines will be displayed along with their correct values.

There are special cases where replay will not perform as expected. Since replay is essentially the same as playing your keystrokes and mouse interactions back to the debugger, your program must behave identically on a subsequent run. Any keyboard or mouse interaction that your program expects must be entered the same way. If your program expects an input file, you must run it on the same data set. Your program should not behave randomly or handle asynchronous events. Finally, your program should not be multithreaded. If you have been tracing just one thread, your program will replay correctly, but multiple threads may not be scheduled the same way on a subsequent run.

Setting breakpoints

The debugger allows you to set breakpoints when specific code is executed or data is modified. These breakpoints may be conditional, based on an expression or a countdown. Simple breakpoints are created with a keystroke or single mouse click. More complex breakpoints are entered using a dialog box. The debugger contains small buttons that appear on the left side of some windows. These buttons are shortcuts for the most common operations, such as setting and clearing a breakpoint by clicking the button to the left of a source line.

Debugger menu items

Context-sensitive menus are present in each debugger window. To use them, you select an item from the screen using the right mouse button. A menu containing a list of actions appropriate for that item is displayed. You can use this capability to perform actions such as displaying the value of an expression which you have selected from the source window. Here is a list of some of the most commonly used menu items.

Inspect Inspect displays the selected item. The debugger determines how to best display the item based on its type. If you inspect a variable or an expression, the debugger opens a new window showing its value. If you inspect a function, the debugger positions the source code window at the function definition. If you inspect a hexadecimal address from the assembly window, a window showing memory at that address is opened, and so on. Experimenting with Inspect will help you learn to use the debugger effectively.

Modify Modify lets you change the selected item. You will normally be prompted for a new value. For example, select the name of a variable from any window and choose Modify to change its value.

New New adds another item to a list of items displayed in a window. For example, choosing New in the Break Point window lets you create a new breakpoint.

Delete Delete removes the selected item from the window. For example, you can use Delete to remove a variable from the Watches window.

Source Source displays the source code associated with the selected item. The debugger will reposition the source code window at the appropriate line. Selecting a module name and choosing Source will display the module's source code.

Assembly Assembly positions the assembly code window at the code associated with the selected item.

Functions Functions shows a list of all functions associated with the selected item or window. For example, choose Functions in the source window to see a list of all functions defined in that module.

Watch Watch adds the selected variable or expression to the Watches window. This allows you to watch its value change as the program runs. Note that this is not a watchpoint. Execution will not stop when the variable changes.

Break Break sets a breakpoint based on the selected item. If a variable is selected, the program will stop when the variable is modified. If a function is selected, the program will stop when the function executes.

Globals Globals shows a list of global variables associated with the selected item.Show presents a cascaded menu that lets you show things related to the selected item. For example, you can use Line from the Show menu in the source code window to see the line number of the selected line.

Type Type presents a cascaded menu that allows you to change the display type of the window or selected item.

Debugging

These are the basic steps to follow when debugging a DLL created for PowerBuilder.

❖ **To debug a DLL created for PowerBuilder:**

1 From the IDE project window, select the source file you want to debug.

2 Invoke the debugger.

3 Set any breakpoints or loop specifications you need to control the DLL when this source module is executed.

4 Select Run>Go from the debugger menu bar.

5 Switch to PowerBuilder and run your application.

As soon as a breakpoint is reached within your DLL, the code will be executed under control of the debugger, allowing you to step through your program, trace program execution, and manipulate variables and the program stack.

FOR INFO For detailed instructions on using breakpoints and traces and inspecting and modifying memory with the debugger, see the debugger online Help.

Debugging DLLs

The debugger automatically detects all DLLs that your application references, as they are loaded. If you created your DLL to include debugging information, you can debug it just as if it were part of your application. Even if it does not have debugging information, the debugger will process system information to make the DLL entry point names visible. There are a few limitations:

- You cannot debug your DLL initialization code. This is the first routine that the operating system runs when it loads the DLL. This is not normally a problem, since most DLLs do not do much in the way of initialization.
- PowerBuilder DLLs are always loaded dynamically. Because of this, the debugging information may not be available immediately. Try tracing a few instructions and it will appear.
- If you restart an application, you will lose any breakpoints that you had set in dynamically loaded DLLs. You need to trace back over the call to the DLL entry point and reset these breakpoints.

Let's assume you want to debug your application in order to locate an error in programming. In the previous section, the Hello World program was compiled with default compile and link options. When debugging an application, it is useful to refer to the symbolic names of routines and variables. It is also convenient to debug at the source line level rather than the machine language level. To do this, you must direct both the compiler and linker to include additional debugging information in the object and executable files.

