INSTANT

Short | Fast | Focused

MinGW Starter

Develop, debug and profile your C++ applications using the MinGW open source software

Foreword by Vitaly Lipatov, CEO, Etersoft

Ilya Shpigor

[PACKT]
PUBLISHING

Instant MinGW Starter

Develop, debug and profile your C++ applications using
the MinGW open source software

Ilya Shpigor

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Instant MinGW Starter

First published: January 2013

Production Reference: 1210113

Credits

Author

Ilya Shpigor

Reviewer

Kyle Schwarz

Acquisition Editor

Edward Gordon

Commissioning Editor

Maria D'souza

Technical Editor

Ankita Meshram

Project Coordinator

Amigya Khurana

Proofreader

Maria Gould

Production Coordinator

Prachali Bhiwandkar

Cover Work

Prachali Bhiwandkar

Cover Image

Conidon Miranda

Foreword

"Yoda's aphorism ("There is no 'try'") sounds neat, but it doesn't work for me. I have done most of my work while anxious about whether I could do the job, and ensure that it would be enough to achieve the goal if I did. But I tried anyway, because there was no one but me between the enemy and my city. Surprising myself, I have sometimes succeeded."

Richard Matthew Stallman,
Launcher of the GNU Project and founder of the Free Software Foundation

Sometimes it seems that the world is limited to just the Microsoft Windows operating system and that the developing of complex applications is impossible without the latest version of MS Visual Studio. But there are open houses where there is no need to look at the sky through the windows. There are a lot of development environments that are different from commonly used ones, and this variety allows us to feel the world of the software in its entirety.

PCs are losing their significance increasingly and are yielding the personal computing device's role to tablet computers and smartphones. The world does not consist of only one hardware architecture nowadays. The different CPU architectures and operating systems for supporting them are available now. The ability to develop cross-platform applications is important too.

Many software development tools have been created as a part of the GNU Project since 1984 when the project started. The GNU compiler collection is a part of these software development tools.

Minimalist GNU for Windows (MinGW) is a software port of the GNU Toolchain for Microsoft Windows operating system. Its minimalism means that MinGW doesn't provide the whole POSIX compatible environment. In the meantime it doesn't yield to its ancestor for Unix-based systems but allows you to feel Unix philosophy.

MinGW allows open source software developers to port their software to the Windows operating system. MinGW software with any of the well-known cross-platform framework integration allows you to develop cross-platform applications even if you don't care about this feature. This feature provides a great competitive advantage if your customer decides to change his or her computing platform to another suddenly. You don't need a special Linux version or additional developer team in this case. Just make minor changes in your software and that's all you need.

The MinGW software allows you to use plenty of open source C and C++ libraries that integrate with MinGW well. You can develop applications with more features and reuse source code thanks to these libraries. Often open source libraries' licenses allow you to choose if your application will be a free software or a proprietary one.

The world of command-line interface, Makefiles, and build systems can be very enthralling like other new things. Compiling your program with new tools always leads to surprises. It finds faults in places with seemingly clean code. Try to calm it down.

You do not get lost. You can integrate MinGW software with an integrated development environment (IDE), such as Code::Blocks, Qt Creator, and even Eclipse if you will tolerate Eclipse's slowness.

Good luck! And don't forget to use a control version system for your source code. The frequent commits are a pledge of quick bugs searching that has been added to the developed source code with any new feature or code correction.

This book allows you to quickly start new software project development thanks to detailed explanation on how to create a necessary development environment. You will know how to download necessary tools, choose user interface library and compile applications. At first, it is important to understand where to start. I believe that this book will dispel your doubts and you will understand that these things are actually simple.

Try it!

Vitaly Lipatov
CEO, Etersoft

About the Author

Ilya Shpigor is a software developer in a flight simulator manufacturing company in Saint Petersburg, Russia. His work is in developing real-time computing systems that work under heavy computational loads. Ilya prefers to use open source software products, such as the Linux operating system and GNU toolchain, for his daily tasks.

He has participated in ALT Linux distribution and Wine open source software development before his current job.

Ilya has experience in cross-platform software development, porting applications to other computing platforms, and real-time computing systems design. He is interested in automating routine tasks and researching the capacities of different programming languages to solve specific problems.

About the Reviewer

Kyle Schwarz has been working on Linux machines for over 7 years and has a deep understanding of low-level system functions. He is a detail-oriented individual, who has experience in many aspects of computer software and hardware. He enjoys working with all technology platforms and is constantly expanding his experience in this field.

He has worked for several major companies on projects that involved FFmpeg, Windows BATCH scripting, Linux BASH scripting, web development, web design, Windows Installer scripting, and much more. He currently operates `zeranoe.com` and does contract work for companies and individuals.

> I would like to thank my Dad for teaching me the value of hard work. His guidance and leadership have been the most valuable aspects of my life.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

packtLib.packtpub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

- ✦ Why Subscribe? Fully searchable across every book published by Packt
- ✦ Copy and paste, print and bookmark content
- ✦ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Instant MinGW Starter

Welcome to *Instant MinGW Starter*.

This book has been especially created to provide you with all the information that you need to start developing applications with MinGW software. You will get to know the basic skills to work with MinGW software and discover some tips and tricks for using it.

This book contains the following sections:

So, what is MinGW? – This section describes what MinGW actually is, its common components, and what you can do with it.

Installation – In this section you will learn how to download and install MinGW software with minimum fuss and then set it up so that you can use it as soon as possible.

Quick start – Creating your first MinGW project – This section will show you how to create a new application with MinGW software step by step. It will be the basis of most your work with MinGW.

Top features you'll want to know about – Here, you will learn how to perform some tasks with the most important features of MinGW. By the end of this section you will be able to use the GNU Make utility for effectively building your projects, importing existing Visual C++ projects to MinGW software, developing applications based on cross-platform GUI libraries, and configuring several Open Source IDEs with MinGW.

People and places you should get to know – Every Open Source project is centered around a community. This section provides you with many useful links to the project page and forums, as well as a number of helpful articles and tutorials on MinGW software.

So, what is MinGW?

Minimalist GNU for Windows (MinGW) is a native software port of the GNU tool chain for the Microsoft Windows operating system. The base components of MinGW software are compiler, linker, and assembler. This minimal tool set is enough to start developing applications. But MinGW contains some service utilities to make the developing process more effective:

- ✦ GNU Make
- ✦ GNU Debugger (GDB)
- ✦ GNU profiler
- ✦ Compiler for Windows resource files
- ✦ Header files and libraries for Windows API
- ✦ Collection of archives and packers

These components allow you to develop native 32-bit Windows applications without any proprietary third-party software. All components of MinGW software are produced under GNU General Public License and therefore this is a free software that you can download, use, and change as you want.

You can develop applications in C, C++, Java, Objective C, Fortran, and Ada programming languages with MinGW software. C++ application development will be described in this book, which is more typical for MinGW usage.

Besides developing new applications, you can import existing Visual C++ projects to MinGW software. It is easy to integrate MinGW with well-known third-party libraries such as DirectX, Boost, Qt, GTK, OpenGL, and SDL. If you are using any of these libraries, you can compile your application with MinGW.

MinGW software is very useful for importing Unix and Mac applications to Windows native code. It provides the same instruments that Unix and Mac developers have used in most cases. Also, you can import your MinGW-based applications to any computing platform supported by the GNU toolchain. Therefore, MinGW software is a great instruments' set for developing cross-platform applications.

Another benefit of MinGW software is modular organization. You can replace most components of the GNU toolchain with your favorite instruments (for example, debugger, profiler, or build automation system). These instruments will be integrated with existing components without any problems. Usage of the MinGW software is the first step to collecting your own developer's instruments' set for comfortable work.

The compiler efficiency is one of most important parameters for software developers. There are a lot of C++ compilers' benchmarks that are available on the Internet. Unfortunately for us, developers of proprietary compilers are not interested in objective researches of this kind. Fair comparison of available compilers is impossible because of this.

The MinGW compiler efficiency is abreast to proprietary compiler efficiency today according to benchmarks of independent software developers. You can find one of them at the following website:

```
http://www.willus.com/ccomp_benchmark.shtml?p9+s6
```

The MinGW software releases are more frequent than the proprietary compilers' releases. This means that MinGW is developed and improved more dynamically. For example, the standard features of C++11 have been supported by the GCC compiler earlier than the Visual Studio one. You can find these features at the following website:

```
http://wiki.apache.org/stdcxx/C++0xCompilerSupport
```

Notice that the GNU toolchain is a product of Unix culture. This culture is earlier than GUI applications with access to any function through menus, dialogs, and icons. Unix software has been developed as a suite of little stand alone utilities. Each of these performs only one task, but this execution is optimized very well. Therefore, all these utilities have a text-based interface. This provides the simplest intercommunication mechanism with a command line shell and saves the system resources.

If the idea of a text-based interface scares you, be relieved because there are a lot of **Integrated Development Environments** (**IDE**) that support MinGW.

Installation

There are several ways to install MinGW software on your computer. For example, you can compile whole MinGW software by yourself, or you can just install MinGW software distribution with a few clicks. The following steps are the simplest and quickest guide to install MinGW software.

Step 1 – What do I need?

You need the following configurations on your computer to install MinGW software according to this guide:

- ✦ Disk space of 500 MB
- ✦ An operating system of any version of Microsoft Windows since Windows XP or newer
- ✦ Internet connection

Step 2 – Downloading MinGW

Download a self-extracting archive with the latest version of the MinGW software distribution from the following web page:

```
http://nuwen.net/mingw.html
```

You will find two types of distribution here: one with Git and one without Git. **Git** is an open source distributed revision control system. I suggest you install the version with Git because it contains Bash command shell. This is a comfortable alternative for the standard Windows Command Prompt. For example, the Bash shell provides the autocomplete function that will complete the typed commands and pathnames by pressing the *Tab* key. Also the command history is available by pressing up and down arrows.

Step 3 – Extracting and configuring

Run the self-extracting archive. Specify the target directory and click on the **Extract** button.

Suppose that you choose `C:\` as the target directory. The archive will be extracted to `C:\ MinGW`. I strongly recommend you not to install MinGW software in `C:\Program Files`. There are problems with paths containing spaces.

Run the `set_distro_paths.bat` script in `C:\MinGW` after the archieve extraction. It will add the MinGW software directory to the `PATH` system variable for integration with the Windows Command Prompt and Bash shell. This script does not work properly on Windows Vista and Windows 7. Check the MinGW directory existence in the `PATH` variable after executing it.

What do you have to get?

Congratulations! You have got the linker, C, and C++ compilers on your computer with header files and libraries for Windows API. Boost, GLEW, SDL, PCRE, Free Type, Vorbis, and many more libraries have been installed too. Moreover, there is profiler, Bash shell, Git, and other utilities.

There are several other ways to install MinGW software. One of them may be more suitable for your goals.

Alternative ways to install MinGW

The installation process described earlier refers to the unofficial distribution of the MinGW software with additional libraries and utilities. It may seem doubtful for users accustomed to proprietary software, but this is common practice for open source users. The third-party distributions are more usable and complete than official ones in some cases. This is achieved by integrating several relative open source products into one distribution. GNU Linux distribution is a typical sample of this practice.

You can download and install the official distribution of MinGW software from the following developers' website:

```
http://www.mingw.org
```

I recommend you use the `mingw-get` installer application with a text-based interface. You can get a list of all the available packages by executing the following command:

```
$ mingw-get list
```

Execute the following command to install the necessary packages (for example, GCC, G++, GDB):

```
$ mingw-get install gcc g++ gdb
```

A more detailed instruction manual is available at the official MinGW website. You can simply install extensions for MinGW software using the `mingw-get` application.

The 64-bit MinGW software version is available from the `MinGW-w64` fork. Fork is an alternative branch of mainstream software development. The goal of any fork is to achieve specific software features. MinGW-w64 is a completely different software package than MinGW with its own staff of developers. However, the basic principles of MinGW and MinGW-w64 are the same. All knowledge gained in this book you can apply to MinGW-w64 software. The following website is for the MinGW-w64 project:

```
http://mingw-w64.sourceforge.net
```

You can download the archive with MinGW software from here and unpack them. After unpacking you will get a ready-to-use MinGW software.

The following is the website of a MinGW-w64 software's unofficial distribution:

```
http://tdm-gcc.tdragon.net
```

This distribution provides a more flexible configuration of the installable components than the official one. The installation will be performed through the standard Windows Installation Wizard application.

MinGW software is supplied with some open source IDE. For example, such integrated product is available on Code::Blocks, official website `http://www.codeblocks.org`.

Quick start – Creating your first MinGW project

Let's create a simple typical C++ Windows application from scratch. Its interface will consist of a dialog window with two buttons. A click on the first button leads to a display of a message while a click on the second button leads to the application termination. The application contains a resource file with Windows Controls captions, sizes, styles, and fonts.

