CHAPTER 3

Compilers and Assemblers

All development systems are essentially a combination of many tools. Compilers, assemblers and debuggers are integral parts of these development tools. Fortunately Linux has a large number of tools available for software developers. These tools have a very rich set of features that we shall discuss in this and coming chapters. The basic reason for including this chapter is to enable a reader to use compilers and assembler in a development environment in a productive way. However this chapter is not a tutorial on any language or programming technique. Examples of code listings presented in this chapter are very simple and are intended to demonstrate the function of development tools.

GNU C compiler, most commonly known as GCC, is the most important part of this discussion. This compiler can be used to compile programs written in the following languages:

- ANSI C
- Objective C
- C++
- Java
- Fortran
- Pascal, after converting Pascal code to C

GCC recently added the capability to generate object code from Java source code. Work has also been done on Ada support.

GNU assembler is often needed in projects where you want to have tight control over a particular part of code and want to write it in assembly language. Most of the time people also write boot code for embedded systems in assembly language. The compiler uses assembler during the compilation process to generate object code.

In addition to the most common compilers, we shall also discuss the use of some less common languages in the Linux environment. These languages include the following:

- Oberon
- Smalltalk
- Ada

The compilation process is completed in different stages. These stages are:

- Preprocessing
- Compiling to generate assembly language code
- Assembling to generate object code
- Linking to generate executable code

The GNU compiler takes help from many programs during this process and can be used to generate intermediate files. For example, the compiler can generate Assembly language code from a C source code file.

This chapter provides information about the GNU set of compilers and how to build a software development environment using GNU tools. During this process you will also learn the building process for the GNU C library, most commonly known as glibc.

3.1 Introduction to GNU C and C++ Compilers

The GNU compiler, most commonly known as GCC, is not a single program. It is a large set of programs, libraries and other utilities that work together to build source code into executable form. A user, however, mostly interacts with gcc command. The gcc program acts as sort of a front end for compilers and other utilities. This section is an introduction to the compiler and its capabilities.

3.1.1 Languages Supported by GCC

GCC is a family of compilers, also known as the GNU Compiler Collection. GCC supports multiple languages, including:

- C
- C++
- Fortran
- Objective C
- Java

Front-end preprocessors for many other languages also exist for GCC. For example, you can use a Pascal language front end with GCC.

C is the primary language and GCC supports ANSI/ISO standards put forward in 1989-90 and 1995-96. These standards are also sometimes called C90 and C95, respectively. Another revision of the ANSI/ISO C language standard was made in 1999 (known as C99), and GCC does not yet fully support this revision. You can find out the current status of support for this revision at http://gcc.gnu.org/gcc-3.0/c99status.html.

Support for these variations in standards can be selected using command line switches when compiling programs. For example, to compile programs according to original 1989-90 standards, you can use one of the following command line switches:

```
-ansi
-std=c89
-std=iso9899:1990
```

The GCC compiler supports Objective C source code. Information on Objective C can be found in different books and online resources. However there is no standard for Objective C yet. Later in this chapter you will learn how to compile a simple program written in Objective C.

GCC supports GNU Fortran language and we shall discuss later in this chapter how to compile programs written in Fortran.

Support for Java is a new addition to GCC and we shall see how GCC handles Java code with the GCJ compiler. This compiler, which is a part of the GCC family of compilers, can create an executable object from Java source code files.

Standard libraries are not part of the GCC compiler. You have to separately install libraries. The most common library used with the compiler is the GNU C library and we shall see how to get and install this library on Linux systems. The GNU C library (also known as glibc) provides support for:

- ANSI/ISO C
- POSIX
- BSD
- System V
- X/Open

For more information on languages supported by GCC, please see http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_2.html.

3.1.2 New Features in GCC 3.x

There are many new features in GNU compiler starting with version 3.0. Some of these are discussed below briefly. In addition to that, version 3.0.x also includes many fixes for bugs present in 2.x versions.