Setting debug switches

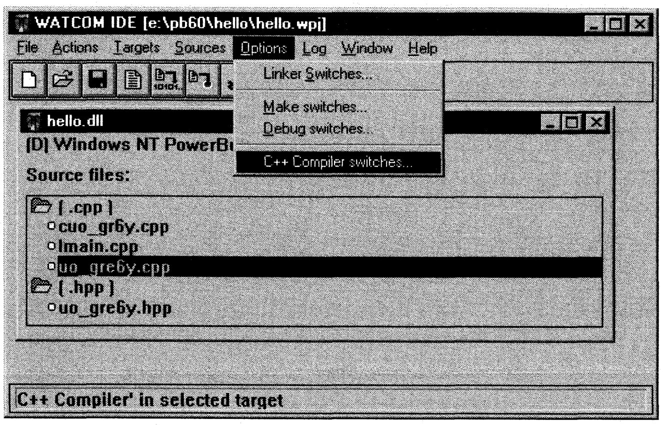
Using the Options menu from the IDE menu bar, you control the amount of debugging information that is built into your executable code.

This example

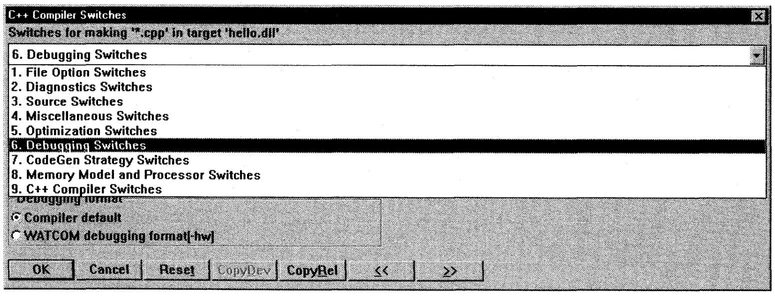
These steps compile and link the Hello World program with full debugging information.

❖ **To compile and link your program with full debugging information:**

1 Select Options>C++ Compiler Switches from the menu bar.



2 Select Compile Switch group 6: Debugging Switches.



3 Set the Debugging Style to Full debugging info [d2].

CHAPTER 4

Sample Program: Forecast

About this chapter

This chapter describes an additional example, called Forecast, which is more complex than the Hello World sample used in Chapter 2, "Creating the C++ Source Code".

Contents

About the Forecast application

Forecast makes use of a hierarchy of C++ classes. The application forecasts the next value in a series, based on the numbers that have come before. The algorithm used can be as simple as assuming the next number will be the same as the last. It may be as complex as calculating the average growth over several previous samples to arrive at a forecast value.

This program is a good example of creating and using C++ classes with PowerBuilder. There are six different forecasting algorithms, each one implemented in its own user object. The same algorithms are also performed by external functions in a DLL.

The application menu and toolbars let the user choose which algorithm to use and whether to perform its calculations using the PowerScript user object functions, the compiled DLL equivalents, or both. For comparison, the two sets of results appear side by side, in two DataWindows. Below each is a benchmark displaying the time taken to compute the series of forecasts. This highlights the performance benefit of using C++ classes over user object functions written in PowerScript.

If you want to experiment with this example, it's best to start PowerBuilder and load the Forecast application. The two libraries dllsampl.pbl and dlltools.pbl contain all the PowerBuilder objects and code you need to run the application. This includes a hierarchy of C++ user objects, each one encapsulating a different forecasting algorithm and statistics about its performance. The file forecast.dll contains the C++ equivalent of these user objects, with a separate class for each forecast type.

You are expected to have a reasonable understanding of PowerBuilder application development, so we are not going to provide a lengthy explanation of these libraries. Armed with your experience plus information in the following topics, you should be able to make significant changes to it.

Forecast application class hierarchy

The basic object class in this program is the *forecast*, and it is coded as a class user object called uobase (user object *base*). It forms the base class in a hierarchy of classes that inherit common properties from it. The user objects in this hierarchy are:

User object name	Inherited from	Description
statistics	—	Provides statistical services to all forecast classes; defined within the definition of uobase
uobase	—	Sets the forecast for the next period to the actual value of the previous period
uoincrement	uobase	Calculates the forecast for the next period by adding a fixed increment to the actual value for the previous period
uogrowth	uobase	Calculates the forecast for the next period by adding a fixed percentage of the previous period's actual value
uoaverage	uobase	Calculates the forecast for the next period by averaging the actual value for a number of previous periods
uovincrement	uoincrement	Calculates the forecast for the next period by adding the average increment over a number of previous periods
uovgrowth	uogrowth	Calculates the forecast for the next period as a factor of the average growth over the previous several periods

These classes are implemented as PowerBuilder user objects, in the tutorial file dlltools.pbl. This library also contains the application's DataWindows, global functions, and a structure to hold statistical information. The application object, along with all windows and menus, are found in dllsampl.pbl in the same directory.

