Introduction to Building

This chapter contains recipes for transforming C++ source code into executable programs and libraries. By working through these recipes, you'll learn about the basic tools used to build C++ applications, the various types of binary files involved in the build process, and the systems that have been developed to make building C++ applications manageable.

If you look at the titles of the recipes in this chapter, you might get the impression that I solve the same problems over and over again. You'd be right. That's because there are many ways to build C++ applications, and while I can't cover them all, I try to cover some of the most important methods. In the first dozen or so recipes, I show how to accomplish three fundamental tasks—building static libraries, building dynamic libraries, and building executables—using a variety of methods. The recipes are grouped by method: first, I look at building from the command line, then with the Boost build system (Boost.Build), and then with an Integrated Development Environment (IDE), and finally with GNU *make*.

Before you start reading recipes, be sure to read the following introductory sections. I'll explain some basic terminology, provide an overview of the command-line tools, build systems and IDEs covered in the chapter, and introduce the source code examples.

line: these recipes introduce some essential concepts that you'll need to understand later in this chapter.

Basic Terminology

The three basic tools used to build C++ applications are the *compiler*, the *linker*, and the *archiver* (or *librarian*). A collection of these programs and possibly other tools is called a *toolset*.

The compiler takes C++ source files as input and produces *object files* , which contain a mixture of machine-executable code and symbolic references to functions and data. The archiver takes a collection of object files as input and produces a *static library*, or *archive*, which is simply a collection of object files grouped for convenient use. The linker takes a collection of object files and libraries and *resolves* their symbolic references to produce either an *executable* or *dynamic library* . Roughly speaking, the linker operates by matching each use of a symbol to its definition. When an executable or dynamic library is created, it is said to be *linked*; the libraries used to build the executable or dynamic library are said to be *linked against*.

An executable, or *application*, is simply any program that can be executed by the operating system. A dynamic library, also called a *shared library*, is like an executable except that it can't be run on its own; it consists of a body of machine-executable code that is loaded into memory after an application is started and can be shared by one or more applications. On Windows, dynamic libraries are also called *dynamic link libraries* (DLLs).

The object files and static libraries on which an executable depends are needed only when the executable is built. The dynamic libraries on which an executable depends, however, must be present on a user's system when the executable is run.

Table 1-1 shows the file extensions typically associated with these four basic types of files on Microsoft Windows and Unix. When I mention a file that has a different extension on Windows and Unix, I'll sometimes omit the extension if it's clear from the context.

Object files	*.obj*	*.o*	*.o*
Static libraries	*.lib*	*.a*	*.a*
Dynamic libraries	*.dll*	*.dylib*	*.so*
Executables	*.exe*	No extension	No extension

TIP

In this chapter, whenever I say Unix, I mean Linux, too.

TIP

When you build the examples in this chapter, your tools will generate a number of auxiliary files with extensions that don't appear in Table 1-1. Unless I mention otherwise, you can safely ignore these files. If you really want to know what they do, consult your toolset's documentation.

IDEs and Build Systems

The compiler, linker, and archiver are *command-line tools* , which means they are designed to be run from a shell, such as *bash* on Unix or *cmd.exe* on Microsoft Windows. The names of the input files and output files, together with any other necessary configuration information, are passed to the compiler, linker, and archiver as text on the command line. Invoking these tools by hand is tedious, however. Even for small projects, it can be hard to remember the command-line options for each tool and the order in which the project's source and binary files must be compiled and linked. When a source file is modified, you must determine which object files need to be recompiled, which static libraries need to be updated, and

of separate files, including source files, object files, libraries, and executables—building from the command line is simply impossible.

There are two basic approaches to building large C++ applications:

- An IDE provides a graphical interface for organizing a collection of source files and describing the binary files that should be generated from them. Once you specify this information, you can generate the binary files simply by selecting an appropriate command from a menu or toolbar. The IDE is responsible for determining the order in which the binary files should be generated, the tools needed to generate them, and the command-line options that must be passed to the tools. Whenever you modify one or more of your source files, you can instruct the IDE to regenerate only those binary files that are out of date.