Step 1 – Adding source files

First of all, you must create a source C++ file and name it `main.cpp`. This file will contain the main function:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "resource.h"


BOOL CALLBACK DialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch(uMsg)
    {
        case WM_CLOSE:
            EndDialog(hwndDlg, 0);
            return TRUE;

        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case IDC_BTN_QUIT:
                    EndDialog(hwndDlg, 0);
                    return TRUE;

                case IDC_BTN_TEST:
                MessageBox(hwndDlg, "Message text", "Information",
MB_ICONINFORMATION);
                    return TRUE;
            }
    }

    return FALSE;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nShowCmd)
{
    return DialogBox(hInstance, MAKEINTRESOURCE(DLG_MAIN), NULL,
(DLGPROC)DialogProc);
}
```

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

The first line is the definition of the `WIN32_LEAN_AND_MEAN` macro, which disables the inclusion of rarely-used Windows header files. The next two lines include the header file with the Windows API functions' declaration and the header file with application resources identifiers.

The `DialogProc` function processes messages sent to our modal dialog. These messages contain information about events that occurred. The message identifier is passed to the function in the `uMsg` parameter. The `wParam` and `lParam` parameters are used for additional message-specific information. The `hwndDlg` parameter defines the dialog window that has received the message.

The `DialogProc` function processes the following messages:

1. `WM_CLOSE`: This message is caused by a click on the standard close window button.
2. `WM_COMMAND` with the `IDC_BTN_QUIT` parameter: This message is caused by a click on the Quit button.
3. `WM_COMMAND` with the `IDC_BTN_TEST` parameter: This message is caused by a click on the Test button.

The `WM_CLOSE` and `WM_COMMAND` with `IDC_BTN_QUIT` parameter messages causes the application to terminate. The `WM_COMMAND` with the `IDC_BTN_TEST` parameter message causes the standard message box displaying.

The function which is defined next is `WinMain`. This function will be called when the application launches. The `DialogBox` Windows API function is called here to create a modal dialog window. We pass the `hInstance` variable to this function with a handle to the module whose executable file contains the resources of the created dialog. These resources are read-only embedded data in a binary file.

Next, the `DialogBox` function parameter is a pointer to the null-terminated string that specifies the dialog template in the resource data. The `MAKEINTRESOURCE` macro is used here to convert the `DLG_MAIN` identifier of the integer type to the null-terminated string. This identifier is defined in the `resource.h` header file.

The third parameter of the `DialogBox` function is the handle of the parent window that owns the dialog window. This is equal to the `NULL` value in our case that means the absence of a parent window.

The last parameter of the function is a pointer to the dialog window procedure to process messages. We pass the pointer to the `DialogProc` function for this parameter.

9

User interface elements and their parameters can be described in the resource file. All data from this file will be embedded into executable files and these will be available when the application runs.

Let's add this resource file to our project (`resource.rc`):

```
#include "resource.h"

DLG_MAIN DIALOGEX 6, 5, 138, 75

CAPTION "Typical Windows Application"

FONT 10, "Tahoma"

STYLE 0x10CE0804

BEGIN
  CONTROL "&Message", IDC_BTN_TEST, "Button", 0x10010000, 46, 15, 46,
15
  CONTROL "&Quit", IDC_BTN_QUIT, "Button", 0x10010000, 46, 45, 46, 15
END
```

You can see the inclusion of the `resource.h` header in the first line of the resource file. The user interface element identifiers are defined in this header file. These identifiers are used to provide access from C++ code to resource data.

The `DLG_MAIN` element of the `DIALOGEX` type is defined in the next line. This element represents the dialog template with the position and size dialog window parameters. All statements in the next lines define the appearance of the dialog box and its elements.

The next line of the resource file contains the `CAPTION` statement. This statement defines the title of the dialog box. The `FONT` statement defines the font size and typeface for the dialog text font. The `STYLE` statement defines the window style of the dialog box.

The dialog box buttons are defined between the `BEGIN` and `END` statements. These parameters are defined for each button as follows:

✦ Type of the user interface element (this is equal to `CONTROL`)
✦ Caption of the element
✦ Element identifier
✦ Class of the element (this is equal to `Button`)
✦ Window style of the element
✦ Position (x, y) and size (width, height)

The third file is a header for the binding resource identifiers and C++ code (`resource.h`):

```
#include <windows.h>

#define DLG_MAIN 100
#define IDC_BTN_TEST 101
#define IDC_BTN_QUIT 102
```

The dialog box and buttons' identifiers are specified here.

Step 2 – Adding a Makefile

The compilation rules are required to build our application. The rules describe algorithms to compile sources and link object files together to assembly executable files and libraries. This kind of algorithm is present in common IDEs, such as Visual Studio. However, it is often hidden inside the graphical user interface and is not available for change. You have the ability to control each step of the building application algorithm with the GNU Make utility.

You can perform each compilation step manually by calling the compiler and linker from the command line interface. However, the rules in the GNU Make utility file can automate these operations.

This is the simplest variant of the rules of the GNU Make utility file to build our application (`Makefile`):

```
win32-app.exe: main.cpp resource.rc.res.o
  g++ -o win32-app.exe main.cpp resource.rc.res.o

resource.rc.res.o: resource.rc
  windres -o resource.rc.res.o -i resource.rc

clean:
  rm -f *.o win32-app.exe
```

Notice that there is a tabulation under each command. Tabulation and spaces are not the same for the GNU Make utility and this is often subjected to criticism.

The Makefile syntax will be described in detail in the later part of this book. It consists of targets (specified under the colon), and commands for producing these targets at the next line. The file list after the colon consists of files on which the target depends. These files are called prerequisites. The target will be rebuilt by the `make` command if one of its prerequisite files has been changed.

The following targets are specified in this `Makefile`:

- ✦ The final executable file, that is, `win32-app.exe`
- ✦ The object file with application resources, that is, `resource.rc.res.o`
- ✦ The utility target to remove temporary files, that is, `clean`

11

The MinGW C++ compiler application name is g++. The compiler for the Windows resource files is `windres`. Each GNU utility has detailed information about command line options and developers' feedback. Running the GNU utility from the Windows command prompt or Bash shell with the `--help` option will lead to displaying this information. This is the example of the same GNU utility run.

```
$ g++ --help
```

The `clean` utility target is required to remove all files generated by the compiler and linker. The `rm` command is called to perform this task. You can use the `clean` target to rebuild the application after changing source files. Just perform this target and then build your application.

Step 3 – Compiling and linking

Now we are ready to compile our first application with MinGW software. First of all, you must run the command shell. There are several ways to do this. The simplest one is by launching the command shell from the Windows **Start** menu. Just type the `cmd` or `bash` command in the menu's **Search** field. Furthermore, there are a lot of file managers with integrated command shells which you will be comfortable to work with. Far Manager is one of these.

You will see a window with the command line shell. Several useful commands for directory navigation have been installed with MinGW software:

✦ `cd <dir>`: This command changes the current directory to the specified one.

For example, to change the directory to `C:\Projects`, type the following command line:

```
$ cd /c/Projects
```

✦ `pwd`: This command writes the absolute pathname of the current directory

✦ `ls`: This command lists the current directory contents

Change the current directory to the project one. The `C:/` path equals to the `/c` path in the Bash shell. This is due to the specific Unix environment's integration with Windows filesystems. Type the `make` command after changing the current directory. This is all that you need to do for compiling and linking applications. You get the executable binary after the GNU Make utility is executed successfully. Retype the `make` command to rebuild the application after changing any of the source files. It may be helpful to remove all the files, which were already generated by the compiler and linker, by using the `make clean` command before rebuilding the application.

Top features you'll want to know about

Several useful features of MinGW software will be described in detail in this section. You will get detailed knowledge about the most commonly used GNU toolchain utilities, such as C++ compiler, GNU Make, GNU Debugger, and the GNU `gprof` profiler. Also the integration of MinGW software with several well-known GUI libraries and IDE systems will be described.

1 – Make utility usage

We created a simple `Makefile` in the previous section. Let's explore the GNU Make utility's behavior and the syntax of Makefile in more detail.

You can run the GNU Make utility from the command shell with the `make` command. It will search one of the `Makefile`, `GNUmakefile`, or `makefile` named files and start to build the first line target. For changing this behavior, type the following command:

$ make -f OtherMakefile other_target

This leads to reading `OtherMakefile` as input file of rules and executing commands to build `other_target`.

In the previous section we created the following `Makefile`:

```
win32-app.exe: main.cpp resource.rc.res.o
  g++ -o win32-app.exe main.cpp resource.rc.res.o

resource.rc.res.o: resource.rc
  windres -o resource.rc.res.o -i resource.rc

clean:
  rm -f *.o win32-app.exe
```

This works, but it has some problems. First of all the `main.cpp` and `resource.rc.res.o` files are specified several times. You must rename these in several places if one of these has been changed. This violates one of the most important principles of software development, **Don't repeat yourself** (**DRY**). Variables can help you to avoid this problem. The GNU Make variables are often used to save constant values and change behavior during the build process. Our `Makefile` with variables will be as follows:

```
EXE=win32-app.exe
CPP=main.cpp
RES=resource.rc

$(EXE): $(CPP) $(RES).res.o
  g++ -o $(EXE) $(CPP) $(RES).res.o
```

```
$(RES).res.o: $(RES)
   windres  -o $(RES).res.o -i $(RES)

clean:
   rm -f *.o *.exe
```

The variable definition contains the name and variable value after the equals sign. You must use the dollar sign before the variable name within parentheses to access its value after the variable definition.

The source, resource, and executable files are defined as variables in our example. Now you should change the variable value in one place only to rename any of these files.

GNU Make allows the user to define variables, but there are several special automatic variables. These variables are computed for each rule that is executed. You can use the `$@` variable to specify the previously defined target and the `$^` variable to specify all the prerequisites of this target. After doing all this `Makefile` will look as follows:

```
CPP=main.cpp
RES=resource.rc

win32-app.exe: $(CPP) $(RES).res.o
   g++ -o $@ $^

$(RES).res.o: $(RES)
   windres -o $@ -i $^

clean:
   rm -f *.o *.exe
```

The automatic variables can be used just like the user-defined ones. Now output files of g++ and the `windres` command are declared with the `$@` variable and source files with the `$^` one.

Suppose you want to link your application with a third-party library. The library is supplied with the header file and a dynamic-link or static-link library. You must inform the compiler about these files' paths and the library to link with. The simplest way to do it is using the GNU Make environment variables.

An environment variable is a variable that comes from the command shell in which GNU Make runs. There are several predefined environment variables with standard names. You can add custom environment variables from your command shell, but the most common practice is to operate with the standard one in `Makefile`. Environment variables affect target producing commands in `Makefile` (for example, additional compiler options are usually passed through environment variables).

The following Makefile is a linking example with the static-link boost program options library (the file path of the library is `C:\MinGW\lib\libboost_program_options.a`):

```
OBJ=main.o options.o
RES=resource.rc

CXXFLAGS+=-IC:\MinGW\include
LIBS+=-LC:\MinGW\lib -lboost_program_options

win32-app.exe: $(OBJ) $(RES).res.o
  g++ -o $@ $^ $(LIBS)

$(RES).res.o: $(RES)
  windres -o $@ -i $^

clean:
  rm -f *.o *.exe
```

`CXXFLAGS` is a predefined environment variable that contains command line options for the C++ compiler. This variable can be used to specify the paths of additional header files required for source compilation. Paths to headers are specified there with the `-I` prefix. You can specify several paths separated by a space as follows:

```
CXXFLAGS+=-IC:\first_include_path -IC:\second_include_path
```

`LIBS` is a simple variable with a list of static-link libraries to link with. This variable is passed to the C++ compiler explicitly in the `win32-app.exe` target producing rule. Libraries to link are specified with the `-l` prefix, and without the first three letters (`lib`), as well as the suffix (`.a`). The full path to linked libraries must be specified too with the `-L` prefix.

The following command is used to print all the predefined GNU Make environment variables:

`$ make -p`

Notice that the additional `options.cpp` source file has been added to the project in this example. Also the `CPP` variable has been renamed to `OBJ` and contains object files list now. GNU Make will compile object files from source files automatically if they have same names (for example, `main.o` and `main.cpp`).

The following Makefile is a linking example with a dynamic-link `zlib` library (the file path of the library is `C:\MinGW\git\bin\libz.dll`):

```
OBJ=main.o
RES=resource.rc

CXXFLAGS+=-IC:\MinGW\include
```

15

```
LIBS+=-LC:\MinGW\git\bin -lz

win32-app.exe: $(OBJ) $(RES).res.o
  $(CXX) -o $@ $^ $(LIBS)

$(RES).res.o: $(RES)
  windres -o $@ -i $^

clean:
  rm -f *.o *.exe
```

Dynamic-link libraries are described in the `LIBS` variable by the same rules as for static-link libraries.

`CXX` is an environment variable with the C++ compiler's application name. This is equal to `g++` by default.

You can call the GNU Make utility for the `Makefile` subdirectory's processing. This is an example of the root project directory, `Makefile`, that performs the GNU Make utility for subdirectories (`foo` and `bar`):

```
SUBDIRS = foo bar

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
  $(MAKE) -C $@
```

The `.PHONY` rule is a special rule. It is used to specify the fact that the target is not a file. This is required in our case because subdirectories always exist and the GNU Make will not rebuild targets whose files already exist. `MAKE` is an environment variable with GNU Make application name. This is equal to `C:\MinGW\bin\make` by default if the installation path of MinGW software is equal to `C:\MinGW`.

Makefiles can include comments. All lines starting with the # symbol will be considered by GNU Make as comments.

Makefiles can be complex and consists of several separated files. To include external file content in `Makefile` use `include` directive, for example:

```
include Makefile.mingw
```

GNU Make is a great instrument for compiling small projects with a couple of source files and headers. There are more powerful instruments for building complex applications that consist of several libraries and executables. Some of them are based on the GNU Make utility and produce Makefiles as the output (for example, GNU Autotools and CMake). I strongly recommend you to study and use one of these for your daily projects.

2 – Compiler options

The MinGW compiler behavior is highly dependent on command line options. These options are usually set through GNU Make environment variables. The following is a list of commonly used predefined environment variables:

- `CC`: This variable sets the C compiler in use
- `CFLAGS`: This variable sets the C compiler command line options
- `CPPFLAGS`: This variable sets the C PreProcessor flags, which are used by C and C++ compilers
- `CXX`: This variable sets the C++ compiler in use
- `CXXFLAGS`: This variable sets the C++ compiler command line options
- `LD`: This variable sets the linker in use
- `LDFLAGS`: This variable sets the linker command line options
- `LDLIBS`: This variable sets libraries to link

The following is a list of commonly used MinGW C and C++ compilers' command line options:

- `-o file-name`: This gives the name of the compiler output file.
- `-c`: This is used for compilation only. The object files will be created without further linking.
- `-Dname=value`: This defines the C preprocessor macro with a specified name and value. The `=value` part can be skipped. The default value (equal to 1) will be used instead. The result of this option will be the same as the following declaration in the source file:

  ```
  #define name value
  ```

- `-llibrary-name`: This uses the specified dynamic-link or static-link library for linking.
- `-Idirectory`: This uses the directory to search headers by compiler.
- `-Ldirectory`: This uses the directory to search dynamic-link libraries for linking.
- `-g`: This produces debugging information to be used by GDB debugger.
- `-pg`: This generates extra code to write profiling information for the `gprof` performance analyzer.

- ✦ `-Wall`: This shows all the compiler's warning messages.

- ✦ `-On`: This sets the compiler's optimization level to an `n` value. The available levels are `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast`. The `-O0` option disables all optimizations. The `-O3` option provides the maximum optimization of code size and execution time. The `-Os` option optimizes for code size. The `-Ofast` option is the `-O3` option with non-accurate math calculation.

- ✦ `-std=standard`: This uses the specified language standard (example for C++ standard: `-std=C++11`).

3 – Importing the existing Visual C++ project

You can use MinGW software even if you already have your project developed with Visual C++. The importing process from Visual C++ to MinGW is quite simple. All you need is to remove unusable headers and create Makefiles to compile and link existing C++ sources.

This process will be described in detail in this section by an example application. The application is a simple command-line archiving utility based on the zlib compression library. The boost library is also used for command-line option parsing. The application consists of the `core` static-link library and the `z_pack` executable. This separation by library and executable is required for complex project linking example with several modules.

You can see the Visual C++ project structure in the **Solution Explorer** window in the following screenshot:
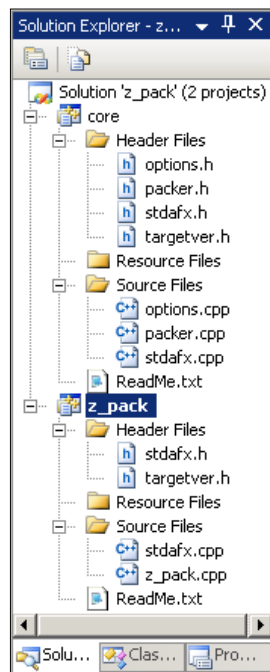
First of all, let's look at the source code of the `core` library. This library contains the implementation of most of the application's functionality. The command-line parsing and compressing mechanisms are implemented here. Each of these mechanisms are encapsulated in a separate class. The `Options` class implements the command-line options' parsing. The `Packer` class implements the compressing and decompressing algorithms.

Let's look at the `Options` class. The following code shows the class declaration in the `options.h` file:

```
namespace po = boost::program_options;
class Options
{
public:
    Options(int argc, char* argv[]);

    std::string GetString(std::string option_name);
    po::options_description& GetDescription();
    bool IsUnzip();
    bool IsComplete();

private:
    po::variables_map options_;
    po::options_description description_;
};
```

The following code shows the constructor definition of the `Options` class in the `options.cpp` file:

```
Options::Options(int argc, char* argv[])
{
    description_.add_options()
        (kHelp.c_str(), "produce help message")
        (kInFile.c_str(), po::value<string>(), "input file name")
        (kOutFile.c_str(), po::value<string>(), "output file name")
        (kUnzip.c_str(), "unzip the archive");

    try
    {
        po::store(po::parse_command_line(argc, argv, description_),
                    options_);
    }
    catch(...)
    {
        cout << GetDescription() << "\n";
        exit(1);
    }
}
```

19

The constructor's input parameters are the count of command-line arguments and the vector with arguments' values. First of all the object from the boost library of the `options_description` class with the `description_` name is configured here with the available command-line arguments. This object is a field of the `Options` class.

All available command-line arguments are defined as global constants of the `string` type in the `options.h` file:

```
static const std::string kHelp = "help";
static const std::string kInFile = "in";
static const std::string kOutFile = "out";
static const std::string kUnzip = "unzip";
```

These arguments with descriptions are passed to the `add_options` method of the `description_` object to configure it. Now this object stores information about available command-line arguments. After that, arguments passed to the constructor input parameters are parsed and saved to the variables map. This map is an object from the boost library of the `variables_map` class with the `options_` name. This object is a field of the `Options` class.

The available command-line arguments are printed if any exception occurs in the arguments' parsing operation. The application's termination will be caused with the `exit` function in this case.

The following code is the `GetString` method's definition of the `Options` class in the `options.cpp` file:

```
string Options::GetString(string option_name)
{
    if ( options_.count(option_name) == 0 )
        return string();

    return options_[option_name].as<string>();
}
```

The method returns the specified program option value of the `string` type or an empty string if the option has not been passed as a command-line argument. I suggest you name command-line arguments as program options after parsing them in the constructor of the `Options` class.

The following three methods of the `Options` class are quite simple. These perform a check on the existence of specific program options or return them:

✦ The `IsComplete` method checks if input and output filenames have been passed as command-line arguments to the application. This method returns the `true` value if the filenames exist.

✦ The `IsUnzip` method checks if the `--unzip` command-line argument exists.

✦ The `GetDescription` method returns the reference to the `description_` field of the `Options` class. You can print available command-line arguments if this object is passed to the `cout` standard output stream.

The second class of the core library is `Packer`. The compressing and decompressing algorithms are implemented here. This class contains the `Compress` and `Decompress` static methods. The static class methods may be very helpful to implement algorithms without state and mutable data. You don't need the object of the class to call static methods.

The following code shows the declaration of the class in the `packer.h` file:

```
class Packer
{
public:
  static void Compress(std::string in_file, std::string out_file);
  static void Decompress(std::string in_file, std::string out_file);
};
```

The following code shows the `Compress` method's definition in the `packer.cpp` file:

```
void Packer::Compress(string in_file, string out_file)
{
  io::filtering_ostreambuf out;
  out.push(io::zlib_compressor());
  out.push(io::file_sink(out_file.c_str(), ios::binary));
  io::copy(io::file_source(in_file.c_str(), ios::binary), out);
}
```

The input parameters of this method are strings with filenames of source and target files.

The compression is performed with the `copy` function from the boost library that copies one stream content to another. The stream of the `file_source` class from the boost library is used in this function as input parameter. The copy function produces the object of the `filtering_ostreambuf` class with the `out` name as the result. The stream of the `file_source` class is anonymous and is created from the source filename and stream type. This is the `binary` type in our case. The `out` object has been configured by the compression filter of the `zlib_compressor` type and anonymous stream of the `file_sink` boost library class. This anonymous stream represents the target file.

After performing the `Compress` method, the compressed file with the specified target filename will be created.

The following code shows the `Decompress` method's definition of the `Packer` class in the `packer.cpp` file:

```
void Packer::Decompress(std::string in_file, std::string out_file)
{
  io::filtering_istreambuf in;
  in.push(io::zlib_decompressor());
  in.push(io::file_source(in_file.c_str(), ios::binary));
  io::copy(in, io::file_sink(out_file.c_str(), ios::binary));
}
```

This method is similar to the `Compress` method, but the filter of the `zlib_decompressor` type has been used here. The decompressed file with a specified target filename will be created after this method is performed.

The rest of the `core` library source files have been automatically generated by Visual C++.

Another solution's project is `z_pack`. This project binds the `core` library classes' functionality in to use them in the complete application. The `z_pack` project consist of one source file with the `main` function implementation and several autogenerated files by Visual C++.

The following code shows the `main` function's definition in the `z_pack.cpp` file:

```
int main(int argc, char* argv[])
{
    Options options(argc, argv);

    If ( ! options.IsComplete() )
    {
        cout << options.GetDescription() << "\n";
        return 1;
    }

    if ( options.IsUnzip() )
        Packer::Decompress(options.GetString(kInFile), options.
GetString(kOutFile));
    else
        Packer::Compress(options.GetString(kInFile), options.
GetString(kOutFile));
    return 0;
}
```

The input parameters of this function are the same as the `Options` class constructor. These are count of command-line arguments and vectors with argument values. First of all the object of the `Options` class with the `options` name is created here. The next operation is checking the source and target filenames for correctness using the `IsComplete` method of the `options` object. The available command-line arguments will be printed and the application will be terminated if this checking fails. The specified compression or decompression operation is performed if command-line arguments are correct. The `IsUnzip` method of the `options` object defines what kind of operation must be performed. The `Compression` and `Decompression` operations perform as per the `Packer` class static methods.

The input parameters for these methods are available from the `GetString` method of the `Options` class execution result.

Full Visual C++ project sources are available in the code bundle uploaded on the Packt website.

You can build this project and check its functionality. The Visual C++ versions of the boost and zlib libraries are required to compile the project. The comfortable distribution of both libraries is available at the following website:

```
http://www.boostpro.com/download
```

You can test the functionality of our example application after building one. Type the following command to compress the existing file:

```
$ z_pack.exe --in test.txt --out test.zip
```

The following command is used to decompress the existing archive:

```
$ z_pack.exe --unzip --in test.zip --out test.txt
```

Now you have the necessary information about our example Visual C++ project to import it to the MinGW software. The following are step-by-step instructions to do this:

1. Create a new directory for the MinGW project version with the `core` and `z_pack` subdirectories.

2. Copy the `options.h`, `options.cpp`, `packer.h`, and `packer.cpp` files to the `core` subdirectory.

3. Copy the `z_pack.cpp` file to the `z_pack` subdirectory.

4. Remove the following line from all the files in the project that have the cpp extension:

    ```
    #include "stdafx.h"
    ```

5. Add the following `Makefile` to the root project directory:

    ```
    SUBDIRS = core z_pack

    .PHONY: all $(SUBDIRS)
    ```

23

```
all: $(SUBDIRS)

$(SUBDIRS):
  $(MAKE) -C $@ clean
  $(MAKE) -C $@
```

This `Makefile` allows you to perform the `make clean` and `make` commands for the `core` and `z_pack` subdirectories. The `make` command will build your application and the `make clean` command will remove output files generated by the compiler and linker.

6. Add the following `Makefile` to the `core` subdirectory:

```
OBJ=options.o packer.o

CXXFLAGS+=-IC:\MinGW\include

libcore.a: $(OBJ)
  ar rcs $@ $^

clean:
  rm -f *.o *.a
```

The `ar` archive utility is used to maintain the `options.o` and `packer.o` object files into the `libcore.a` archive here. This archive will be used for static linking with the `z_pack` executable. The `libcore.a` archive is conceptually the same as the Visual C++ static-link library. You can get additional information about the `ar` archive utility by the executing following command:

`$ ar --help`

7. Add the following Makefile to the `z_pack` subdirectory:

```
OBJ=z_pack.o

MINGW_DIR=C:\MinGW
CXXFLAGS+=-I..\core
LIBS+=-L..\core -lcore -L$(MINGW_DIR)\lib -lboost_program_options
-lboost_iostreams -L$(MINGW_DIR)\git\bin -lz

z_pack.exe: $(OBJ)
  $(CXX) -o $@ $^ $(LIBS)

clean:
  rm -f *.o *.exe
```

Linking with all necessary external libraries (boost and zlib) occurs here because the `ar` archive utility does not perform any linking.

8. Type the `make` command in the root project directory to build our application.

This is all you need to import our example Visual C++ project to the MinGW software. You will get the z_pack.exe executable file after building it.

It is quite simple to change the libcore.a static-link library to the dynamic-link variant one. All you need is to change the Makefile files for the z_pack executable and the core library.

The following Makefile will create the dynamic-link core library variant:

```
OBJ=options.o packer.o

MINGW_DIR=C:\MinGW
CXXFLAGS+=-I$(MINGW_DIR)\include
LIBS+=-L$(MINGW_DIR)\lib -lboost_program_options -lboost_iostreams
-L$(MINGW_DIR)\git\bin -lz

libcore.dll: $(OBJ)
  $(CXX) -shared -o $@ $^ $(LIBS)

clean:
  rm -f *.o *.dll
```

The C++ compiler and linker will be called here unlike the static-link core library variant. Furthermore, the -shared compiler option must be specified to create the dynamic-link library.

The following Makefile is for the z_pack executable to link with the core library dynamically:

```
OBJ=z_pack.o

CXXFLAGS+=-I..\core
LIBS+=-L..\core -lcore

z_pack.exe: $(OBJ)
  $(CXX) -o $@ $^ $(LIBS)

clean:
  rm -f *.o *.exe *.dll
```

Linking with external libraries (boost and zlib) is not required here because it has occurred in the core library building.

You must copy the libcore.dll library and the z_pack.exe executable files into one directory to run the application.

25

It is important to note that libraries, objects, and executable files compiled with Visual C++ and MinGW software are incompatible. This means that you need the MinGW version of all static-link and dynamic-link libraries that you want to link with your application compiled with MinGW software. You must check the existing MinGW version of all third-party libraries that have been used in your project before importing this to MinGW software.

4 – Debugging application

MinGW software contains **GNU Debugger** (**GDB**). This is a standard tool to debug applications developed with MinGW software. You can't use GDB to debug projects compiled with the Visual C++ compiler.

You can install GDB manually if it doesn't exist in your MinGW software distribution. Perform the following instructions to do the same:

1. Download the GDB application archive from the official download page:

    `http://sourceforge.net/projects/mingw/files/MinGW/Extension/gdb`

    Some problems might occur with the latest GDB version's launching. Try to install the previous debugger version if you get errors.

2. Extract the downloaded archive to the MinGW software installation directory. I recommend you to extract the archive to `C:\MinGW\git` if you have installed MinGW software with Git as described in this book.

    After installing GDB you can test its functionality by typing the following command:

    `$ gdb --help`

GDB has been installed correctly if you see the application using information. Now you have the necessary debugger utility to start debugging your MinGW-based application.

Let's debug the example application with GDB. The application is a simple program with a null-pointer assignment.

The following code shows the content of the source file named `segfault.cpp`:

```cpp
#include <string.h>

void bar()
{
  int* pointer = NULL;
  *pointer = 10;
}

void foo()
{
  bar();
```

```
  }

  int main()
  {
    foo();

    return 0;
  }
```

The null-pointer assignment operation occurs in the `bar` function. The `bar` function is called from `foo` and the `foo` function is called from the `main` function.

The following `Makefile` is to compile the example:

```
  OBJ=segfault.o

  CXXFLAGS+=-g

  segfault.exe: $(OBJ)
    $(CXX) -o $@ $^

  clean:
    rm -f *.o *.exe
```

The MinGW C++ compiler doesn't include debug information in output binary files by default. We need to add the `-g` compiler option to do it in this `Makefile`.

You will get the `segfault.exe` executable file with debugging symbols after compilation. It means that GDB can inform you not only about memory addresses but also the names of routines and variables.

Type the following command to start debugging our example application:

`$ gdb segfault.exe`

To start the application with command-line arguments use the `--args` GDB option. For example:

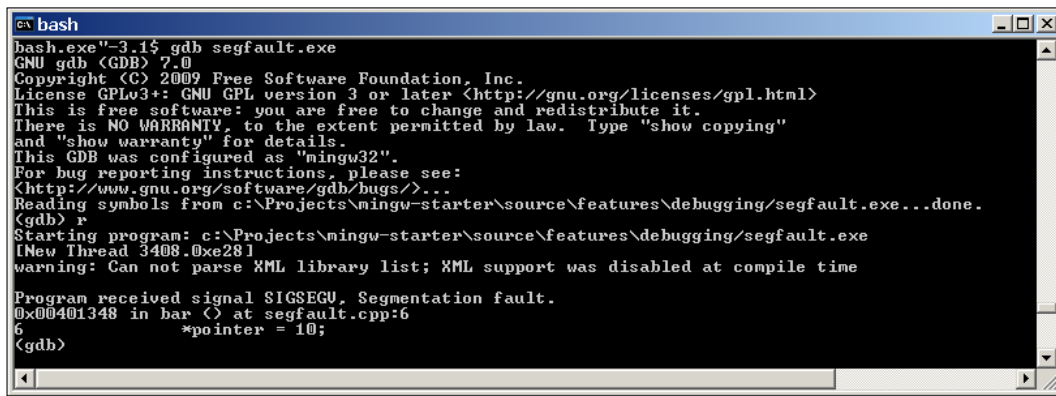**`$ gdb --args z_pack.exe --in test.txt --out test.zip`**

The input and output filenames will be passed to the `z_pack.exe` application in this case.

You will see the GDB command prompt after launching the debugger. Type the `r` command to run the loaded application:

`(gdb) r`

This is the short variant of the `run` command. Most of the GDB commands have common and short variants. All future commands will be described with short variants because they are easy to remember and type.

You will see this program crash message after the application starts running, as shown in the following screenshot:
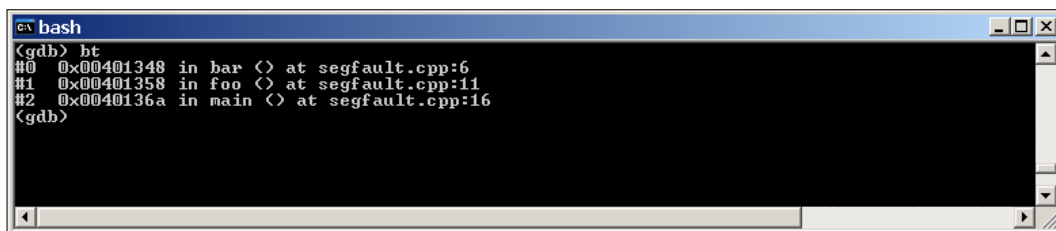

You can see that the program abortion is receiving the SIGSEGV signal by application. The POSIX system sends this signal to process when it makes an invalid virtual memory reference or segmentation fault. Furthermore the source file's line, where the error has occurred, is displayed here. The following is the sixth line of the segfault.cpp file:

```
*pointer = 10;
```

Our example program execution has stopped after the crash. You can interact with the debugger through the command line when the program is stopped. For example, you can get the **stack backtrace** to explore the nested functions' calls. Type the bt command as follows:

(gdb) bt

You will see something like the following screenshot:

You will see called functions' names and source file lines, where these have been called by the program just before it crashed.

Moreover, you can get a list of source file lines in the error place by the l command:

(gdb) l

You will see the `bar` function source code. Now you have enough information to fix the segmentation fault error.

To quit from GDB type the `q` command:

(gdb) q

And type `y` to confirm.

GDB allows you to set breakpoints to stop program execution in a predefined place. There are several breakpoint types. The simplest types are a breakpoint at entry to the function and a breakpoint at a line in a source file. You need to specify the application source files directory to provide access to these for GDB. The command to start the application debugging in this case will look like the following:

$ gdb --directory=. segfault.exe

The `--directory` command-line option defines the source files directory. The current directory has been added in this example (the current directory equals to point symbol).

All the loaded directories are available by typing the following command:

(gdb) show directories

The following command is used to set a breakpoint at the `foo` function from the `segfault.cpp` source file:

(gdb) b segfault.cpp:foo

To set a breakpoint at the fifth line of the `segfault.cpp` source file type the following command:

(gdb) b segfault.cpp:5

A list with information about all the current defined breakpoints is available by using the following command:

(gdb) i b

You can run our example program after setting breakpoints:

(gdb) r

Program execution will be stopped at the first breakpoint in the `foo` function after that. You can continue execution by using the `c` command:

(gdb) c

Type this and the next breakpoint at the fifth line of the `segfault.cpp` source file will be achieved. The backtrace information and source file lines are available at each breakpoint.

29

Watchpoint is a special breakpoint type to detect read and write variable operations. This command is used to set a watchpoint for the `pointer` variable:

`(gdb) aw pointer`

GDB can set such watchpoints if variables have been defined in the current context. This means that you must stop program execution at the `bar` function and then set a watchpoint for its local `pointer` variable. The program can be stopped in the `main` function to set a watchpoint for any global variable.

Moreover, you can configure GDB to display a list of variables with the current values at each breakpoint hit. To add the `pointer` variable in this list type the following command:

`(gdb) display pointer`

The value of the `pointer` variable will be displayed at the next breakpoint program stop.

You can disable a breakpoint of any type to prevent the program from stopping at it. The following is the command to do it:

`(gdb) disable 1`

The specified number is a breakpoint identifier. This identifier is declared in the breakpoints information list.

To enable the breakpoint type the following command:

`(gdb) enable 1`

The breakpoint can be removed if it is no longer needed:

`(gdb) d 1`

There are several useful commands for tracing a program:

✦ Continue to run a program until the control reaches a different source line (analogous of trace into):

   `(gdb) s`

✦ Continue to the next source line in the current stack frame (analogous of step over):

   `(gdb) n`

✦ Continue execution until the function in the selected stack frame returns:

   `(gdb) fin`

These commands can be used after the program has run and been stopped by breakpoint.

You can rebuild your application without debug information to produce release variants of executable files and libraries. But there is a possibility to remove debug information from existing binary files. Use the `strip` command as follows:

`$ strip segfault.exe`

GDB is quite useful to debug abnormal program behavior (segmentation faults, for example). But using the same approaches as the test suite and event logging can complete the GDB capabilities for the algorithms' correctness checking.

5 – Profiling application

MinGW software contains the gprof performance analyzing tool. This instrument can be useful for tracking down an application's bottlenecks. The gprof tool is a part of the GNU Binutils collection and therefore it is present in all MinGW distributions. You can't use gprof to analyze the performance of applications compiled with Visual C++ compiler.

Let's profile an example application with gprof. This application reads bytes from a file to an STL `vector` type container, sorts them, and writes the result to an output file. The application source code is present in the file named `sorting.cpp`.

The following is the `main` function definition:

```
int main()
{
  ReadData();

  SortData();

  WriteResult();

  return 0;
}
```

You can see the basic application algorithm in this function. First of all, the data is reading from the input text file. Then the data is sorted and written to the output file. Each of these steps are implemented in the separate function.

The following code shows the `ReadData` function's definition:

```
void ReadData()
{
  ifstream in_file("source.txt", ios::in | ios::binary);

  copy(istream_iterator<char>(in_file), istream_iterator<char>(),
      back_inserter(gData));
}
```

The stream of the `ifstream` class with the `in_file` name is used here to read the `source. txt` input file content. Then the STL `copy` algorithm is used to copy the `in_file` stream content to the global `gData` container of the `vector<char>` class. We use the iterator of the `istream_iterator` class to access elements of the `in_file` stream in the `copy` algorithm.

The `SortData` function implements the simplest bubble sort algorithm:

```
void SortData()
{
  char temp;
  size_t size = gData.size();

  for (int i = (size - 1); i > 0; i--)
  {
    for (int j = 1; j <= i; j++)
    {
    if (gData[j-1] > gData[j])
    {
      temp = gData[j-1];
      gData[j-1] = gData[j];
      gData[j] = temp;
    }
    }
  }
}
```

This sort algorithm processes the elements of the `gData` container.

The following is the `WriteResult` function definition:

```
void WriteResult()
{
  ofstream out_file("result.txt", ios::out | ios::binary);
  out_file.write(&gData[0], gData.size());
}
```

The stream of the `ofstream` class with the `out_file` name is used here to write the `gData` container content to the output `result.txt` file. The `write` method of the `out_file` stream is called here to perform the file writing operation. An empty file with the `result.txt` name will be created to write if this file already exists. The `sorting.cpp` file is available in the code bundle uploaded on the Packt website.

The following is `Makefile` for the example application:

```
OBJ=sorting.o

CXXFLAGS+=-pg
```

```
sorting.exe: $(OBJ)
  $(CXX) -o $@ $^ $(CXXFLAGS)

clean:
  rm -f *.o *.exe *.out *.dot
```

Profiling information is needed for performance analyzing. This information can be generated by an executable file's extra code. The same kind of extra code creation is an optional feature of the compiler and this feature can be specified by the compiler `-pg` option. This option must be specified for each object file compilation and final linking. The `CXXFLAGS` environment variable is used to do this in our `Makefile`.

You can test our example application after compilation. Just copy any plain text file to project directory with, name it `source.txt`, and run the `sorting.exe` executable file. You will get the `result.txt` file with the sorted source file content after the application finishes its work. Perform the following steps to profile our example application:

1. Launch the application's executable file. After that you will get the `gmon.out` file with the profiling data in the current working directory.

2. Run the gprof utility to interpret the `gmon.out` file information and write the result to the `profile.txt` output file:

   ```
   $ gprof -zq sorting.exe > profile.txt
   ```

These gprof utility options have been used in the preceding:

✦ The `-z` option is required to include all used functions in the output file

✦ The `-q` option causes a call-graph of the program for a more detailed report

You have got a text report with the profiling data in the `profile.txt` file. It can be opened in any text editor. But this text representation of the profiling information is not comfortable to discovery. There are handy tools for visualizing a gprof report.

For visualization we need the following tools:

✦ Python 2.7 interpreter

✦ Python script to convert the gprof report to dot file format

✦ Graphviz package with visualization utilities for dot format

You can download Python version 2.7 from the following official website:

`http://python.org/download`

Python can be installed with the standard Windows Installer. To do this just run the downloaded MSI file.

The script to convert a gprof report text file to dot format is available on the following developer page:

```
http://code.google.com/p/jrfonseca/wiki/Gprof2Dot
```

This script is a free software and you can use, distribute, and modify it as you want. To install this script copy it to `C:\MinGW\bin` or `C:\MinGW\git\bin`.

The Graphviz package is available at the following official website:

```
http://www.graphviz.org/Download_windows.php
```

Graphviz can be installed with Windows Installer in the same way as Python interpreter.

Now you have the necessary scripts and the visualization utility to visualize our profiling results. Perform the following steps to do this:

1. Run the `gprof2dot.py` script to get the dot file:

    `$ gprof2dot.py -s profile.txt > profile.dot`

    This `gprof2dot.py` utility option has been used here.

    The `-s` option removes functions and templates argument information.

    After that you will get the `profile.dot` file with call-graph of the program.

2. Run the `gvedit.exe` application. It is available from the **Start** menu of Windows. Go to **File | Open** from the main menu. Specify the `profile.dot` file.

You will see a call-graph of the program as shown in the following screenshot:
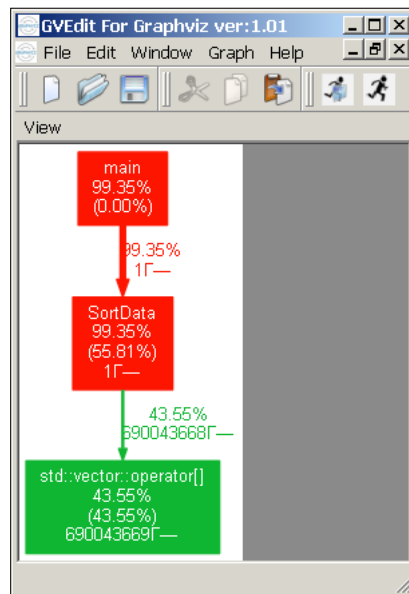
You can see a call-graph in the preceding screenshot. Nodes of this graph are represented as colored edges. These nodes represent called functions of the analyzing program. Each edge contains the following information:

- ✦ **Total time %** is the percentage of the running time spent in this function and all its children
- ✦ **Self time %** (in brackets) is the percentage of the running time spent in this function alone
- ✦ **Total calls** is the total number of times this function was called (including recursive calls)

Moreover, nodes have temperature-like colors according to the total time percent value. Most time expensive functions display as red (hot-spot) edges and the most cheap ones as dark blue.

The most execution time of our example application has been spent in the red colored `SortData` function. Statistic information about the `SortData` function at graph confirms it:

- ✦ Total time is equal to 99.35 percent
- ✦ Self time is equal to 55.81 percent
- ✦ Total calls is equal to 1

We can use the STL `sort` algorithm instead of the bubble one to optimize our example application. Let's change the code of the `SortData` function as follows:

```
void SortData()
{
  sort(gData.begin(), gData.end());
}
```

Now you need to rebuild the application, run the application, and run gprof and the
`gprof2dot.py` script. Open the resulting dot file in the Graphviz application. You will
see the call-graph as shown in the following screenshot:
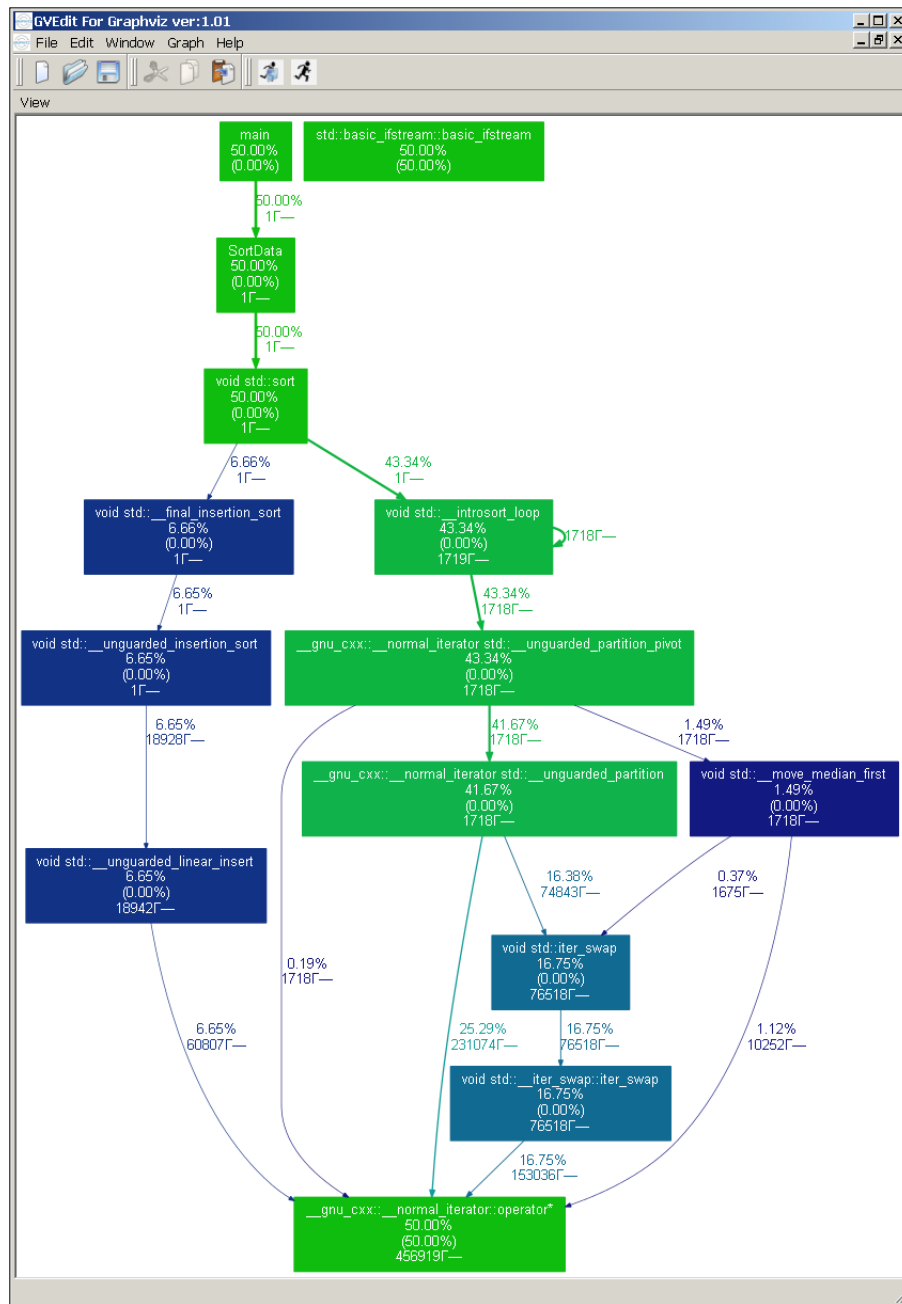
The red edges disappear from the graph. This means that we don't have any explicit bottleneck. All program execution time has been evenly distributed between several functions.

You can remove the profiling extra code from executable files and libraries after profiling with the `strip` command:

```
$ strip sorting.exe
```

Profiling is a significant technique that allows you to find and remove bottlenecks in your applications. This may be very helpful for performance analyzing of the complex systems with many components and libraries. MinGW software allows you to include this technique in your software development process in a simple and fast manner.

6 – Developing with cross-platform libraries

MinGW software allows you to develop applications based on any library compiled with the MinGW C++ compiler. Open source libraries are often supplied in Visual C++ and MinGW compilation variants. Moreover, you can always get the source code of these libraries and build them with your MinGW software. Several well-known open source cross-platform frameworks and toolkits are described as follows:

- ✦ Qt framework
- ✦ Gtk+ widget toolkit
- ✦ wxWidgets framework

The same functionality example application will be developed with each of these libraries. The goal of these examples is to show the first steps to deploy and to start working with these libraries.

Our example application consists of a window with two buttons. A click on the first button leads to the display of a message and a click on the second button leads to the application termination. This functionality is the same as the one described in the *Quick start* section.

The Qt framework

The Qt framework provides not only a cross-platform widget toolkit for GUI development but also contains features for SQL database access, XML parsing, thread management, interaction over a network, and internalization support. The Qt framework has its own container classes such as `Qstring` or `QVector` and a set of well-known algorithms such as sorting, searching, and copying to process data in these containers that allow you to substitute the capabilities of STL and Boost libraries by the Qt ones. This kind of Qt framework self-sufficiency is of great advantage to develop cross-platform applications. All you need to do to import your Qt application to a new platform is building a Qt framework for this one. There are a lot of platforms that are supported by Qt framework developers:

- ✦ Windows
- ✦ Mac OS X

- Linux
- Solaris
- Symbian
- Android (unofficial framework port with the name Ministro)

Let's begin our work with the Qt framework. It is important to note that Qt libraries have been compiled with the specific MinGW software version. Your application must be compiled with the same MinGW version and therefore you must install it. The alternative way is the compilation of Qt libraries sources with your already installed MinGW software, but this variant will not be described in this book. You can find some helpful instructions to do this at the following website:

`http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler`

All other toolkits described here don't need a specific MinGW software version. The following are instructions to install Qt libraries version 4.6.4 and some necessary software to start developing with it:

1. Download the MinGW version of the Qt libraries for Windows from the following official website:

   `http://qt-project.org/downloads`

   Note that this library has been compiled with a specific (equal in our case to 4.4) MinGW software version.

2. Install the downloaded Qt libraries with the setup wizard. Run the downloaded exe file to do it.

3. Download the MinGW software of version 4.4 from the following official download page:

   `http://sourceforge.net/projects/mingw/files/MinGW/Base/gcc/Version4/`
   `Previous%20Release%20gcc-4.4.0/`

   You need the file to be named `gcc-full-4.4.0-mingw32-bin-2.tar.lzma`. This is an archive with the MinGW compilers' executable files and necessary core libraries.

4. Extract the downloaded MinGW software archive to the directory without spaces in the path (for example, `C:\MinGW4.4`).

   You can use the 7-Zip application to extract the LZMA archive type. This application is available at the following official website:

   `http://www.7-zip.org/download.html`

5. Download the GNU Binutils for MinGW software from the following official download page:

   `http://sourceforge.net/projects/mingw/files/MinGW/Base/binutils/`
   `binutils-2.19`

   You need the archive to be named `binutils-2.19-2-mingw32-bin.tar.gz`.

6. Extract the downloaded GNU Binutils archive to the same directory as the MinGW software. This is `C:\MinGW4.4` in our case.

38

7. Download the GNU Make utility from the following web page:

   ```
   http://sourceforge.net/projects/mingw/files/MinGW/Extension/make/
   mingw32-make-3.80-3
   ```

   You need to download the `mingw32-make-3.80.0-3.exe` file.

8. Install GNU Make utility to the MinGW software directory (`C:\MinGW4.4`). Run the downloaded exe file and use the setup wizard to do it.

9. Add the installation MinGW software path to the `PATH` Windows environment variable. Remove the existing paths of other MinGW software installations from there. The `C:\MinGW4.4\bin` path must be added in our example.

Now you have the necessary libraries and specific MinGW software version to start developing an application based on the Qt framework.

Our example application is implemented in the `main.cpp` source file. The following is the `main` function definition:

```cpp
int main(int argc, char* argv[])
{
  QApplication application(argc, argv);

  QMainWindow* window = CreateWindow();

  CreateMsgButton(window);

  CreateQuitButton(window, application);

  return application.exec();
}
```

First of all, the object of the `QApplication` class named `application` is created here. This object is used to manage the application's control flow and main settings. After that, the window of the `QMainWindow` class named `window` is created by the `CreateWindow` function. Two window buttons are created by the `CreateMsgButton` and `CreateQuitButton` functions. The `exec` method of the `application` object is called to enter the main event loop when all user interface objects have been created. Now the application starts processing events such as button pressing.

The following is the `CreateWindow` function that encapsulate's the main application window creation:

```cpp
QMainWindow* CreateWindow()
{
  QMainWindow* window = new QMainWindow(0, Qt::Window);

  window->resize(250, 150);
```

39

```
        window->setWindowTitle("Qt Application");
        window->show();

        return window;
    }
```

The main application window is an object of the `QMainWindow` class named `window`. It is created by a constructor that has two input parameters. The first parameter is of the `QWidget*` type. This is a pointer to the window's parent widget. It equals to zero in our case which means that there is no parent widget. The second parameter is of the `Qt::WindowFlags` type and defines the window style. It equals to `Qt::Window` that matches the standard window appearance with the system frame and title bar.

After the main window creation its size is set with the `resize` method of the `window` object. Then the window title is set with the `setWindowTitle` method of the `window` object. The next action is to make the main window visible by using the `show` method of the `window` object. The function returns the pointer to the created `window` object.

The following code shows the `CreateMsgButton` function that encapsulates the creation of button with message showing action:

```
    void CreateMsgButton(QMainWindow* window)
    {
      QMessageBox* message = new QMessageBox(window);
      message->setText("Message text");

      QPushButton* button = new QPushButton("Message", window);
      button->move(85, 40);
      button->resize(80, 25);
      button->show();
      QObject::connect(button, SIGNAL(released()), message, SLOT(exec()));
    }
```

At first, we need to create a message window to display it when we click on the button. This message window is an object of the `QMessageBox` class named `message`. It is created by a constructor with one input parameter of the `QWidget*` type. This is a pointer to the parent widget. The text displayed in the message window is set by the `setText` method of the `message` object.

Now we need a button that will be placed at the main window. This button is the object of the `QPushButton` class named `button`. The constructor with two input parameters is used here to create this object. The first parameter is a string with the button text of the `QString` class. The second parameter is the parent widget pointer. The position and size of the `button` object are set by the `move` and `resize` methods. After configuring the button we make it visible by using the `show` method.

Now we need to bind the button press event and message window displaying. The `connect` static method of the `QObject` class is used here for this goal. It allows to create a connection between the `button` object's release signal and the `exec` method of the `message` object. The `exec` method causes the displaying of window with message.

The following code shows the `CreateQuitButton` function that encapsulates the creation of the close application button:

```
void CreateQuitButton(QMainWindow* window, QApplication& application)
{
  QPushButton* quit_button = new QPushButton("Quit", window);
  quit_button->move(85, 85);
  quit_button->resize(80, 25);
  quit_button->show();
  QObject::connect(quit_button, SIGNAL(released()), &application,
SLOT(quit()));
}
```

This function is similar the `CreateMsgButton` function. But the message window isn't created here. The close application button is an object of the `QPushButton` class named `quit_button`. The `release` signal of the `quit_button` button is connected to the `quit` method of the `application` object. This method leads to the application closing with successful code.

The full `main.cpp` file is available in the code bundle uploaded on the Packt website.

Now you are ready to build an example application. You need to perform the following instructions:

1. Create a Qt project file with the following command:

    $ qmake -project

    You will get a file with the `pro` extension and name of the current directory.

2. Create `Makefile`, service files, and subdirectories with the following command:

    $ qmake

3. Compile the project with the GNU Make utility:

    $ mingw32-make

The GNU Make utility executable name is `mingw32-make` in the official distribution. This one has been installed with the MinGW software of version 4.4.

You will get the `qt.exe` executable file in the `debug` subdirectory after compilation.

This is the debugging version of our example application. Type the following command to build the `release` version:

$ mingw32-make release

The `qt.exe` executable file will be created in the `release` subdirectory after this compilation.

41

The Gtk+ widget toolkit

Gtk+ is a cross-platform toolkit with many widgets to construct user interfaces. It is important to note that Gtk+ has been written in C language. This toolkit has an object model, but there are no C++ classes and objects. You can use the toolkit in your C++ applications, but it may be helpful to use the gtkmm. The gtkmm is an official C++ interface for Gtk+. The gtkmm interface is not described in this book, but you can get more information about it at the following official website:

```
http://www.gtkmm.org
```

Gtk+ is a good choice if you are looking for a widget toolkit for user interface creation and you don't need any additional features provided by the Qt framework.

The following are the instructions to install the Gtk+ widget toolkit:

1. Download the all-in-one bundle archive with the Gtk+ widget toolkit from the following official website:

   ```
   http://www.gtk.org/download/win32.php
   ```

2. Extract the archive to the directory without spaces in the path (for example, `C:\Gtk+`).

3. Add the installation path of the Gtk+ toolkit to the `PATH` Windows environment variable. The path, `C:\Gtk+\bin`, must be added in our case.

4. Create a new Windows environment variable named `PKG_CONFIG_PATH`. Specify the path to the `pkg-config` utility files as a value of this variable. This is `C:\Gtk+\lib\pkgconfig` in our case.

Now you have the necessary libraries to start developing applications with the Gtk+ widget toolkit. Unlike the Qt framework you don't need the same MinGW software version as Gtk+ libraries have been compiled with. Any already installed MinGW software can be used to compile your applications.

Our example application is implemented in the `main.cpp` source file. The following is the `main` function definition:

```cpp
int main(int argc, char* argv[])
{
  gtk_init(&argc, &argv);

  GtkWidget* window = CreateWindow();

  GtkWidget* box = gtk_vbox_new(FALSE, 0);
  gtk_widget_show(box);

  CreateMsgButton(box);

  CreateQuitButton(box);
```

```
gtk_container_add(GTK_CONTAINER(window), box);
gtk_widget_show(window);
gtk_main();

return 0;
}
```

First of all the `gtk_init` function is called. It initializes everything that we need to operate with Gtk+ toolkit. Four widgets are created in the `main` function after that. There are the main window, vertical box container, and two buttons. The main window is a pointer to the `GtkWidget` Gtk+ structure that is created by the `CreateWindow` function.

The box container is needed to place several widgets in the main window (two buttons in our case). This box container is a pointer to the `GtkWidget` structure that is created by the `gtk_vbox_new` Gtk+ function. This function has two input parameters. The first parameter is of the `gboolean` type that defines whether all children widgets are to be given equal space allotments. The second parameter is of the `gint` type that defines the number of pixels to place between child widgets. The `gtk_widget_show` function is called to make the box container visible.

Buttons to show messages and close applications are created in the `CreateMsgButton` and `CreateQuitButton` functions.

The `gtk_container_add` function is used to put one widget into another. The box container has been put into the main window in our case. After that the `window` widget is made visible with the `gtk_widget_show` function.

The `gtk_main` function leads to run main application loop to process events.

The following is the `CreateWindow` function that creates the main window:

```
GtkWidget* CreateWindow()
{
  GtkWidget* window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title(GTK_WINDOW(window), "Gtk+ Application");
  g_signal_connect(window, "delete-event",
G_CALLBACK(gtk_main_quit), NULL);
  g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit),
NULL);

  return window;
}
```

The main window of the `GTK_WINDOW_TOPLEVEL` type is created here with the `gtk_window_new` function. This window type equals to a regular application window. Then the window title is set with the `gtk_window_set_title` function.

43

Now we must bind the application closing and main window destroy events. This is needed to close the application after the main window closes. The events are named signals in Gtk+ terminology. The `g_signal_connect` macro is used here for signal binding. This macro has four arguments. The first argument is the pointer to the widget that you need to connect to. The second argument is a string with the signal name. The third argument is a callback function that will process this signal. The fourth argument is the pointer to additional signal data.

There are two signals named `delete-event` and `destroy` that match the window closing event. The `gtk_main_quit` function will be called when these signals occur in our case. This function will stop the application event loop and will lead to the application termination.

The following is the `CreateMsgButton` function that creates the message showing button:

```
void CreateMsgButton(GtkWidget* box)
{
  GtkWidget* button = gtk_button_new_with_label("Message");
  gtk_widget_show(button);
  g_signal_connect(G_OBJECT(button), "clicked",
                     G_CALLBACK(ShowMessage), NULL);
  gtk_container_add(GTK_CONTAINER(box), button);
}
```

This function has one input parameter of the pointer to the `GtkWidget` structure type. This is used to pass the pointer of the button container widget.

The button widget is a pointer to the `GtkWidget` structure named `button`. It is created by the `gtk_button_new_with_label` function. This function has only one input parameter with the button label string. After that the button widget is made visible with the `gtk_window_show` function. The `clicked` signal of the button widget is connected to the `ShowMessage` function. The button widget is added to the box container when it has been configured.

The following is the `ShowMessage` function to create and display the message window:

```
void ShowMessage(GtkWidget* widget, gpointer data)
{
  GtkWidget* message = gtk_message_dialog_new((GtkWindow*)data, GTK_
DIALOG_MODAL, GTK_MESSAGE_INFO, GTK_BUTTONS_OK, "Message text");
  gtk_dialog_run(GTK_DIALOG(message));
  gtk_widget_destroy(message);
}
```

The message window is the Gtk+ dialog widget that has been created with the `gtk_message_dialog_new` function. Parent window pointers, dialog flags, message types, dialog buttons, and message text are passed to this function. The `gtk_dialog_run` function is called to display the created dialog. This function will return control when the dialog button has been clicked. The `gtk_widget_destroy` function at the next line will destroy our message window.

The following is the `CreateQuitButton` function that creates the quit button:

```
void CreateQuitButton(GtkWidget* box)
{
  GtkWidget* button = gtk_button_new_with_label("Quit");
  gtk_widget_show(button);
  g_signal_connect(G_OBJECT(button), "clicked",
                     G_CALLBACK(gtk_main_quit), NULL);
  gtk_container_add(GTK_CONTAINER(box), button);
}
```

This is the same as the `CreateMsgButton` function, but the button labels differ. Moreover, the `clicked` signal of this button is bound with the `gtk_main_quit` function. This function will lead to the application termination.

The full `main.cpp` file is available in the code bundle uploaded on the Packt website.

The following is `Makefile` to compile our example application:

```
OBJ=main.o

CXXFLAGS+='pkg-config --cflags gtk+-win32-2.0'
LIBS+='pkg-config --libs gtk+-win32-2.0'

gtk.exe: $(OBJ)
  $(CXX) -o $@ $^ $(LIBS)

clean:
  rm -f *.o *.exe
```

The `pkg-config` utility from the Gtk+ toolkit is used here to get the compiler and linker Gtk+ specific flags. The grave accent mark means that the wrapped string will be performed as a command and the result of the execution will be returned as a string value.

The following `pkg-config` options have been used here:

- ✦ `--cflags`: This prints the preprocessor and compiler flags to compile the application with a specified library
- ✦ `--libs`: This prints the linker flags to link the application with a specified library

`gtk+-win32-2.0` is the name of the package with the Gtk+ libraries that we want to link with.

The results of the `pkg-config` utility execution are assigned to the `CXXFLAGS` and `LIBS` variables.

Now you have all the source files that are needed to compile our example application. Type the `make` command to do this. You will get the `gtk.exe` executable file in the current directory.

45

wxWidgets framework

The wxWidgets framework contains the widget toolkit for user interface development and features for network programming, threading support, image processing, database support, and HTML and XML processing. The wxWidgets have a set of their own containers and algorithms that can be enough to develop applications with this framework resources only.

Implementing own user interface elements is common practice for many widget toolkits and frameworks. This is what makes the Qt framework and Gtk+ widget toolkit. wxWidgets unlike these, uses the native platform's user interface elements through the platform's API. Thanks to this, wxWidgets-based applications look and feel like a native one.

wxWidgets is a good base to develop cross-platform high-quality, native-looking applications for Mac OS X, Windows, Linux, and other Unix family operating systems.

Several wxWidgets framework versions are available from the official website as source code archives. You can download one of these and build it with your currently installed MinGW software.

The following are the instructions to install the wxWidgets widget toolkit:

1. Download the wxMSV version of wxWidgets from the following official website:

   `http://www.wxwidgets.org/downloads`

2. Install the downloaded wxWidgets toolkit with the setup wizard. Run the downloaded exe file to do it. We will assume that the `C:\wxWidgets` target directory has been specified for example.

3. Build the wxWidgets toolkit with your already installed MinGW software. Type the following commands to do it:

   cd C:\wxWidgets\build\msw

   make -f makefile.gcc SHARED=1 UNICODE=1 BUILD=release clean

   make -f makefile.gcc SHARED=1 UNICODE=1 BUILD=release

4. Download the `wx-config` utility from the developer's page:

   `http://code.google.com/p/wx-config-win`

   This utility is used to search the wxWidgets toolkit libraries and header files by GNU Make.

5. Copy the `wx-config` utility to the wxWidgets installation directory (in our case `C:\wxWidgets`).

6. Add the path to the wxWidgets installation directory and the path to the dynamic-linked libraries in the PATH Windows environment variable. The `C:\wxWidgets` and `C:\wxWidgets\lib\gcc_dll` values must be added in our example.

Now you have the necessary libraries to start developing applications with the wxWidgets widget toolkit.

The example wxWidgets application is implemented in the `main.cpp` source file. User classes must be created here unlike the Gtk+ and Qt example application variants. The `MyApp` class is the base application class that is derived from the `wxApp` wxWidgets library standard class.

The following is the `MyApp` class definition:

```
class MyApp : public wxApp
{
public:
  virtual ~MyApp() {}

private:
  virtual bool OnInit();
};
```

The virtual destructor and the `OnInit` virtual method are defined here. The virtual destructor is needed here for correct parent class data deleting. All application functionality is implemented in the `OnInit` method of the `MyApp` class.

The following is the `OnInit` method of the `MyApp` class:

```
bool MyApp::OnInit()
{
  MyDialog* dialog = new MyDialog(NULL, 0, _("wxWidgets
application"));

  wxSizer* sizer = dialog->CreateButtonSizer(wxOK | wxCANCEL);
  sizer->SetDimension(175, 50, 100, 100);

    while ( dialog->ShowModal() == wxID_OK )
    {
      wxMessageBox(_("Message text"),
                   _("Information"),
                   wxOK | wxICON_INFORMATION, dialog);
  }

  dialog->Destroy();
  return true;
}
```

The main application window is created here. This window is the object of the `MyDialog` class with the `dialog` name. The `MyDialog` class is a user defined class derived from the `wxDialog` widget class. The constructor of the `MyDialog` class has three input parameters. These are parent widget pointer, widget identifier, and title bar caption string.

Then two buttons with `OK` and `Cancel` captions are created for the `dialog` object by the `CreateButtonSizer` method. This method has one parameter that defines the standard buttons list to creation. Each of these buttons is represented by the bit flag. The `CreateButtonSizer` method returns the pointer to the object of the `wxSizer` class with the `sizer` name. This object represents the sub-window that contains the buttons. Thanks to the `sizer` object the button's position can be changed with the `SetDimensions` method. This method has four input parameters. The first two parameters are x and y coordinates and the second two parameters are the width and height of the `sizer` subwindow.

The `ShowModal` method of the `dialog` object method will be called in a loop after the widget's initialization. This is the main application loop. It will be interrupted when the main application window has been destroyed.

The information message will be displayed when the button with the `wxID_OK` identifier is clicked on. The `wxMessageBox` function is used to show an informational message. This function has six input parameters. The four of these are used here and the other two have values by default. The first parameter is a string with the message text. The second parameter is a string with message window caption bar text. The third parameter defines the message window style by bit flags. The fourth parameter is a pointer to the parent widget. The last two parameters are the coordinates of the message window.

The main application window will be hidden when the button with the `Cancel` caption will be clicked. The `Destroy` method of the `dialog` object is called here to destroy the main application window.

The following is the `MyDialog` class definition:

```
class MyDialog : public wxDialog
{
public:
  MyDialog(wxWindow* parent, wxWindowID id,
              const wxString& title) : wxDialog(parent, id, title) {}
  virtual ~MyDialog() {}
};
```

The `MyDialog` class is derived from the `wxDialog` widget. The class constructor and destructor are defined here. The `MyDialog` class constructor just passes input parameters to the parent class constructor.

You must specify this macro in the `main.cpp` source file to create the application instance and start the program:

```
IMPLEMENT_APP(MyApp)
```

The full `main.cpp` file is available in the code bundle uploaded on the Packt site.

The following is `Makefile` to compile our example application:

```
OBJ=main.o

CXXFLAGS+='wx-config --cxxflags --wxcfg=gcc_dll/mswu'
LIBS+='wx-config --libs --wxcfg=gcc_dll/mswu'

wxwidgets.exe: $(OBJ)
  $(CXX) -o $@ $^ $(LIBS)