3.1.2.1 Optimization Improvements

A major improvement in this version is new optimization features. Some of these features greatly improve compile and run-time performance. Important features are listed below.

- Basic block reordering pass is now optimized. In some cases, it improves performance by eliminating a JMP instruction in assembly code. An explanation can be found at http://www.gnu.org/software/gcc/news/reorder.html.
- A new register-renaming pass is included.
- Static Single Assignment or SSA is introduced in GCC. A dead code elimination pass is included in new versions of GCC to improve optimization.
- A global Null pointer elimination test elimination pass is included. Information about this enhancement can be found at http://www.gnu.org/software/gcc/news/null.html.
- New optimizations for `stdio`.h, `string.h` and old BSD functions are added.

3.1.2.2 Support of new Languages

Support of Java is now added through GCJ compiler. It includes a run-time library having non-GUI Java classes.

3.1.2.3 Language Improvements

There are some improvements in the support of different languages and these are listed below.

- New C++ support library is added. Details are available at http://www.gnu.org/software/gcc/libstdc++/.
- A new inliner is added that significantly reduces both memory utilization as well as compile time. Details are present at http://www.gnu.org/software/gcc/news/inlining.html.
- The C preprocessor is rewritten with many improvements. Details are available at http://www.gnu.org/software/gcc/news/dependencies.html.
- More warning messages and a new warning option are added.
- There are many improvements in Fortran language support area and these are listed on http://gcc.gnu.org/onlinedocs/g77_news.html.
- There are other improvements that are not listed here. Go to the GCC home page at http://gcc.gnu.org/ for a complete list.

3.1.2.4 Addition of New Targets

Support for the following platforms is added.

- HP-UX 11
- IA-64
- MCore 210
- MCore 340
- Xscale
- Atmel AVR Micro- controller
- D30V from Mitsubishi
- AM33 from Matsushita
- FR30 from Fujitsu
- 68HC11
- 68HC12
- picoJava from Sun Microsystems

3.1.2.5 Target Specific Improvements

Some of the improvements are as follows:

- New back end is added for x86 targets
- New back end for Arm and Strongarm
- Improvements in PowerPC code generation

3.1.2.6 Improved Documentation

Significant improvements in this area are:

- Many manuals are re-written.
- There are some improvements to man pages.

3.1.2.7 Additional GCC Version 3.0.1 Improvements

This version includes some bug fixes in version 3.0. It also addresses some bugs for exception handling. A port to IBM S/390 is added and some improvements in cross-compiling are made.

3.1.2.8 Additional GCC Version 3.0.2 Improvements

This version mostly contains some bug fixes.

3.1.2.9 GCC Version 3.0.3 Improvements

This version contains some more bug fixes as well as improvement in generation of debugging information.

At the time of revision of this book, GCC 3.1 is released and it has all of these features, among others.

3.2 Installing GNU Compiler

In most cases, GCC comes with all Linux distributions. However you can download the latest version, and build and install it according to your requirements. You may also need to build your compiler if you are building a cross-compiling development system. The best way to build a new version of the compiler is to have some pre-installed and pre-configured Linux distribution on your system that will be used to build the new compiler. For the purpose of writing this book, we have used Red Hat 7.1 but the process is the same on any distribution of Linux.

The installation process is done in multiple steps. After downloading, you have to untar the source code and build it in a directory. This directory should be separate from the source code directory tree. The building process includes configuration and compiling stages. Once you have successfully created the new compiler, you can install it in a directory of your choice. It is advised to keep this installation directory separate from the location where the original compiler is installed.

3.2.1 Downloading

You can download the latest version of GCC from ftp://ftp.gnu.org/gnu/gcc/. I downloaded GCC 3.0.4 and it is about 17.5 MB. You can also find a mirror site near you to get GCC. A list of mirror sites is available on http://www.gnu.org/order/ftp.html.