  IDEs organize source files into collections called *projects*. An IDE project is usually associated with a single binary file, or with several *variants* of a binary file, such as the debug and release builds of an application. Most IDEs allow users to organize projects into groups called *project groups* , or *solutions*, and to specify the dependencies between projects in a group.

- A *build system* provides a text file format for describing a collection of source files and the binary files that should be generated from them, together with a *build tool* that reads these text files and generates the binary files by invoking the appropriate command-line tools. Typically, these text files are created and edited using a text editor, and the build tool is invoked from the command line. Some build systems, however, provide a graphical interface for editing the text files and invoking the build tool.

  While IDEs organize files into *projects*, build systems organize files into *targets*. Most targets correspond to binary files that must be generated; other targets correspond to actions the build tool must perform, such as installing an application.

used for much more than building C++ applications. It also has the advantage of being widely used and well-understood by developers. Unfortunately, getting GNU *make* to do exactly what you want it to do can be a challenge, especially with complex projects involving multiple toolsets. For that reason, I will also discuss *Boost.Build*, a powerful and extensible build system designed from the ground up for building C++ applications.

> **TIP**
>
> For a thorough treatment of GNU *make*, see *Managing Projects with GNU make*, Third Edition, by Robert Mecklenburg (O'Reilly).

Boost.Build was developed by members of the Boost C++ Libraries project. It has been used by a large community of developers for several years, and is currently under active development. Boost.Build uses a build tool called *bjam* and text files called *Jamfiles*. Its greatest strength is the ease with which it allows you to manage complex projects involving multiple platforms and build configurations. Although Boost.Build started out as an extension of Perforce's *Jam* build system, it has since undergone extensive redesign. As this book goes to press, the Boost.Build developers are preparing for the official release of the second major version of the build system, which is the version described in this chapter.

Toolset Overview

In this chapter I'll discuss seven collections of command-line tools: GCC, Visual C++, Intel, Metrowerks, Borland, Comeau, and Digital Mars. Table 1-2 shows the names of the command-line tools from the various toolsets; Table 1-3 shows where they are located on your system, if you have them installed. Tool names for Windows use the *.exe* suffix required for Windows executables; for toolsets that are available for both Windows and Unix, I've put this suffix in brackets.

GCC	*g++[.exe]*	*g++*	*ar[.exe]ranlib[.exe]*
Visual C++	*cl.exe*	*link.exe*	*lib.exe*
Intel (Windows)	*icl.exe*	*xilink.exe*	*xilib.exe*
Intel (Linux)	*Icpc*	*icpc*	*arranlib*
Metrowerks	*mwcc[.exe]*	*mwld[.exe]*	*mwld[.exe]*
Comeau	*como[.exe]*	*como[.exe]*	Toolset-dependent
Borland	*bcc32.exe*	*bcc32.exeilink32.exe*	*tlib.exe*
Digital Mars	*dmc.exe*	*link.exe*	*lib.exe*

GCC (Unix)	Typically */usr/bin or /usr/local/bin*
GCC (Cygwin)	The *bin* subdirectory of your Cygwin installation
GCC (MinGW)	The *bin* subdirectory of your MinGW installation
Visual C++	The *VC/bin* subdirectory of your Visual Studio installation[1]
Intel (Windows)	The *Bin* subdirectory of your Intel compiler installation
Intel (Linux)	The *bin* subdirectory of your Intel compiler installation
Metrowerks	The *Other Metrowerks Tools/Command Line Tools* subdirectory of your CodeWarrior installation
Comeau	The *bin* subdirectory of your Comeau installation
Borland	The *Bin* subdirectory of your C++Builder, C++BuilderX or Borland command-line tools installation

[1] In previous versions of Visual Studio, the *VC* directory was called *VC98* or *Vc7*.

Don't let the number of toolsets scare you—you don't need to learn them all. In most cases you can simply skip the material that doesn't relate to your toolset. If you want to learn a little about other toolsets, however, be sure to read the sections on Visual C++ and GCC, since these are the dominant toolsets on Windows and Unix.