The C++ class equivalent of uobase is cobase (C++ object *base*). Its function declarations are shown below. You can enter these declarations and compile the DLL yourself, or use the compiled version included with this package.

Function name	Return type	Parm name	Parm type
Arithmetic_ mean	double	which_one	integer

Function name	Return type	Parm name	Parm type
Initialize	n/a	given_actual	double
Maximum	double	which_one	integer
Minimum	double	which_one	integer
Next_ Actual	n/a	given_actual	double
Period_ Forecast	double	n/a	n/a
Prediction	double	n/a	n/a
Range	double	which_one	integer
Standard_deviation	double	which_one	integer
Title	string	n/a	n/a
Variance	double	n/a	n/a

Cobase source code

PowerBuilder generates three C++ skeleton files for the class cobase: cobase.cpp, cobase.hpp, and ccobase.cpp. Remember that ccobase.cpp is never to be altered, because it consists of interface functions that sit between PowerBuilder and the C++ code you write in cobase.cpp.

The statistical functions shown at the beginning of this listing are members of the statistics class, defined in file cobase.hpp. This class is not derived from a PowerBuilder user object, but is instead defined within the C++ domain and therefore cannot be accessed from PowerBuilder. One class, zero_based_var, is inherited from statistics. The statistical methods in these classes are used by the forecasting algorithms in cobase and its descendent classes to track their success.

The header files string.h and maths.h are added to include the definitions for string functions such as strcpy, and math functions used in some of the derived algorithm classes. Notice also that you can add code to the cobase constructor function, just as you can to the constructor event of a PowerBuilder user object. This is a handy way to initialize data members, or execute function code that must be performed whenever a new instance of the class is created.

The last three functions in this file, initialize_statistics, update_statistics, and compute_forecast, are examples of internal functions. They are not prototyped in the PowerBuilder user object cobase and cannot be called from the Forecast application. They are called from within this DLL only.