3.2.2 Building and Installing GCC

The GCC installation process can be divided into four steps for simplicity and understanding.

1. Download and extract
2. Configure
3. Build
4. Install

3.2.2.1 Download and Extract

First create a directory where you will unpack the source code. Use the `tar` command to unpack the code. For our purpose, I have created a directory `/gcc3` to compile and build the GCC compiler. The untar process looks like the following and it creates a directory `gcc-3.0.4` under `/gcc3` directory.

```
[root@laptop /gcc3]# tar zxvf gcc-3.0.4.tar.gz
gcc-3.0.4/
gcc-3.0.4/INSTALL/
gcc-3.0.4/INSTALL/index.html
gcc-3.0.4/INSTALL/README
gcc-3.0.4/INSTALL/specific.html
gcc-3.0.4/INSTALL/download.html
gcc-3.0.4/INSTALL/configure.html
```

```
gcc-3.0.4/INSTALL/build.html
gcc-3.0.4/INSTALL/test.html
gcc-3.0.4/INSTALL/finalinstall.html
gcc-3.0.4/INSTALL/binaries.html
gcc-3.0.4/INSTALL/gfdl.html
gcc-3.0.4/.cvsignore
gcc-3.0.4/COPYING
gcc-3.0.4/COPYING.LIB
gcc-3.0.4/ChangeLog
gcc-3.0.4/MAINTAINERS
gcc-3.0.4/Makefile.in
gcc-3.0.4/README
```

This is a partial output of the command. Most of the output is truncated to save space.

3.2.2.2 Running `configure` Script

After uncompressing the source code, the first thing is to run the `configure` script. This script is found in `/gcc3/gcc-3.0.4` directory. You can specify three major things when you run the `configure` script.

1. Build machine. This is the machine where you compile and build the compiler.
2. Host machine. This is the machine where compiler will be installed. This is usually the same as the build machine.
3. Target machine. This is the machine for which the newly built compiler will generate executable code. If you are building a native compiler, this is the same machine as the host machine. If you are building a cross-compiler, this machine will be different than the host machine.

Each of these machines names are used in the following format:

CPUname-CompanyName-SystemName

For example, if you want to build a compiler that will run on a sparc processor made by Sun Microsystems and on SunOS 4.1 operating system, the command line for `configure` script will be as follows:

```
./configure –host=sparc-sun-sunos4.1
```

Please see a list of supported systems at http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc_4.html .

For a native compiler (a compiler that runs on the same machine where it is built and generates code for the same machine), you don't need to specify any options on the command line when you run the `configure` script. However, if you are building a cross-compiler, you must specify the target machine. Similarly, if you are building a compiler that will be installed on some other machine, you have to specify the host machine.

NOTE There may be some other requirements when you are building a non-native compiler.

It is recommended to run the `configure` script in a directory other than the source code directory so that the source code tree is completely separate from the place where you build the compiler. For the sake of this example, I have created a directory `/gcc3/objdir` and I ran the `configure` script from that directory. As you already know, the source code tree is under the `/gcc3/gcc-3.0.4` directory. You may need to add a prefix for GCC installation files. The prefix shows the directory where GCC files will be finally installed. The default prefix is `/usr/local`. This means that GCC binary files will be installed in `/usr/local/bin` directory. For installation purposes, I have selected `/opt/gcc-3.0.4` directory as the prefix. In a typical development environment, you would do something similar to that. Just create a directory under `/opt` and install all of your tools and applications under `/opt`. The following command line will configure the GCC compilation process and will create many files under `/gcc3/objdir`. These files will be used later on to build the compiler. When you start the `configure` script, the following messages will start scrolling up.

```
[root@laptop objdir]# ../gcc-3.0.4/configure --prefix=/opt/
gcc-3.0.4
Configuring for a i686-pc-linux-gnu host.
Created "Makefile" in /gcc3/gcc-3.0.4 using "mt-frag"
Configuring libiberty...
creating cache ../config.cache
checking host system type... i686-pc-linux-gnu
checking build system type... i686-pc-linux-gnu
checking for ar... ar
checking for ranlib... ranlib
checking for gcc... gcc
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking for POSIXized ISC... no
checking for working const... yes
checking for inline... inline
checking for a BSD compatible install... /usr/bin/install -c
checking how to run the C preprocessor... gcc -E
checking for sys/file.h... yes
checking for sys/param.h... yes
checking for limits.h... yes
checking for stdlib.h... yes
checking for string.h... yes
```