The GNU Compiler Collection (GCC)

GCC is a collection of compilers for a wide assortment of languages, including C and C++. It's remarkable for being open source, available on almost every imaginable platform, and highly conformant to the C++ language standard. It's the dominant compiler on many Unix platforms, and is also widely used on Microsoft Windows. Even if GCC is not your primary toolset, you can learn a lot by compiling your code with GCC. Also, if you think you know a way to improve the C++ language, you can test your idea with the GCC code base.

GCC comes with libstdc++, a good open source implementation of the C++ standard library. It can also be used with the open source STLPort C++ standard library and with Dinkumware's standard library.

TIP

To obtain GCC, see Recipe 1.1.

TIP

The GCC examples in this chapter were tested with GCC 3.4.3 and GCC 4.0.0 on GNU/Linux (Fedora Core 3), with GCC 4.0.0 on Mac OS X (Darwin 8.2.0), and with GCC 3.4.2 (MinGW) and 3.4.4 (Cygwin) on Windows 2000 Professional.

Visual C++

Microsoft's toolset is the dominant toolset on the Windows platform. While several old versions are still in wide use, the most recent version is highly standards conforming. It is also capable of producing highly optimized code. Microsoft's tools are distributed with the Visual C++ and Visual Studio development environments, discussed in the next section. As of this writing, they are also available as part of the Visual C++ Toolkit 2003, which can be downloaded for free from www.microsoft.com.

Visual C++ comes with a customized version of the Dinkumware C++ standard library implementation. Dinkumware's C++ standard library is among the most efficient and stan-

TIP

The Visual C++ examples in this chapter were tested with Microsoft Visual Studio .NET 2003 and Microsoft Visual Studio 2005 (Beta 2). See Table 1-4.

Table 1-4. Versions of Microsoft Visual Studio

Product name	IDE version	Compiler version
Microsoft Visual Studio	6.0	1200
Microsoft Visual Studio .NET	7.0	1300
Microsoft Visual Studio .NET 2003	7.1	1310
Microsoft Visual Studio 2005 (Beta 2)	8.0	1400

Intel

Intel produces several C++ compilers for use with Intel processors. They are notable for generating extremely fast code—perhaps the fastest available for the Intel architecture. Based on the C++ frontend from the Edison Design Group (EDG), they are also highly standards conforming.

The Intel C++ Compiler for Windows makes use of Microsoft's Visual C++ or Visual Studio development environments, which must be installed for the Intel compiler to function properly. The compiler is designed for compatibility with Visual C++: it can be used as a plug-in to the Visual C++ development environment, it can generate code that is binary-compatible with code generated by the Visual C++ compiler, it offers many of the same command-line options as the Visual C++ compiler, and—unless you tell it not to—it even emulates some

Whereas Intel's compiler for Windows is designed to be compatible with the Visual C++ compiler, Intel's compiler for Linux is designed to be compatible with GCC. It requires GCC to operate, supports a number of GCC options, and by default implements some GCC language extensions. The commercial version of the Intel C++ Compiler for Linux is available for purchase at www.intel.com. A noncommercial version is available as a free download.

On Windows, the Intel compiler uses the Dinkumware standard library that ships with Visual C++. On Linux, it uses libstdc++.

TIP

The Intel examples in this chapter were tested with the Intel C++ Compiler 9.0 for Linux on GNU/Linux (Fedora Core 3) and with the Intel C++ Compiler 9.0 for Windows on Windows 2000 Professional.

Metrowerks

Metrowerks's command-line tools, distributed with its CodeWarrior development environment, are among the best available, both in terms of standards conformance and the efficiency of the code they generate. They also come with MSL, Metrowerks's first-rate implementation of the C++ standard library. Until recently, Metrowerks produced tools for Windows, Mac OS, and a variety of embedded platforms. In 2004, however, Metrowerks sold its Intel x86 compiler and debugger technology to Nokia and discontinued its CodeWarrior product line for Windows. In 2005, after Apple Computer announced plans to switch to chips made by Intel, Metrowerks disclosed that the forthcoming CodeWarrior 10 for Mac OS will likely be the final release for that platform. In the future, Metrowerks's focus will be on embedded development targeted at chips made by Freescale Semiconductor.