cobase.cpp

```
/* Watcom Interface Generator Version 1.0 */
/* This file contains code generated by PowerBuilder.
* Do not modify code delimited by comments of the
form:
* // $PB$ -- begin generated code for object <>.
  // Do not modify this code
* // $PB$ -- end generated code for object <>.
* This file contains the bodies of the functions for
your user object.
*/
#include <string.h>
#include <math.h>
#include <pbdll.h>
#include "cobase.hpp"
```

```
// - - - - - - - - - - - - - - - - - - - -
// statistics class
// - - - - - - - - - - - - - - - - - - - -

void statistics::next_value(double given_value)
{
   this->value_count ++;
   this->sum_of_values += given_value;
   this->sum_of_squares += pow (given_value, 2);
   if (given_value < this->minimum_value)
      this->minimum_value = given_value;
   if (given_value > this->maximum_value)
      this->maximum_value = given_value;
}

void statistics::initialize()
{
   this->value_count = 0;
   this->sum_of_values = 0;
   this->sum_of_squares = 0;
   this->minimum_value = 100000000;
   this->maximum_value = -100000000;
}

double statistics::range()
{
   return (this->maximum_value - this->minimum_value);
}

double statistics::arithmetic_mean()
{
   return (this->sum_of_values / this->value_count);
}
double statistics::standard_deviation()
{
   return (sqrt (this->standard_variance ()));
}

double statistics::standard_variance()
{
   double StdVar;
   if (this->value_count == 1)
      StdVar = 0;
```

```
    else
       StdVar = ((this->sum_of_squares * \
       this->value_count) - \
       pow (this->sum_of_values, 2))/ \
       (this->value_count * (this->value_count - 1));
    return (StdVar);
}

double statistics::maximum()
{
    return (this->maximum_value);
}

double statistics::minimum()
{
    return (this->minimum_value);
}

// - - - - - - - - - - - - - - - - - - - -
// zero_based_var class
// - - - - - - - - - - - - - - - - - - - -

double zero_based_var::standard_variance()
{
    double StdVar;
    if (this->value_count == 1)
       StdVar = this->sum_of_squares;
    else
       StdVar = this->sum_of_squares / (this- \
       value_count - 1);
    return (StdVar);
}

// - - - - - - - - - - - - - - - - - - - -
// cobase class
// - - - - - - - - - - - - - - - - - - - -
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
void cobase::next_actual(double given_actual) {
// $PB$ -- end generated code for object <cobase>.
//=================================
   this->last_actual = given_actual;
   this->last_forecast = this->next_forecast;
   this->last_variance = this->next_forecast - \
      given_actual;
   this->update_statistics ();
   this->compute_forecast ();
}
#endif // PowerBuilder code, do not remove

// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
void cobase::initialize(double given_actual) {
// $PB$ -- end generated code for object <cobase>.
//===================================
   this->last_actual = given_actual;
   this->last_forecast = given_actual;
   this->last_variance = 0;
   this->initialize_statistics ();
   this->update_statistics ();
   this->compute_forecast ();
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::period_forecast() {
// $PB$ -- end generated code for object <cobase>.
//===================================
   return(this->last_forecast);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::variance() {
// $PB$ -- end generated code for object <cobase>.
//===================================
   return(this->last_variance);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
```

```
#if 1
double cobase::prediction() {
// $PB$ -- end generated code for object <cobase>.
//==================================
   return((this->next_forecast);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::arithmetic_mean(int which_one) {
// $PB$ -- end generated code for object <cobase>.
//==================================
double Mean;
switch (which_one)
{
   case 1:
      Mean = this->actual_stats.arithmetic_mean ();
      break;
   case 2:
      Mean = this->forecast_stats.arithmetic_mean ();
      break;
   case 3:
      Mean = this->variance_stats.arithmetic_mean ();
      break;
}
return(Mean);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::standard_deviation(int which_one) {
// $PB$ -- end generated code for object <cobase>.
//==================================
double StdDev;
switch (which_one)
{
   case 1:
      StdDev=this->actual_stats.standard_deviation\();
      break;
```

```
    case 2:
     StdDev=this->forecast_stats.standard_deviation\();
        break;
    case 3:
       StdDev=this>variance_stats.standard_deviation();
       break;
}
return(StdDev);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::range(int which_one) {
// $PB$ -- end generated code for object <cobase>.
//==================================
double Range;
switch (which_one)
{
    case 1:
       Range = this->actual_stats.range();
       break;
    case 2:
       Range = this->forecast_stats.range();
       break;
    case 3:
       Range = this->variance_stats.range();
       break;
}
return(Range);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
char * cobase::title() {
// $PB$ -- end generated code for object <cobase>.
//==================================
return(this->title_tag);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
```

```
#if 1
double cobase::maximum(int which_one) {
// $PB$ -- end generated code for object <cobase>.
//===================================
double Maximum;
switch (which_one)
{
   case 1:
      Maximum = this->actual_stats.maximum();
      break;
   case 2:
      Maximum = this->forecast_stats.maximum();
      break;
   case 3:
      Maximum = this->variance_stats.maximum();
      break;
}
return(Maximum);
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
#if 1
double cobase::minimum(int which_one) {
// $PB$ -- end generated code for object <cobase>.
//===================================
double Minimum;
switch (which_one)
{
   case 1:
      Minimum = this->actual_stats.minimum();
      break;
   case 2:
      Minimum = this->forecast_stats.minimum();
      break;
   case 3:
      Minimum = this->variance_stats.minimum();
      break;
}
return(Minimum);
}
#endif // PowerBuilder code, do not remove
```

```
cobase::cobase ()
   : actual_stats(), forecast_stats(),
variance_stats()
{
strcpy (this->title_tag, "Constant Performance");
}

cobase::cobase (char const * title)
   : actual_stats(), forecast_stats(),
variance_stats()
{
strcpy (this->title_tag, title);
}

void cobase::initialize_statistics ()
{
   this->actual_stats.initialize ();
   this->forecast_stats.initialize ();
   this->variance_stats.initialize ();
}

void cobase::update_statistics ()
{
   this->actual_stats.next_value (this->last_actual);
   this->forecast_stats.next_value (this->
   \last_forecast);
   this->variance_stats.next_value (this->
   \last_variance);
}

void cobase::compute_forecast ()
{
   this->next_forecast = this->last_actual;
}
```

cobase.hpp

This is the header file that defines the classes cobase, statistics, and zero_based_var. Only cobase has a corresponding PowerBuilder user object; statistics and its descendant zero_based_var are not known to PowerBuilder and are used only in the C++ domain.