clean:
  rm -f *.o *.exe
```

The `wx-config` utility is used here. This is the `wxWidgets` framework's alternative of the `pkg-config` utility of Gtk+ toolkit. The `wx-config` utility is used here to get the compiler and linker wxWidgets specific flags.

The next `wx-config` utility options have been used here:

✦   `--cxxflags`: This prints the preprocessor and compiler flags to compile the application with the wxWidget library

✦   `--libs`: This prints linker flags to link the application with the wxWidget library

✦   `--wxcfg`: This specifies a relative path to the `build.cfg` configuration file

Now you are ready to compile our example application. Type the `make` command to do it. You will get the `wxwidgets.exe` executable file in the current directory after compilation.

It is important to note that you can debug and profile your applications based on any of the described toolkit and frameworks with the MinGW software tools (GDB debugger and gprof profiler).

7 – Integrating with IDE

MinGW software can be integrated with a lot of well-known free IDE systems. Integration with the following systems will be described here:

✦   Code::Blocks

✦   Qt Creator

✦   Eclipse

Integration in this case means the ability to edit MinGW-based project source code, building this project, and debugging it from the IDE. This integration provides a comfortable interface to interact with the most commonly used MinGW instruments.

Code::Blocks

Code::Blocks is an open source cross-platform IDE for developing C and C++ applications. Code::Blocks has an open architecture. Thanks to this, IDE capabilities can be expanded with plugins with significant facilitates software development process.

Code::Blocks supports a lot of C and C++ compilers and several debuggers. This IDE is a good alternative for the Visual C++ one to develop C++ applications.
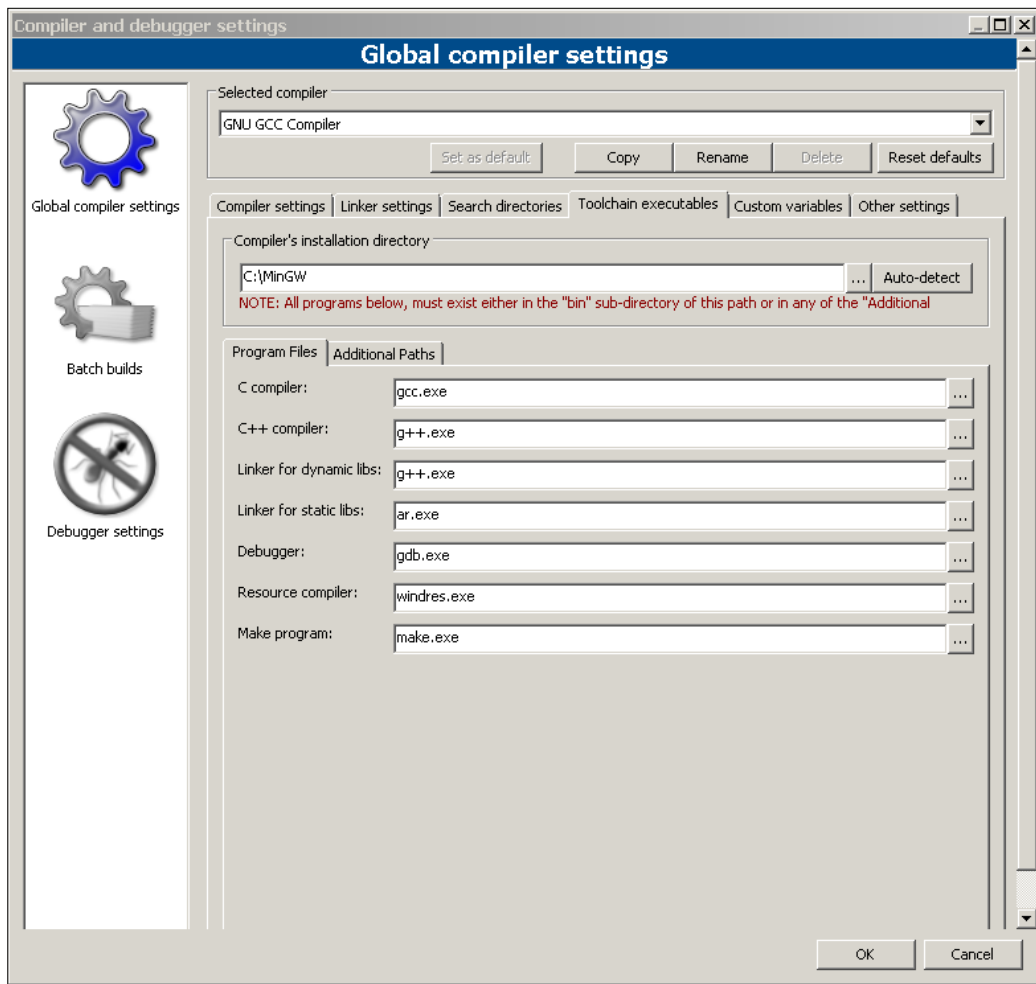
The following are instructions to install and configure the Code::Blocks IDE:

1. Download the Code::Blocks IDE distribution with the already integrated MinGW software from the official website:

   ```
   http://www.codeblocks.org/downloads/binaries
   ```

2. Install the downloaded Code::Blocks distribution. Run the downloaded exe file to do it.

3. Select additional IDE components in **Choose Components** dialog during installation process.

   The default installable components will be enough to start developing the C++ application. But you can choose additional languages supporting GNU utilities and IDE plugins.

4. Start installed Code::Blocks system and select **GNU GCC Compiler** to use in the compilers auto-detection dialog.

Now you are ready to use the Code::Blocks IDE system with integrated MinGW software.

You can install the Code::Blocks IDE without integrated MinGW software if you have already installed it. The following are instructions to set up Code::Blocks working with your already installed MinGW software:

1. Run Code::Blocks IDE.

2. Select **Settings | Compiler and Debugger…** in the main menu. You will see the compiler setting dialog:

3. Switch to the Toolchain executables tab. Specify the MinGW software installation directory and its utility executables file names.

Perform the following actions to create a new MinGW based project in Code::Blocks IDE:

1. Run Code::Blocks IDE. You will see the main system window:
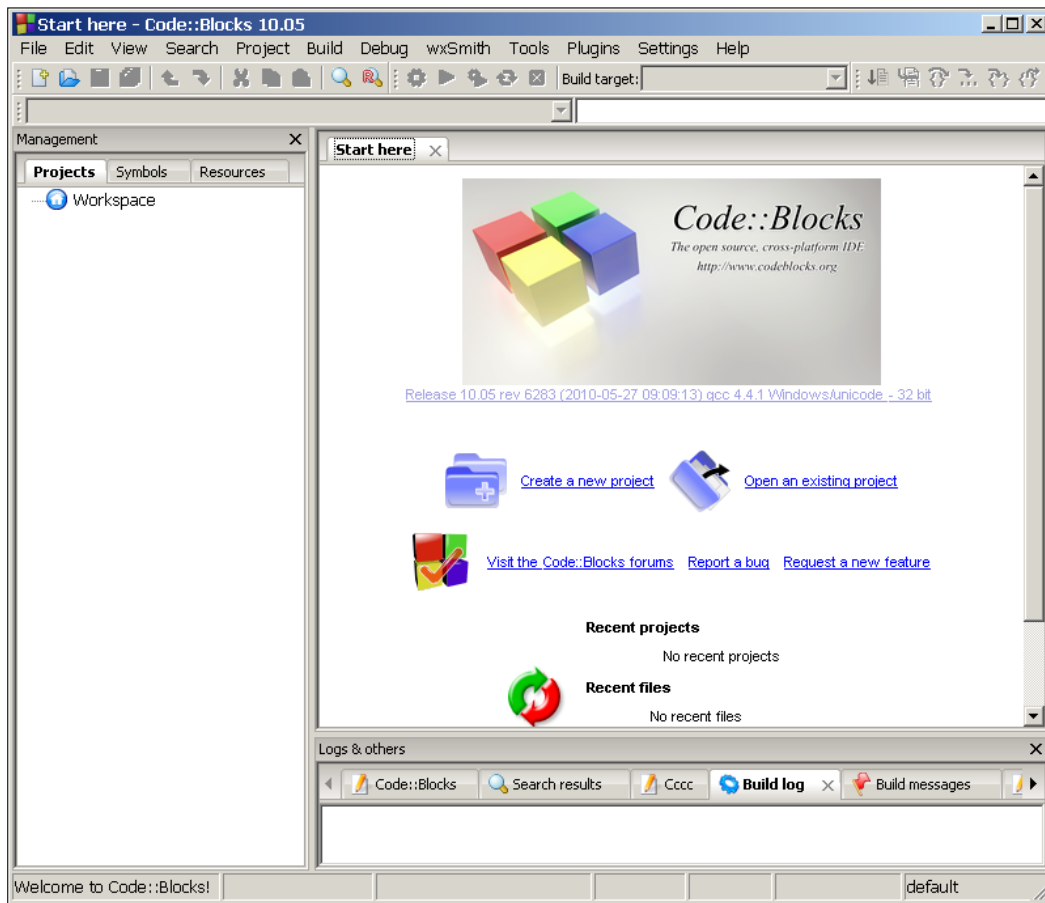
You can see the **Start here** tab content in the middle of the window. The main IDE menu is placed at the top of the window. The messages panel is placed at the bottom of the window. The build error messages, build log, debugger messages, and search results will be displayed here.

2. Click on the **create a new** project icon on the **Start here** tab or choose **File | New | Project...** in the main menu.

3. We will create a template Windows application in this example. Choose the **Win32GUI project** icon in the **new from template** dialog to do it.

4. Choose a frame-based type of project in the next dialog.

5. Specify the project name and path to store its sources.

6. Select the compiler to build the application in the last dialog. It will be equal to the GNU GCC compiler by default. Leave it unchanged.

You will get template source files of the MinGW-based Windows API application. This application just shows an empty window at launch.

You can build our example application from the **Code::Blocks IDE** interface. Choose **Build | Build** in the main menu or press *Ctrl + F9* to do it. The build log and build messages will be displayed in the messages panel after that. Note what debug variant of build has been performed. You can change it to a release variant by switching the **Build target** combobox in the main menu. The application executable files will be created at the `bin` subdirectory of the our project directory.

Now the application is ready to be run. Choose **Build | Run** in the main menu or press *Ctrl + F10* to run the application. You will see the application window and console window , where the standard output stream messages will be displyed.

Code::Blocks IDE allows you to debug applications with the GDB debugger. First of all you must set breakpoints to stop program execution in specified places. Press the *F5* key to set a breakpoint at the current line in the source file. Press *F8* or choose **Debug | Start** from the main menu to start debugging. After that program execution stops at the specified line. Now you can get information about variables, call stack, CPU registers, and threads, or continue execution. All this functionality is available from the **Debug** submenu. You can continue the application debugging with the next line (*F7*) and step into (*Shift + F7*) commands.

Qt Creator

Qt Creator is a cross-platform open source IDE, which is part of the Qt software development kit. It includes a source code editor, visual debugger, and forms designer. It supports both MinGW and Visual C++ software.

Qt Creator is the best choice to develop Qt framework-based C++ applications. But other frameworks, toolkits, and programming languages are supported poorly. This obstacle must be considered when you choose this IDE for your application's development.

MinGW software and Qt libraries are not present in the Qt Creator IDE distribution and therefore you must install these separately.

The following are instructions to install and configure Qt Creator IDE:

1. Download Qt Creator IDE distribution from the official website:

    ```
    http://qt-project.org/downloads
    ```

2. Install Qt Creator with the help of the setup wizard. Just run the downloaded exe file to do it. Run the installed Qt Creator IDE. You will see the main system window:
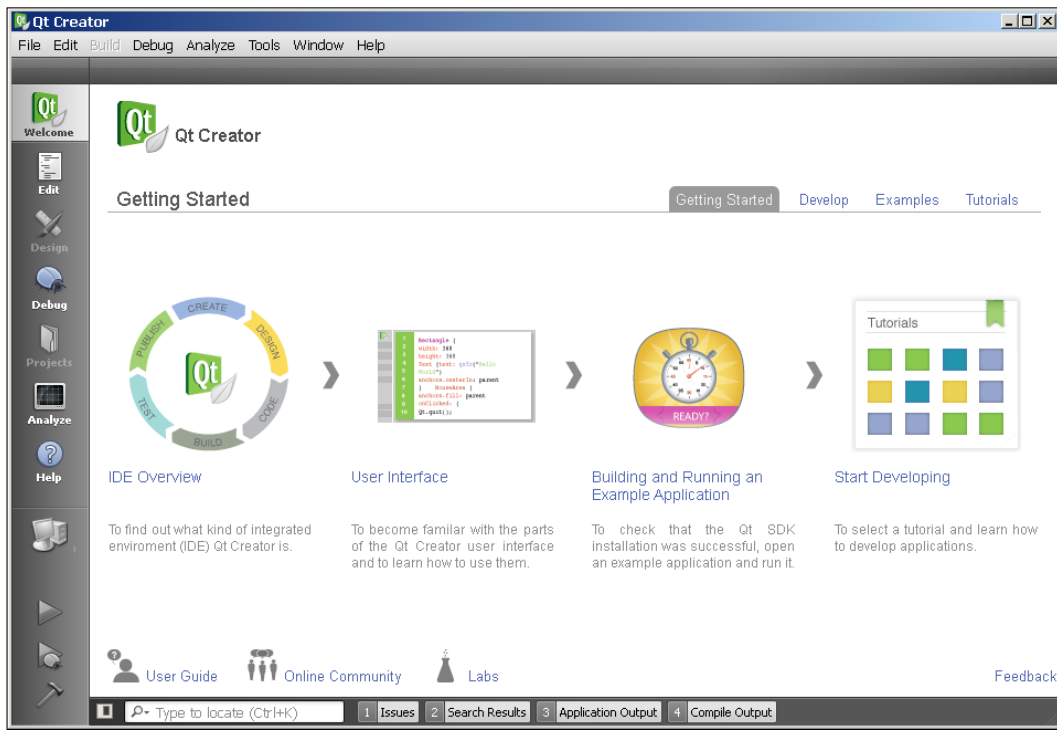

The **Welcome** tab content is placed in the middle of the window. You can see the main system menu at the top of the window. The control panel is placed at the left-hand side of the window. The most commonly used commands are available in this panel. Moreover the icons of different Qt Creator IDE modes (Editor, Designer and so on) are placed here. The messages' output panels are available at the bottom of window.

3. Select **Tools | Options…** in the main menu. You will see the **Options** dialog.

4. Switch to the **Build & Run** icon in the left-hand side of the dialog panel.

5. Switch to the **Qt Versions** tab at the top of the dialog window. Now you see the following screenshot:
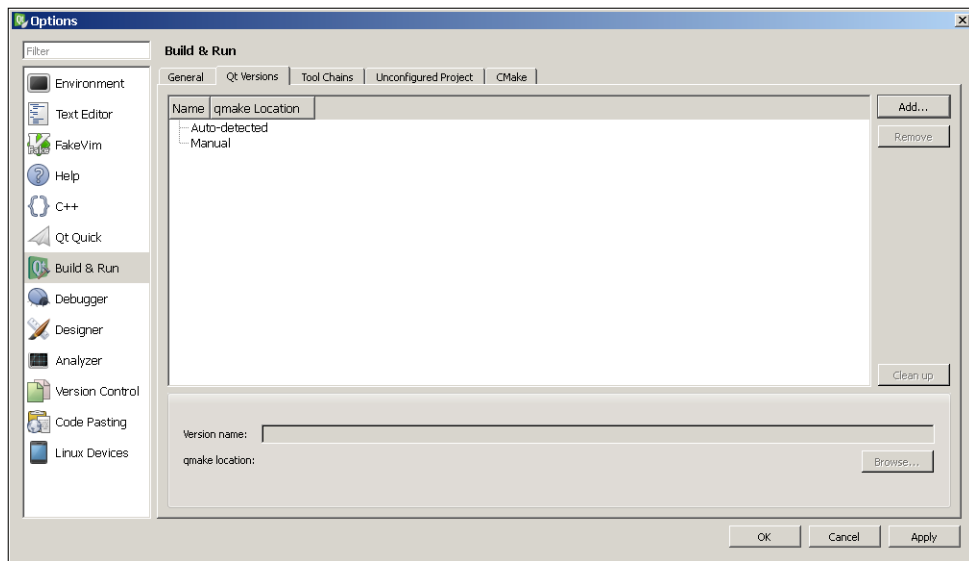
6. Click on the **Add** button and specify the path to the `qmake.exe` file. This is `C:\Qt\4.6.4\bin\qmake.exe` for the default Qt libraries installation path.

7. Switch to the **Tool Chains** tab at the top of the dialog window. You will see the following screenshot:
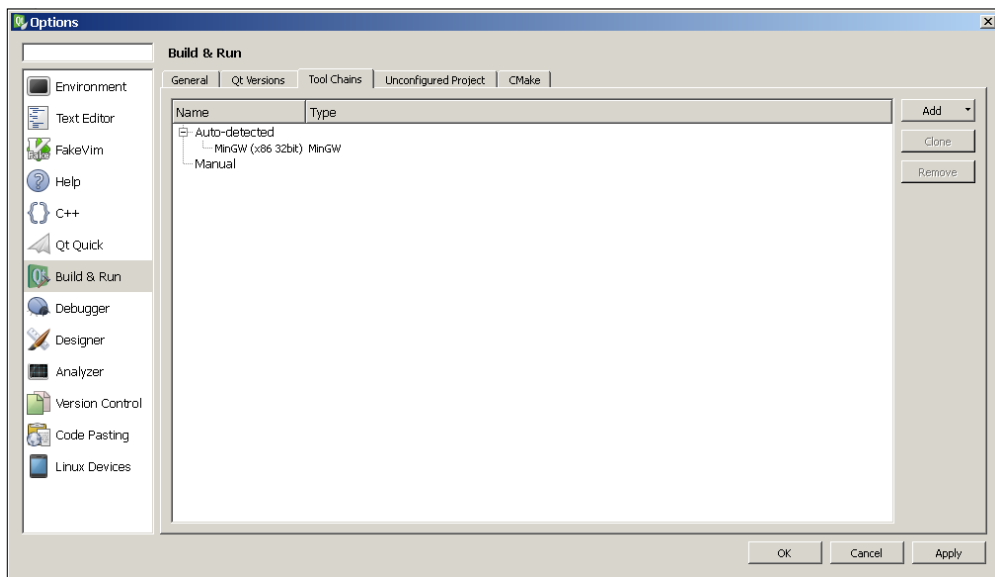

Qt Creator can find already installed MinGW software automatically, but I recommend you manually add it for detailed configuration.

8. Click on the **Add** button and choose the MinGW variant in the pop-up list.

9. Click on the **Browse...** button near the **Compiler path** field and specify the path to the `g++.exe` file. This is equal to `C:\MinGW4.4\bin\g++.exe` if the MinGW software has been installed to the `C:\MinGW4.4` directory.

10. Click on the **Browse...** button near the **Debugger** input field and specify the path to the `gdb.exe` file. You can install the GDB debugger separately if your MinGW software distribution doesn't already contain it. The debugger path equals to `C:\MinGW4.4\bin\gdb.exe` for our example.

11. Click on the **Apply** and then **OK** button to complete the configuration.

Now you are ready to use the Qt Creator IDE system integrated with MinGW software and Qt libraries. Note that your MinGW software version must be the same as the MinGW software that has been used to build your installed Qt libraries.

The following are the instructions to create example Qt-based application with Qt Creator IDE:

1. Select **File** | **New File** or **Project** from the main menu. You will see the template application choosing dialog.

2. Select **Applications** and **Qt Gui Application** variant in the project type selecting fields:
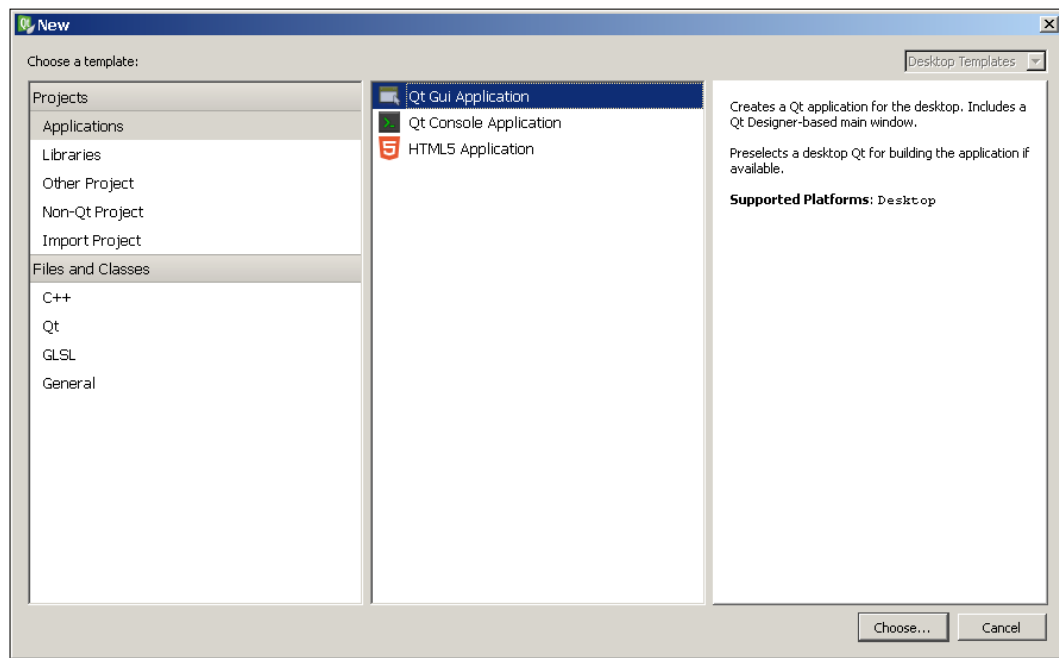
3. Click on the **Choose...** button.

4. Specify the project name (for example, `qt`) and location to store the source files in the **Location** tab of the **Qt Gui Application** dialog. Click on the **Next** button.

5. Choose **Desktop** target in the **Targets** tab of the dialog and click on **Next**.

6. You can change the default source files and class names in the **Details** tab. I suggest you leave these unchanged for our example application. Just click on the **Next** button.

7. The subproject and version control can be added in the **Summary** dialog tab. Switch version control to **None** and click on the **Finish** button.

You will get a template of the Qt-based application with the source files located in a specified location.

Click on the **Build Project** icon at the left-hand side of the control panel or press *Ctrl + B* to build our example application. The `qt.exe` executable file will be created in the `debug` subdirectory of the project directory. You can select the debug or release build variant in the control panel's project configuration submenu (this is placed under the **Run** icon).