Most of the output is truncated from the configure script to save space. When the configure script is completed, you will get back the command prompt. Now you may start building the compiler.

If you want to enable threads on Linux systems, you can use `--enable-threads=posix` as a command line option.

3.2.2.3 Building GCC

It is recommended that you use GNU `make` for building purpose. Running the `make bootstrap` command will build the compiler, libraries and related utilities. Following is part of the output when you start the building process.

```
[root@laptop objdir]# make bootstrap
make[1]: Entering directory `/gcc3/objdir/libiberty'
if [ x"" != x ] && [ ! -d pic ]; then \
  mkdir pic; \
else true; fi
touch stamp-picdir
if [ x"" != x ]; then \
  gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/
libiberty/../include  -W -Wall -Wtraditional -pedantic  ../../
gcc-3.0.4/libiberty/argv.c -o pic/argv.o; \
else true; fi
gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/libiberty/
../include  -W -Wall -Wtraditional -pedantic ../../gcc-3.0.4/
libiberty/argv.c
if [ x"" != x ]; then \
  gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/
libiberty/../include  -W -Wall -Wtraditional -pedantic  ../../
gcc-3.0.4/libiberty/choose-temp.c -o pic/choose-temp.o; \
else true; fi
gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/libiberty/
../include  -W -Wall -Wtraditional -pedantic ../../gcc-3.0.4/
libiberty/choose-temp.c
if [ x"" != x ]; then \
  gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/
libiberty/../include  -W -Wall -Wtraditional -pedantic  ../../
gcc-3.0.4/libiberty/concat.c -o pic/concat.o; \
else true; fi
gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/libiberty/
../include  -W -Wall -Wtraditional -pedantic ../../gcc-3.0.4/
libiberty/concat.c
if [ x"" != x ]; then \
  gcc -c -DHAVE_CONFIG_H -g -O2 -I. -I../../gcc-3.0.4/
libiberty/../include  -W -Wall -Wtraditional -pedantic  ../../
gcc-3.0.4/libiberty/cplus-dem.c -o pic/cplus-dem.o; \
else true; fi
```

Again most of the output is truncated. The building process may take a long time depending upon how powerful a computer you are using for building GCC. The process is completed in

three stages. Stage 1 and stage 2 binaries are deleted as soon as they are no longer needed. Typically this building process will do the following:

- Build some basic tools like bison, gperf and so on. These tools are necessary to continue the compilation process.
- Build target tools like gas, ld, binutils and so on.
- Perform a three-stage bootstrap of the compiler.
- Perform comparison of stage 2 and stage 3.
- Build run-time libraries.
- Remove unnecessary files.

You should have enough disk space to build GCC. If you are short of disk space, you can use the `bootstrap-lean` option on the command line instead of `bootstrap`.