The initialize_statistics, update_statistics, and compute_forecast functions of class cobase have been manually prototyped here. These functions do not exist in the C++ user object cobase, because there is no need to call them from PowerBuilder. They are known as nonexported, or internal functions, and are called from within this DLL only.

```
/* Watcom Interface Generator Version 1.0 */
/* This file contains code generated by PowerBuilder.
* Do not modify code delimited by comments of the
form:
* // $PB$ -- begin generated code for object <>. Do
not modify this code
* // $PB$ -- end generated code for object <>.
* This file contains the the C++ class definition for
your user object.
*/
#include <string.hpp>
#include <windows.h>

#ifndef _COBASE_DEFN_

class statistics {
public:
   virtual void next_value(double given_value);
   virtual void initialize();
   virtual double arithmetic_mean();
   virtual double standard_deviation();
   virtual double range();
   virtual double maximum();
   virtual double minimum();
protected:
   virtual double standard_variance();
protected:
   intvalue_count;
   doublesum_of_values;
   doublesum_of_squares;
   doubleminimum_value;
   doublemaximum_value;
};
```

```
class zero_based_var : public statistics {
protected:
   virtual double standard_variance();
};
// $PB$ -- begin generated code for object <cobase>.
// Do not modify this code
class cobase {
public:
   virtual double minimum(int which_one);
   virtual double maximum(int which_one);
   virtual char * title();
   virtual double range(int which_one);
   virtual double standard_deviation(int which_one);
   virtual double arithmetic_mean(int which_one);
   virtual double prediction();
   virtual double variance();
   virtual double period_forecast();
   virtual void initialize(double given_actual);
   virtual void next_actual(double given_actual);
// $PB$ -- end generated code for object <cobase>.
//===================================
/*
* PUT YOUR DECLARATIONS HERE
*/
protected:
   doublenext_forecast;
   doublelast_variance;
   doublelast_forecast;
   doublelast_actual;
   chartitle_tag[30];
   statisticsactual_stats;
   statisticsforecast_stats;
   zero_based_varvariance_stats;
protected:
   virtual void initialize_statistics ();
   virtual void update_statistics ();
   virtual void compute_forecast ();
public:
   cobase ();
   cobase (char const * title);
};
#define _COBASE_DEFN_
#endif
```

ccobase.cpp

```
/* Watcom Interface Generator Version 1.0 */
/* This file is generated by PowerBuilder.
* Do not modify this file.
* This file contains interface code called by
PowerBuilder.
*/
#include <pbdll.h>
#include "cobase.hpp"
extern "C" {
double PB_EXPORT cobaseminimum(cobase *this_hdl, \
int which_one);
double PB_EXPORT cobasemaximum(cobase *this_hdl, \
int which_one);
char * PB_EXPORT cobasetitle(cobase *this_hdl);
double PB_EXPORT cobaserange(cobase *this_hdl, \
int which_one);
double PB_EXPORT cobasestandard_deviation(cobase \
*this_hdl, int which_one);
double PB_EXPORT cobasearithmetic_mean(cobase \
*this_hdl, int which_one);
double PB_EXPORT cobaseprediction(cobase *this_hdl);
double PB_EXPORT cobasevariance(cobase *this_hdl);
double PB_EXPORT cobaseperiod_forecast(cobase \
*this_hdl);
void PB_EXPORT cobaseinitialize(cobase *this_hdl, \
double given_actual);
void PB_EXPORT cobasenext_actual(cobase *this_hdl, \
double given_actual);
cobase *PB_EXPORT cobase_CPP_CONSTRUCTOR();
void PB_EXPORT cobase_CPP_DESTRUCTOR(cobase \
*this_hdl);
}
double PB_EXPORT cobaseminimum(cobase *this_hdl, int
which_one) {
   return(this_hdl->minimum(which_one));
}
double PB_EXPORT cobasemaximum(cobase *this_hdl, int
which_one) {
   return(this_hdl->maximum(which_one));
}
```

```
char * PB_EXPORT cobasetitle(cobase *this_hdl) {
   return(this_hdl->title());
}

double PB_EXPORT cobaserange(cobase *this_hdl, int
which_one) {
   return(this_hdl->range(which_one));
}

double PB_EXPORT cobasestandard_deviation(cobase
*this_hdl, int which_one) {
   return((this_hdl->standard_deviation \
   which_one));
}

double PB_EXPORT cobasearithmetic_mean(cobase
*this_hdl, int which_one) {
   return(this_hdl->arithmetic_mean(which_one));
}

double PB_EXPORT cobaseprediction(cobase *this_hdl) {
   return(this_hdl->prediction());
}

double PB_EXPORT cobasevariance(cobase *this_hdl) {
   return(this_hdl->variance());
}

double PB_EXPORT cobaseperiod_forecast(cobase
*this_hdl) {
   return(this_hdl->period_forecast());
}

void PB_EXPORT cobaseinitialize(cobase *this_hdl,
double given_actual) {
   this_hdl->initialize(given_actual);
}

void PB_EXPORT cobasenext_actual(cobase *this_hdl,
double given_actual) {
   this_hdl->next_actual(given_actual);
}
```

```
cobase *PB_EXPORT cobase_CPP_CONSTRUCTOR() {
   return(new cobase);
}

void PB_EXPORT cobase_CPP_DESTRUCTOR(cobase
*this_hdl) {
   delete this_hdl;
}
```