Click on the **Run** icon to launch our example application. You will see the empty application window with the status bar and toolbar.

Qt Creator IDE allows you to debug your applications with the GDB debugger. This is standard procedure to do it; you must specify breakpoints at source file lines and then start debugging. Use the *F9* key to set a breakpoint and the *F5* key to start debugging.

The variables' current values, call-stack, and watchpoints configuration are available in debugging mode. You can find all this information in the additional panels of Qt Creator main window. Use **Step Over** (*F10*) and **Step Into** (*F11*) **Debug** submenu items to continue application execution.

Eclipse

Eclipse is a cross-platform open source IDE with multi-language support. There are a huge amount of plugins that have been developed for this IDE. Eclipse supports many compilers, interpreters, debuggers, version control systems, unit testing frameworks, and so on. You can use Eclipse for developing applications in any popular programming language and framework.

Eclipse is an excellent IDE for comfortable application developing, but it has been developed in Java and has a massive architecture. Therefore there are high demands on the performance of your computer.

This is not the Eclipse IDE system distribution that contains the MinGW software. You must install it separately to integrate with Eclipse.

The following are instructions to install and configure the Eclipse IDE system:

1. Download **Java Runtime Environment** (**JRE**) from the official website:

   `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

2. Install it with the setup wizard. Just run the downloaded exe file to do it.

3. Add the path of the JRE executable files to the `PATH` Windows environment variable (for example, `C:\Program Files\Java\jre7\bin`).

4. Download the Eclipse IDE for C/C++ Developers version archive from the official website:

   `http://www.eclipse.org/downloads`

   Note what you need the 32-bit Eclipse IDE system version for 32-bit JRE and 64-bit one for 64-bit JRE.

5. Unpack the Eclipse archive to any directory (for example, `C:\Program Files`)

Now you are ready to use the Eclipse IDE system. The MinGW software utilities will be integrated automatically thanks to Windows environment variables. Therefore the MinGW `bin` subdirectory with executable files must be specified there.

The following are instructions to create an example application in Eclipse IDE:

1. Run Eclipse IDE. You will see the **Welcome** screen.

2. Select **File** | **New** | **C++ Project** from the main menu. You will see the project configuration dialog as shown in the following screenshot:
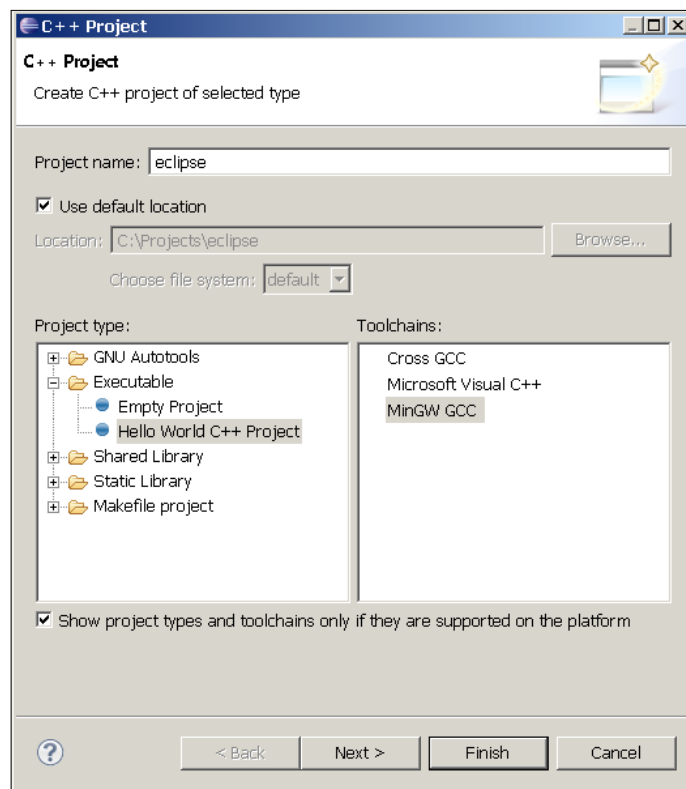
3. Select the **Hello world C++ Project** item in the **Project type** selection field. Select the **MinGW GCC** item in the **Toolchains** selection field. Click on the **Next** button. You will see the **Basic Settings** dialog. The author, copyright notice, and source files directory can be specified here.

4. Click on the **Next** button. This is the **Select Configurations** dialog. Debug, release, or both build configurations availability can be selected here.

5. Click on the **Finish** button.

After that you get a template C++ console project with the source files placed at the `src` subdirectory of the project directory. Now you see the Eclipse IDE window as shown in the following screenshot:
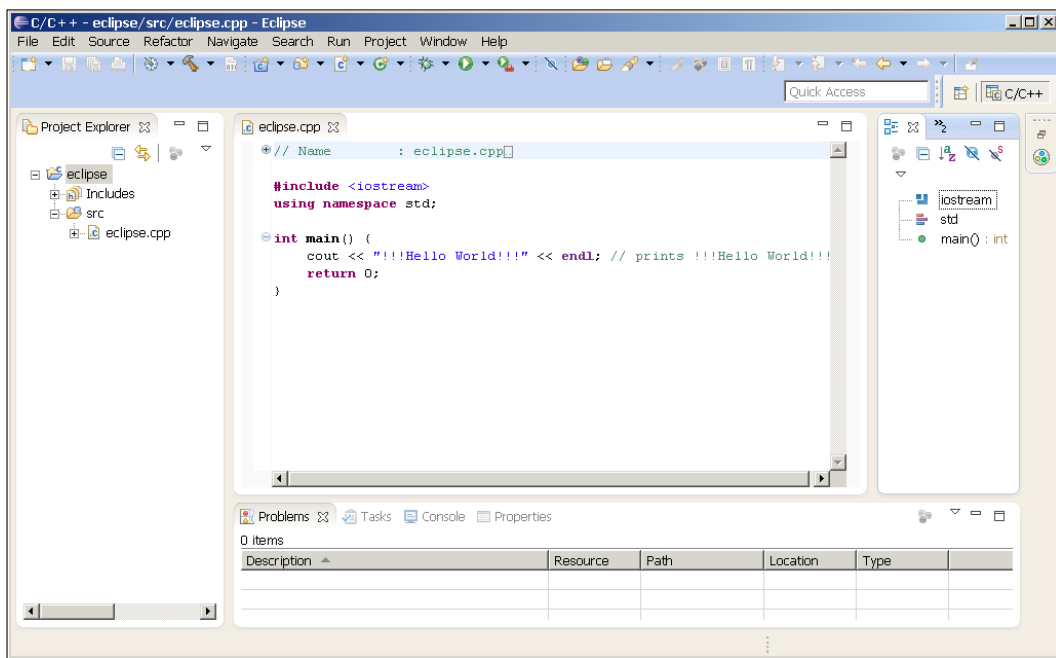
The **Project Explorer** panel is placed at the left-hand side of the window. You can find all the project source files here. The source file editor is placed in the middle part of the window. You can find the messages' output panel at the bottom of the window.

Select **Project | Build Project** from the main menu to build our example application. After that the executable file will be created in the `Debug` subdirectory of the project directory. You can switch to the release configuration to build the project. Click on the build icon (hammer icon) at the main menu and select **Release** configuration to build. The `Release` subdirectory will be created with the compiled executable file.

Select **Run** | **Run** from the main menu or press *Ctrl + F11* to launch our example application. You will see the result of the application execution in the console output panel at the bottom of the Eclipse window.

You can debug applications with the Eclipse IDE system. Press *Ctrl + Shift + B* key to set the breakpoint at the current line of a source file. After setting the breakpoints select **Run** | **Debug** from the main menu or press the *F11* key to start debugging. You will see panels with call-stack, variable values, CPU registers values, breakpoints, and loaded modules in the debugging mode. Use the **Step Over** (*F6*) and **Step Into** (*F5*) **Run** submenu items to continue application execution.

People and places you should get to know

There is a lot of information about MinGW software available on the Internet. You will get to know some helpful sources from this section.

MinGW official sites

✦ **Homepage**: `http://www.mingw.org`

✦ **Manuals**: `http://www.mingw.org/wiki/HOWTO`

✦ **Wiki**: `http://www.mingw.org/wiki`

✦ **Source code and binary files**: `http://sourceforge.net/projects/mingw/files/MinGW`

✦ **Available mailing lists**: `http://www.mingw.org/lists.shtml`

MinGW-w64 official sites

✦ **Homepage**: `http://mingw-w64.sourceforge.net`

✦ **Wiki**: `http://sourceforge.net/apps/trac/mingw-w64`

✦ **Source code and binary files**: `http://sourceforge.net/projects/mingw-w64/files`

✦ **Discussion forum**: `http://sourceforge.net/projects/mingw-w64/forums/forum/723797`

✦ **Public support mailing list**: `https://lists.sourceforge.net/lists/listinfo/mingw-w64-public`

✦ **Support IRC channel**: `irc://irc.oftc.net/#mingw-w64`

GNU Compiler Collection official sites

You can find a lot of useful information about MinGW utilities on the GCC official site:

✦ **Homepage**: `http://gcc.gnu.org`

✦ **Manual and documentation**: `http://gcc.gnu.org/onlinedocs`

✦ **Wiki**: `http://gcc.gnu.org/wiki`

✦ **Available mailing lists**: `http://gcc.gnu.org/lists.html`

GNU Debugger official sites

All the information that you need to debug an application with GNU Debugger can be found at the official site:

- **Homepage**: `http://www.gnu.org/software/gdb`
- **Manual and documentation**: `http://www.gnu.org/software/gdb/documentation`
- **Wiki**: `http://sourceware.org/gdb/wiki`
- **Available mailing lists**: `http://www.gnu.org/software/gdb/mailing-lists`

GNU Make official sites

- **Homepage**: `http://www.gnu.org/software/make`
- **Manual and documentation**: `http://www.gnu.org/software/make/manual/html_node/index.html`
- **Mailing lists**: `<bug-make@gnu.org>; <help-make@gnu.org>`

Articles and tutorials

Here you can find several useful articles and tutorials for MinGW software usage:

- `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`: This is a detailed article about gprof profiler usage
- `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor`: This is quite simple and demonstrative tutorial on GNU Make utility
- `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`: You will find detailed information about MinGW C++ compiler optimization capabilities
- `http://gcc.gnu.org/onlinedocs/gcc/Standards.html`: You can find information about compiler's supporting standards

Community

- **Largest free and open source-focused IRC network**: `http://freenode.net`
- **Unofficial distributions**: `http://nuwen.net/mingw.html`; `http://tdm-gcc.tdragon.net`
- **Several MinGW developers sites**: `http://www.willus.com/mingw/colinp`; `http://www.megacz.com`; `http://rmathew.com`

Twitter

- **Packt Publishing**: `https://twitter.com/packtopensource`

[PACKT] Thank you for buying
PUBLISHING

Thank you for buying
Instant MinGW Starter

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
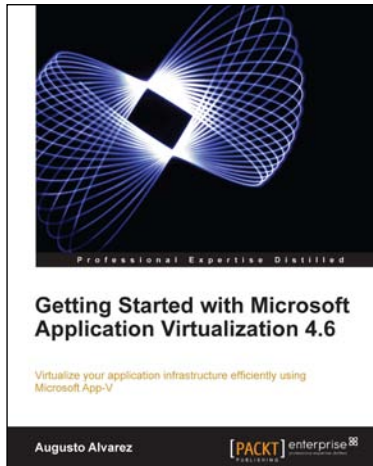
Getting Started with Microsoft Application Virtualization 4.6

ISBN: 978-1-84968-126-1   Paperback: 308 pages

Virtualize your application infrastructure efficiently using Microsoft App-V

1. Publish, deploy, and manage your virtual applications with App-V

2. Understand how Microsoft App-V can fit into your company.

3. Guidelines for planning and designing an App-V environment.

4. Step-by-step explanations to plan and implement the virtualization of your application infrastructure
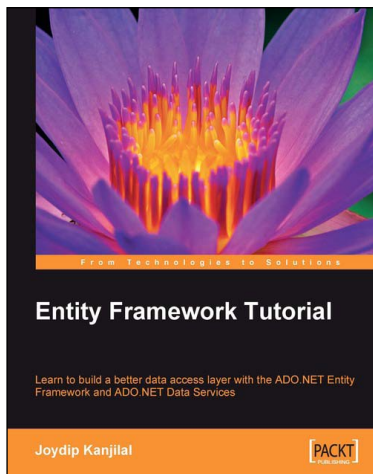
Entity Framework Tutorial

ISBN: 978-1-84719-522-7   Paperback: 228 pages

Learn to build a better data access layer with the ADO. NET Entity Framework and ADO.NET Data Services

1. Clear and concise guide to the ADO.NET Entity Framework with plentiful code examples

2. Create Entity Data Models from your database and use them in your applications

3. Learn about the Entity Client data provider and create statements in Entity SQL

4. Learn about ADO.NET Data Services and how they work with the Entity Framework

Please check **www.PacktPub.com** for information on our titles
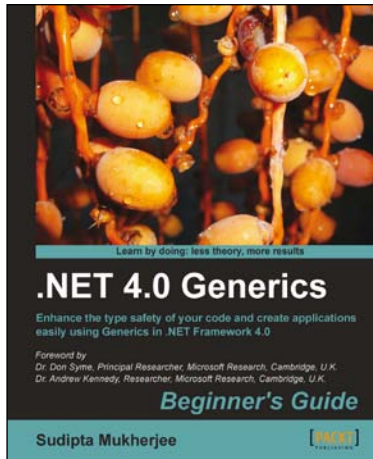
.NET 4.0 Generics Beginner's Guide

ISBN: 978-1-84969-078-2   Paperback: 396 pages

Enhance the type safety of your code and create applications easily using Generics in .NET Framework 4.0

1. Learn how to use Generics' methods and generic collections to solve complicated problems.

2. Develop real-world applications using Generics

3. Know the importance of each generic collection and Generic class and use them as per your requirements

4. Benchmark the performance of all Generic collections
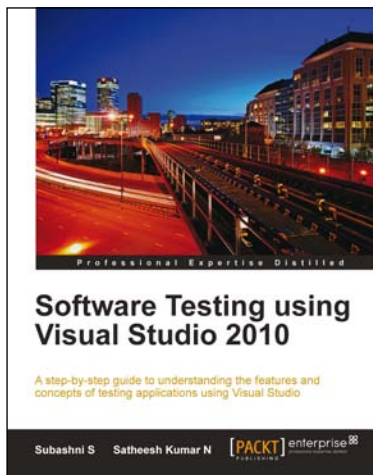
Software Testing using Visual Studio 2010

ISBN: 978-1-84968-140-7   Paperback: 400 pages

A step by step guide to understand the features and concepts of testing applications usng Visual Studio.

1. Master all the new tools and techniques in Visual Studio 2010 and the Team Foundation Server for testing applications

2. Customize reports with Team foundation server.

3. Get to grips with the new Test Manager tool for maintaining Test cases

4. Take full advantage of new Visual Studio features for testing an application's User Interface

Please check **www.PacktPub.com** for information on our titles