TIP

By the time you read this, Metrowerks will be a part of Freescale Semiconductor, and the name Metrowerks may no longer be associated with the CodeWarrior product line. I'll still use the name Metrowerks, however, because it's not yet clear what the future names will be.

8.2.0) and with CodeWarrior 9.4 on Windows 2000 Professional.

Borland

Borland's command-line tools were once considered pretty good. As of September 2005, however, the last major update is over three years old and represents only an incremental improvement of over the previous version, which was released in 2000. As a result, Borland's tools are now quite out-of-date. In 2003 Borland announced plans for an ambitious redesign of its C++ compiler, using the EGD frontend; unfortunately, Borland has made no new announcements about this plan for quite some time. Borland's command-line tools remain important, however, because they are still in wide use.

Currently, the most recent versions of Borland's command-line tools can be obtained by purchasing the C++Builder or C++BuilderX development environments, described in the next section, or by downloading the free personal edition of C++BuilderX.

The Borland toolset comes with two C++ standard libraries: STLPort and an outdated version of Rogue Wave's standard library. Borland is also working on producing a version of its tools that will be distributed with the Dinkumware standard library.

TIP

The Borland examples in this chapter were tested with Borland C++ Builder 6.0 (compiler version 5.6.4) on Windows 2000 Professional.

Comeau

The Comeau C++ compiler is widely regarded as the most standards-conforming C++ compiler. In addition to implementing the most recent version of the C++ language, it supports several versions of C and a number of early dialects of C++. It's also among the least expensive, currently priced at $50.

Comeau is available for Microsoft Windows and for many Unix platforms. If Comeau is not available on your platform, you can pay Comeau Computing to produce a custom port, but this is substantially more expensive. You can order the Comeau compiler at www.comeaucomputing.com.

> **TIP**
>
> When I discuss Comeau on Unix, I'll assume the backend compiler is GCC. When I discuss Comeau on Windows, I'll try to indicate how the command-line options depend on the backend compiler. Since Comeau can be used with so many backends, however, it's not always possible to be exhaustive.

Comeau comes with libcomo, an implementation of the C++ standard library based on Silicon Graphics's standard library. It can also be used with Dinkumware's standard library.

> **TIP**
>
> The Comeau examples in this chapter assume that you're using libcomo and that you've configured the compiler to find libcomo automatically. The examples have been tested with Comeau 4.3.3 and libcomo 31 using GCC 3.4.3 as backend on GNU/Linux (Fedora Core 3) and using Visual C++ .NET 2003 as backend on Windows 2000 Professional. (See Table 1-4.)

Digital Mars

Digital Mars is a C++ compiler written by Walter Bright. You can download it for free from www.digitalmars.com; for a modest fee you can order a CD containing the Digital Mars compiler, an IDE, and some other useful tools. The free version of the compiler can be used to compile all the Digital Mars examples in this chapter except for the ones that require a dynamic version of the runtime library, which is only available on the CD.

Digital Mars is a very fast compiler and produces highly optimized code. Unfortunately, it currently has some problems compiling code that uses advanced template idioms. Fortunately, Walter Bright is very responsive to bug reports and is committed to making Digital Mars standards-conforming.

chapter use the STLPort standard library.

> **TIP**
>
> The Digital Mars examples in this chapter have been tested using Digital Mars 8.45 on Windows 2000 Professional.

IDE Overview

In this chapter I'll cover four IDEs: Microsoft Visual C++, Metrowerks CodeWarrior, Borland C++Builder, and Bloodshed Software's Dev-C++. There are a number of important IDEs I won't discuss—Apple's Xcode and the Eclipse Project are prominent examples—but the treatment of the four IDEs I do discuss should give you a good start on learning to use other IDEs.

> **TIP**
>
> As with the command-line tools, feel free to skip material that doesn't relate to your IDE.