How to run the Forecast program

The two application PBLs, dllsampl.pbl and dlltools.pbl, are not compiled. This makes it easier for you to play with the application and extend its function. The DLL dllsaml.dll is compiled and resides in the same directory as the PowerBuilder libraries. To use it, make sure that directory is in your path.

The names of the toolbar button images referenced by dllsampl.pbl are not fully qualified. While this gives you more flexibility in locating the program libraries where you want them, you must make sure the button images are in your current directory or they won't display.

One more thing: you will notice certain menu items and toolbar buttons are inactive. This should give you some clues about where this application could be enhanced and extended. Feel free to implement the disabled functions as you see fit.

❖ **To run the Forecast application:**

1 Start PowerBuilder and the Forecast application.

2 Click the blue P-Series button to create a series of random numbers that will serve as actual values on which to base the forecasts.

   You are prompted for the min and max values of the series, as well as the number of values within the range. This series is used by the PowerBuilder user object function version of the calculations.

3 To open an equivalent window with the same set of numbers as any of the other forecasting algorithms, click the C-Series button.

4 To begin the forecasting operation, click any of the other forecasting algorithms.

   The forecast value for the present period is calculated according to the algorithm selected. For example, if you choose the Constant button, the forecast value for each point is set to the actual value of the previous point. The difference between each point's actual and forecast values appears in the variance column, with negative variances shown in red.

   Notice the timings displayed below the series windows. They show the actual calculation time, exclusive of screen updates, and are a good indicator of the speed advantage of using compiled C++ class objects.

5 Click the Graph button to see the actual and forecast values plotted graphically.

6 Click the Chart button to see a plot of the variance between the two.

7. Click the Statistics button to display a window that shows statistics such as value ranges, average values, and standard deviations.

   The application runs in an MDI frame, with a number of sheets—data series, graph, statistics—open at any time. Clicking the Close button closes whichever sheet is currently active.

8. Click the Exit button to quit the Forecast application.

APPENDIX A

Conversion Error Messages

About this appendix

This appendix lists error messages related to converting from PowerBuilder to C++ source code.

The messages are listed alphabetically.

Error messages

Attempt to delete 'filename' failed The system tried unsuccessfully to delete a file. The file may be in use by Windows or another application, or it may be protected, or read-only.

Attempt to open 'filename' failed The system was unable to open a file for read or write; check that it exists on the current path; if the file is a DLL, it may already be loaded by this or another application.

Attempt to read from a file failed The file may be protected; if the file is on a network device, it may have been busy. This message may also indicate the contents of the C++ skeleton (cpp, hpp) files have been changed between the comments marked \\PB. Save a copy of each.cpp file (except the cover function file whose name begins with C) and delete them from the current directory. Follow the procedure for creating C++ skeletons from the PowerBuilder User Object painter, to rebuild them.

Attempt to rename 'filename1' to 'filename2' failed The filename may be protected or read-only; it may be in use by another application; or the target filename, filename2 may already exist on the current path.

Attempt to write to a file failed The file may be in use by Windows or another application, or it may be protected, or read-only. If the file is a DLL, it may be in use by Windows or another application.

Improper file format The system attempted to read a file and expected it to be in a particular format. It was not. Compare the expected format to that of the actual file. If necessary, the file may have to be rebuilt.

Out of memory The interface converter ran out of memory translating your PowerBuilder user object into cpp and hpp files. If you have suffered a Windows protection fault during the current session, you may have to reboot to reset memory pointers. If this doesn't solve the problem, try to maximize available memory: close any unused windows and shut down all unnecessary applications.

In Windows 3.1 only
In Windows 3.1 only, try to maximize available memory: remove TSRs that are not required; load DOS and other essential programs into high memory; fine-tune your memory management program (emm386 and so on) if any; or reduce the size of your user object.

Parameter 'inparm' in function 'funcname' contains an array element definition In C++, you cannot define an array by specifying the bounds—that is, int months [1 TO 12]. Array indexes in C++ start at 0 and go to the number of elements-1. Convert array definitions to the form int months [12].

Parameter 'inparm' in function 'funcname' is a multidimensional array You can only pass one-dimensional arrays as a parameter to a function.