3.2.2.4 Final Install

The final installation process is necessary to move GCC binaries under the directory used as a prefix during the `configure` script execution. We have used `/opt/gcc-3.0.4` as the prefix directory. Executing the following command in `/gcc3/objdir` will move GCC binaries in a tree structure under `/opt/gcc-3.0.4`.

```
make install
```

A typical directory tree after installation is shown below.

```
[root@conformix gcc-3.0.4]# tree -d |more
.
|-- bin
|-- include
|   |-- g++-v3
|   |   |-- backward
|   |   |-- bits
|   |   |-- ext
|   |   `-- i686-pc-linux-gnu
|   |       `-- bits
|   |-- gcj
|   |-- gnu
|   |   |-- awt
|   |   |   `-- j2d
|   |   |-- classpath
|   |   |-- gcj
|   |   |   |-- awt
|   |   |   |-- convert
|   |   |   |-- io
|   |   |   |-- jni
|   |   |   |-- math
|   |   |   |-- protocol
```

```
|   |   |   |   |-- file
|   |   |   |   |-- http
|   |   |   |   `-- jar
|   |   |   |-- runtime
|   |   |   |-- text
|   |   |   `-- util
|   |   `-- java
|   |       |-- beans
|   |       |   |-- editors
|   |       |   `-- info
|   |       |-- io
|   |       |-- lang
|   |       |   `-- reflect
|   |       |-- locale
|   |       `-- security
|   |           `-- provider
|   `-- java
|       |-- applet
|       |-- awt
|       |   |-- color
|       |   |-- datatransfer
|       |   |-- event
|       |   |-- geom
|       |   |-- image
|       |   `-- peer
|       |-- beans
|       |   `-- beancontext
|       |-- io
|       |-- lang
|       |   |-- ref
|       |   `-- reflect
|       |-- math
|       |-- net
|       |-- security
|       |   |-- cert
|       |   |-- interfaces
|       |   `-- spec
|       |-- sql
|       |-- text
|       `-- util
|           |-- jar
|           `-- zip
|-- info
|-- lib
|   `-- gcc-lib
|       `-- i686-pc-linux-gnu
|           `-- 3.0.4
|               `-- include
```

```
|                   |-- SDL
|                   |-- X11 -> root/usr/X11R6/include/X11
|                   |-- linux
|                   |-- mozilla
|                   |-- ncurses
|                   |-- objc
|                   |-- openssl
|                   |-- pcap
|                   |   `-- net
|                   |-- pgsql
|                   |   `-- utils
|                   |-- root
|                   |   `-- usr
|                   |       `-- X11R6
|                   |           `-- include
|                   |               `-- X11
|                   |-- schily
|                   |   `-- scg
|                   |-- slang
|                   |-- ucd-snmp
|                   |-- w3c-libwww
|                   |-- wnn
|                   `-- wnn6
|-- man
|   |-- man1
|   `-- man7
`-- share

95 directories
[root@conformix gcc-3.0.4]#
```

NOTE Detailed instructions for compiling and installing GCC are available at http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc_4.html as well as at http://gcc.gnu.org/install/.

3.2.3 Environment Variables

The GCC compiler relies on many environment variables for its operation. These variables are used for different purposes including location of library and header files, location of helping programs and so on. Some important variables and their respective use are introduced in this section.

`TMPDIR` This variable shows location of temporary file location. GCC uses this location to store temporary files during the compiling and linking processes.

GCC_EXEC_PREFIX	If this variable is set, GCC will look into the directory to find sub programs.
COMPILER_PATH	This is a colon-separated list of directories that GCC uses to find out sub programs if search fails using GCC_EXEC_PREFIX variable.
LIBRARY_PATH	This is a colon-separated list of directories that is used to find out libraries for linking process.
C_INCLUDE_PATH	Colon separated list of directories to find out header files for C programs.
OBJC_INCLUDE_PATH	Colon separated list of directories to find out header files for Objective C programs.
CPLUS_INCLUDE_PATH	Colon separated list of directories to find out header files for C++ programs.
LD_LIBRARY_PATH	Path for shared libraries.