Visual C++

Microsoft Visual C++ is the dominant C++ development environment for Microsoft Windows. It's available as a standalone application or as part of the Visual Studio suite, and it comes with a wide assortment of tools for Windows development. For portable C++ development, its most notable features are the following:

- A highly conformant C++ compiler
- The Dinkumware C++ standard library
- A good visual debugger
- A project manager that keeps track of dependencies between projects

The first version of Visual C++ to include a first-class C++ compiler and standard library appears in the third row of Table 1-4. All previous versions had serious standards-conformance problems.

CodeWarrior

CodeWarrior is Metrowerks's cross platform development environment. It has many of the same features as Visual C++, including:

- A highly conformant C++ compiler
- An excellent C++ standard library
- A good visual debugger
- A project manager that keeps track of dependencies between projects

One of CodeWarrior's strengths has traditionally been the large number of platform for which it was available; as explained in the last section, however, its Windows product line has been discontinued and its Macintosh product line will likely be discontinued soon. However, it should remain an important platform for embedded development.

TIP

When I discuss the CodeWarrior IDE, I'll assume you're using CodeWarrior 10 for Mac OS X. The CodeWarrior IDE on other platforms is very similar.

C++Builder

C++Builder is Borland's development environment for Microsoft Windows applications. One of its main attractions is its support for Borland's Visual Component Library. For portable C++ development, however, its most notable features are

- An aging C++ compiler

- A project manager with limited ability to handle dependencies between projects

I cover C++Builder because it is widely used and has a dedicated community of users.

C++Builder should not be confused with C++BuilderX, a cross-platform development environment released by Borland in 2003. Although C++BuilderX is a useful development tool, it has not been a commercial success and it's uncertain whether Borland will release an updated version.

Dev-C++

Bloodshed Software's Dev-C++ is a free C++ development environment for Windows that uses the MinGW port of GCC, described in Recipe 1.1. It features a pretty decent text editor and a visual interface to the GNU debugger.

Dev-C++ offers an incomplete graphical interface to GCC's numerous command-line options: in many cases users must configure their projects by entering command-line options in text boxes. In addition, its project manager can only handle one project at a time and its visual debugger is unreliable. Despite these limitations, Dev-C++ has an active community of users, including many university students. It is a good environment for someone who wants to learn C++ and doesn't own any C++ development tools.

John, Paul, George, and Ringo

Ever since Brian Kernighan and Dennis Ritchie published *The C Programming Language* in 1978, it's been traditional to begin learning a new programming language by writing, compiling and running a toy program that prints "Hello, World!" to the console. Since this chapter covers static and dynamic libraries as well as executables, I'll need a slightly more complex example.

Example 1-1, Example 1-2, and Example 1-3 present the source code for the application *hellobeatles*, which prints:

to the console. This application could have been written as a single source file, but I've split it into three modules: a static library *libjohnpaul*, a dynamic library *libgeorgeringo*, and an executable *hellobeatles*. Furthermore, while each of the libraries could easily have been implemented as a single header file and a single *.cpp* file, I've split the implementation between several source files to illustrate how to compile and link projects containing more than one source file.

> **TIP**
>
> Before you start working through the recipes in this chapter, create four sibling directories *johnpaul*, *geogreringo*, *hellobeatles*, and *binaries*. In the first three directories, place the source files from Example 1-1, Example 1-2, and Example 1-3. The fourth directory will be used for binary files generated by IDEs.

The source code for *libjohnpaul* is presented in Example 1-1. The public interface of *libjohnpaul* consists of a single function, `johnpaul()`, declared in the header *johnpaul.hpp*. The function `johnpaul() is` responsible for printing:

```
John, Paul,
```

to the console. The implementation of `johnpaul()` is split between two source files, *john.cpp* and *paul.cpp*, each of which is responsible for printing a single name.