Parse error The interface generator program did not understand the syntax of your PowerBuilder user object definition. Check your user object definition and try again. If the error persists, call Powersoft Technical Support.

Prototype for 'funcname' has changed The interface generator is trying to update C++ source code for an existing user object and has found the function definition in PowerBuilder no longer matches the prototype in filename.hpp.

Scanner error The interface generator program did not understand the syntax of your PowerBuilder user object definition. Check your user object definition and try again. If the error persists, call Powersoft Technical Support.

Unknown type 'datatype' in function/subroutine 'funcname' Either the return type of a function or the type of a parameter declared in its parameter list is not a valid data type.

Variable 'varname' contains an array element definition In C++, you cannot define an array by specifying the bounds—that is, int months [1 TO 12]. Array indexes in C++ start at 0 and go to the number of elements-1. Convert array definitions to the form int months [12].

Variable 'varname' has unknown type 'datatype' Check the definition for *varname* and make sure its data type is a valid PowerBuilder data type and that it is spelled correctly.

Variable 'varname' is a dynamic array The interface generator cannot create a C++ definition for a variable-length array. Make sure that all array variables to be converted to C++ code refer to fixed-length arrays only.

Variable 'varname' is a multidimensional array Arrays may be one-dimensional only.

APPENDIX B The C++ Source Code

About this appendix

This appendix presents the C++ source code created by the example in Chapter 2, "Creating the C++ Source Code".

Contents

uo_gre6Y.cpp

This file is generated by PowerBuilder 6.0 when you save the C++ user object. It is a skeleton C++ program, with each of your user object functions "stubbed out." You will complete the program by filling in the code for these functions.

Caution
The C++ class definition (uo_gre6Y.hpp in our example) and skeleton file (uo_gre6Y.cpp) must be modified carefully. Any changes made on or between the two lines starting with //PB could corrupt these files and cause the compiler to fail. If you return to PowerBuilder to make changes to your user object and enter the IDE again, all code enclosed within the //PB comments gets regenerated, destroying any changes you have made.

Any function code added to the C++ skeleton outside the //PB comments will be maintained, even if the user object is changed in PowerBuilder and resaved. This allows changes to your user object without losing the C++ code you've already written.

The C++ cover function file, cuo_gr6Y.cpp, should not be changed. It contains interface code only, and no user-serviceable parts. It is regenerated every time you save the PowerBuilder user object, so any changes you make here will be destroyed. Changes made here can cause unpredictable errors when you attempt to call the user object DLL from PowerBuilder.

Although more #INCLUDE lines can be added, you should not remove any that PowerBuilder provides for you:

- The first of these brings in a header file, pbdll.h, that consists of a single line definition for the macro PB_EXPORT. This is described below.
- The second, uo_gre6Y.hpp, contains function prototypes and data member definitions for your C++ user object that may be used by other code modules.

In the code examples that follow, the '\' character at the end of a line indicates the line of C++ code continues on the next line. You may enter this code as a single line when working in the editor.

```
#include <pbdll.h>
#include "uo_gre6Y.hpp"
/* WATCOM Interface Generator Version 1.0 */
/* This file contains code generated by PowerBuilder.
* Do not modify code delimited by comments of the
```

```
  form:
* // $PB$ -- begin generated code for object <>. Do
  not modify this code.
* // $PB$ -- end generated code for object <>.
* This file contains the bodies the functions for
* your user object.
*/

#include <pbdll.h>
#include <windows.h>
#include "uo_gre6Y.hpp"
// $PB$ -- begin generated code for object
// <uo_greeting>. Do not modify this code
// #if 1
void uo_greeting::cf_display_message() {
// $PB$ -- end generated code for object
// <uo_greeting>.
//===================================
   /*
    * PUT YOUR CODE HERE
    */
}
#endif // PowerBuilder code, do not remove
// $PB$ -- begin generated code for object
// <uo_greeting>. Do not modify this code
#if 1
void uo_greeting::cf_save_message(char * \
input_message) {
// $PB$ -- end generated code for object //
<uo_greeting>.
//===================================
   /*
   * PUT YOUR CODE HERE
   */
}
#endif // PowerBuilder code, do not remove
```

cuo_gr6y.cpp

This file is also created when you save a C++ user object and contains DLL cover function definitions created from your user object. Sometimes called wrapper functions, these routines stand between the calling function in a PowerBuilder user object and the compiled DLL function that does the work. In PowerBuilder's case, this is necessary because of a phenomenon known as name mangling.

The cover function file, cuo_gr6y.cpp, is listed below.