There are other environment variables and settings that GCC uses while building a target. You can display these using the –v command line switch with the gcc command when you compile a program. This will show you the path including files, programs used during the compilation process, and command line arguments and switches passed to each of these programs. The following is the output of the command when you compile the hello.c program.

```
[rr@conformix 4]$ gcc -v hello.c
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/specs
Configured with: ../gcc-3.0.4/configure --prefix=/opt/gcc-
3.0.4 --enable-threads=posix
Thread model: posix
gcc version 3.0.4
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/cc1 -lang-
c -v -D__GNUC__=3 -D__GNUC_MINOR__=0 -D__GNUC_PATCHLEVEL__=4 -
D__ELF__ -Dunix -Dlinux -D__ELF__ -D__unix__ -D__linux__ -
D__unix -D__linux -Asystem=posix -D__NO_INLINE__ -
D__STDC_HOSTED__=1 -Acpu=i386 -Amachine=i386 -Di386 -D__i386 -
D__i386__ -D__tune_i686__ -D__tune_pentiumpro__ hello.c -quiet
-dumpbase hello.c -version -o /tmp/ccJsUmYa.s
GNU CPP version 3.0.4 (cpplib) (i386 Linux/ELF)
GNU C version 3.0.4 (i686-pc-linux-gnu)
compiled by GNU C version 3.0.4.
ignoring nonexistent directory "/opt/gcc-3.0.4/i686-pc-linux-
gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
```

```
 /opt/gcc-3.0.4/include
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/include
 /usr/include
End of search list.
 as --traditional-format -V -Qy -o /tmp/ccn7wLgw.o /tmp/
ccJsUmYa.s
GNU assembler version 2.10.91 (i386-redhat-linux) using BFD
version 2.10.91.0.2
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/collect2 -
m elf_i386 -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
/usr/lib/crti.o /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/
3.0.4/crtbegin.o -L/opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4 -L/opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/
3.0.4/../../.. /tmp/ccn7wLgw.o -lgcc -lc -lgcc /opt/gcc-3.0.4/
lib/gcc-lib/i686-pc-linux-gnu/3.0.4/crtend.o /usr/lib/crtn.o
[rr@conformix 4]$
```

If you examine the output of this command, you can find out which helper programs `gcc` uses and what command line switches are passed to these programs.

3.2.4 Post-Installation Tasks

There are a few tasks that you need to carry out after the installation process of the compilers.

3.2.4.1 Setting PATH Variable

This is the first important task. Your `PATH` variable must include the directory where GCC binaries are installed. We have installed these in `/opt/gcc-3.0.4/bin` directory because we used `/opt/gcc-3.0.4` as the prefix while running the `configure` script. This directory should come before the directories where the old compiler is installed. A typical command to do this in bash or other POSIX-compliant shells to include our installation location is as follows:

```
export PATH=/opt/gcc-3.0.4/bin:$PATH
```

where `/opt/gcc-3.0.4/bin` is the path to newly installed compilers.

It is also extremely important that you make sure the GCC in the path is the correct one. The '`which gcc`' command will provide this.

3.2.4.2 Setting the Location of Libraries

There are two steps to set up the location of libraries. First edit `/etc/ld/so.config` and add the path of any newly created libraries. This directory is `/opt/gcc-3.0.4/lib` because we used `–prefix=/opt/gcc-3.0.4` while building the compiler. Typical contents of this file after adding the new directory are as follows:

```
/opt/gcc-3.0.4/lib
/usr/lib
/usr/kerberos/lib
/usr/X11R6/lib
/usr/lib/sane
```

```
/usr/lib/qt-2.3.0/lib
/usr/lib/mysql
/usr/lib/qt-1.45/lib
```

After editing this file, execute the `ldconfig` program, which will configure dynamic linker binding. You can use the –v command line option to get more information when you run this command. Note that the order of commands is important.

The next step is to setup the `LD_LIBRARY_PATH` variable. You can do this by adding the following line at the end of `/etc/profile` file so that it is set for all users at login time.

```
export LD_LIBRARY_PATH=/opt/gcc-3.0.4/lib
```

Again note that this is the path where new library files are installed. Please note that if you make these changes, some older programs that are compiled for some other set of shared libraries may not function properly.