Example 1-1. Source code for libjohnpaul

johnpaul/john.hpp

```
#ifndef JOHN_HPP_INCLUDED
#define JOHN_HPP_INCLUDED

void john(); // Prints "John, "

#endif // JOHN_HPP_INCLUDED
```

johnpaul/john.cpp

```
#include <iostream>
```

```
{
    std::cout << "John, ";
}
```

johnpaul/paul.hpp

```
#ifndef PAUL_HPP_INCLUDED
#define PAUL_HPP_INCLUDED

void paul(); // Prints " Paul, "

#endif // PAUL_HPP_INCLUDED
```

johnpaul/paul.cpp

```
#include <iostream>
#include "paul.hpp"

void paul()
{
    std::cout << "Paul, ";
}
```

johnpaul/johnpaul.hpp

```
#ifndef JOHNPAUL_HPP_INCLUDED
#define JOHNPAUL_HPP_INCLUDED

void johnpaul(); // Prints "John, Paul, "

#endif // JOHNPAUL_HPP_INCLUDED
```

johnpaul/johnpaul.cpp

```
#include "john.hpp"
#include "paul.hpp"
#include "johnpaul.hpp"

void johnpaul()
{
    john();
```

The source code for *libgeorgeringo* is presented in Example 1-2. The public interface of *libgeorgeringo* consists of a single function, `georgeringo()`, declared in the header *georgeringo.hpp*. As you might well guess, the function `georgeringo()` is responsible for printing:

```
George, and Ringo
```

to the console. Again, the implementation of `georgeringo( )` is split between two source files, *george.cpp* and *ringo.cpp*.

Example 1-2. Source code for libgeorgeringo

georgeringo/george.hpp

```
#ifndef GEORGE_HPP_INCLUDED
#define GEORGE_HPP_INCLUDED

void george(); // Prints "George, "

#endif // GEORGE_HPP_INCLUDED
```

georgeringo/george.cpp

```
#include <iostream>
#include "george.hpp"

void george()
{
    std::cout << "George, ";
}
```

georgeringo/ringo.hpp

```
#ifndef RINGO_HPP_INCLUDED
#define RINGO_HPP_INCLUDED

void ringo(); // Prints "and Ringo\n"

#endif // RINGO_HPP_INCLUDED
```

```
#include <iostream>
#include "ringo.hpp"

void ringo()
{
    std::cout << "and Ringo\n";
}
```

georgeringo/georgeringo.hpp

```
#ifndef GEORGERINGO_HPP_INCLUDED
#define GEORGERINGO_HPP_INCLUDED

// define GEORGERINGO_DLL when building libgerogreringo.dll
# if defined(_WIN32) && !defined(__GNUC__)
#  ifdef GEORGERINGO_DLL
#   define GEORGERINGO_DECL _  _declspec(dllexport)
#  else
#   define GEORGERINGO_DECL _  _declspec(dllimport)
#  endif
# endif // WIN32

#ifndef GEORGERINGO_DECL
# define GEORGERINGO_DECL
#endif

// Prints "George, and Ringo\n"
#ifdef __MWERKS__
# pragma export on
#endif

GEORGERINGO_DECL void georgeringo();
#ifdef __MWERKS__
# pragma export off
#endif

#endif // GEORGERINGO_HPP_INCLUDED
```

georgeringo/ georgeringo.cpp

```
#include "george.hpp"
#include "ringo.hpp"
#include "georgeringo.hpp"
```

```
    george();
    ringo();
}
```

The header *georgeringo.hpp* contains some complex preprocessor directives. If you don't understand them, that's okay. I'll explain them in Recipe 1.4.

Finally, the source code for the executable *hellobeatles* is presented in Example 1-3. It consists of a single source file, *hellobeatles.cpp*, which simply includes the headers *johnpaul.hpp* and *georgeringo.hpp* and invokes the function `johnpaul()` followed by the function `georgeringo()`.

Example 1-3. Source code for hellobeatles

```
hellobeatles/ hellobeatles.cpp

#include "johnpaul/johnpaul.hpp"
#include " georgeringo/ georgeringo.hpp"

int main()
{
    // Prints "John, Paul, George, and Ringo\n"
    johnpaul();
    georgeringo();


}
```