Do not change this code
None of this code is user-serviceable. It is intended strictly as an interface between PowerBuilder and the DLL you create, and you should not change it.

```
/* WATCOM Interface Generator Version 1.0 */
/* This file is generated by PowerBuilder.
 * Do not modify this file.
 * This file contains interface code called by
 * PowerBuilder.
 */
#include <pbdll.h>
#include "uo_gre6Y.hpp"
extern "C" {
void PB_EXPORT uo_greetingcf_save_message(\
   uo_greeting *this_hdl, char * input_message);
void PB_EXPORT uo_greetingcf_display_message(\
   uo_greeting *this_hdl);
uo_greeting *PB_EXPORT \
   uo_greeting_CPP_CONSTRUCTOR();
void PB_EXPORT uo_greeting_CPP_DESTRUCTOR(\
   uo_greeting *this_hdl) ;
}

void PB_EXPORT uo_greetingcf_save_message(\
uo_greeting *this_hdl, char * input_message){
   this_hdl->cf_save_message(input_message) ;
}
```

```
void PB_EXPORT uo_greetingcf_display_message(\
uo_greeting *this_hdl){
   this_hdl->cf_display_message();
}

uo_greeting *PB_EXPORT uo_greeting_CPP_CONSTRUCTOR()
{
   return(new uo_greeting) ;
}

void PB_EXPORT uo_greeting_CPP_DESTRUCTOR(\
uo_greeting *this_hdl) {
   delete this_hdl;
}
```

uo_gre6Y.hpp

```
/* WATCOM Interface Generator Version 1.0 */
/* This file contains code generated by PowerBuilder.
 * Do not modify code delimited by comments of the
 * form:
 * // $PB$ -- begin generated code for object <>.
 * Do not modify this code.
 * // $PB$ -- end generated code for object <>.
 * This file contains the the C++ class definition
 * for your user object.
 */

#include <string.hpp>
#include <windows.h>

// $PB$ -- begin generated code for object
//<uo_greeting>. Do not modify this code.
class uo_greeting {

public:
   virtual void cf_save_message(char * \
    input_message ;
   virtual void cf_display_message();

private:
   String stored_message;
// $PB$ -- end generated code for object
// <uo_greeting>.

   /*
    * PUT YOUR DECLARATIONS HERE
    */
};
```

pbdll.h

This header file consists of a single line definition for the macro PB_EXPORT:

```
#define PB_EXPORT       __pascal        __export
```

This macro definition is brought into the compiled skeleton file by the line:

```
#INCLUDE <pbdll.h>
```

Do not remove this line. In this line, **__pascal** tells the compiler to use the pascal calling convention, and the **__export** keyword in a function declaration means that it is to be an exported DLL function. It is vital that both of these keywords appear in the declaration of each exported function in your code. Otherwise, these DLL functions will not be available from within your PowerBuilder application.

lmain.cpp

LibMain and WEP are two required functions included in every Windows 3.1 and Windows 95 DLL. Windows NT requires only DLLMain. LibMain is called to perform initialization when the DLL is loaded, and WEP does cleanup just before it unloads.

If you are an experienced Windows DLL programmer (in either C or C++), you probably have your own favorite Main and WEP routines. You will no doubt have noticed PowerBuilder supplies these for you, called LibMain and WEP, in the source file lmain.cpp. This file is compiled and linked with your other source files to produce the DLL.

LibMain

```
/* This file is generated by PowerBuilder.
 * You may modify it in any way you wish but do not
 * remove Libmain and WEP. Without them you will be
 * unable to link yur DLL.
 */

#include <windows.h>
#include "pbdll.h"

#ifdef _NT_

extern "C" {

int _stdcall DLLMain(DWORD, DWORD reason, DWORD)
{
   if(reason == DLL_PROCESS_ATTACH) {
      extern char _WD_Present;
      if(_WD_Present  { // This is a hook for the
            Watcom debugger.
         extern void Int3WithSignature(char _far *);
         #pragma aux Int3WithSignature parm
               caller[] = \
            "int 3"\
            "jmp short L1" \
            'W' 'V' 'I' 'D' 'E' 'O' \
            "L1:"
         Int3WithSignature("DLL Loaded");
      }
   }
   return(1);
}
};

#else

int PB_EXPORT LibMain(HANDLE, WORD, WORD, LPSTR)
{
   return(1);
}
```

```
int PB_EXPORT WEP(int)
{
return(1);
}

#endif // PowerBuilder code, do not remove
```

LibMain
These assignments generate no code, but they prevent compiler warnings about unreferenced variables.

WEP

```
int PB_EXPORT WEP(int res) {
res = res
return(1);
}
```