3.2.4.3 Setting Location of include Files

The default search path for include files can be found by executing the following command:

```
[rr@conformix 4]$ gcc -v -E -
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/specs
Configured with: ../gcc-3.0.4/configure --prefix=/opt/gcc-
3.0.4 --enable-threads=posix
Thread model: posix
gcc version 3.0.4
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/cpp0 -
lang-c -v -D__GNUC__=3 -D__GNUC_MINOR__=0 -
D__GNUC_PATCHLEVEL__=4 -D__ELF__ -Dunix -Dlinux -D__ELF__ -
D__unix__ -D__linux__ -D__unix -D__linux -Asystem=posix -
D__NO_INLINE__ -D__STDC_HOSTED__=1 -Acpu=i386 -Amachine=i386 -
Di386 -D__i386 -D__i386__ -D__tune_i686__ -
D__tune_pentiumpro__ -
GNU CPP version 3.0.4 (cpplib) (i386 Linux/ELF)
ignoring nonexistent directory "/opt/gcc-3.0.4/i686-pc-linux-
gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /opt/gcc-3.0.4/include
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/include
 /usr/include
End of search list.
```

The last part of the output shows that include files will be searched in the following directories by default.

```
/usr/local/include
```

```
/opt/gcc-3.0.4/include
/opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/
include
/usr/include
```

You can place include files in other directories if you have set the C_INCLUDE_PATH environment variable. Setting this variable to /opt using the following command will include /opt directory also in the search list. Also note that order is once again extremely important.

```
export C_INCLUDE_PATH=/opt
```

If you again execute the gcc -v -E - command, /opt path will be included in the last part of the output.

To set additional include paths permanently for all users, it is a good idea to add a line with the export command in /etc/profile file.

3.2.4.4 Setting Manual Pages Path

To be able to use manual pages installed with GCC, you have to add a line in the /etc/man.config file. This will enable the man command to also look into the /opt/gcc-3.0.4/man directory when searching for manual pages. The line looks like the following:

```
MANPATH /opt/gcc-3.0.4/man
```

Location of this line with respect to other MANPATH entries is important. If you put this line after other entries, you will still get the same old man pages. That is why it is recommended to put this entry BEFORE any other line that starts with MANPATH keyword in this file.

3.2.5 What Not to Do when Installing Development Tools

When building GCC, don't build it into the source directory. I would also recommend not installing your development tools in the default location (under /usr/local). Instead use some place under the /opt directory. This way if something goes wrong, you can just delete this directory under /opt without making any other change in the system. If you install these tools in the default location, you may overwrite some existing files and may not be able to reverse the installation process.

3.3 Compiling a Program

The GCC compiler is commonly invoked using the gcc command. The command accepts many command line switches that can be used to invoke different options for the compilation process. You can use the same command line switch multiple times. For example, if you need to specify multiple include paths, you can use -I option multiple times. However you can't combine two switches into one. For example -c and -o can't be combined as -co. This section provides information about different methods of compilation and things you need to consider in the compilation process. Please note that in most of software development projects, you don't invoke gcc from the command line. Instead, the GNU make utility that reads one or many Makefiles

is used. These Makefiles contain information about how the compiler will be invoked and what command line switches will be used. Information about GNU `make` and Makefiles is presented in Chapter 5.

3.3.1 Simple Compilation

Consider the following C source code file, which is named `hello.c`. We shall frequently refer to this program in this as well as coming chapters.

```
#include <stdio.h>
main ()
{
  printf("Hello world\n");
}
```

To compile this file, you just need to run the following command.

```
[rr@conformix 4]$ gcc hello.c
[rr@conformix 4]$
```

By default, this command will generate an output file named `a.out,` which can be executed on the command line as follows:

```
[rr@conformix 4]$ ./a.out
Hello world
[rr@conformix 4]$
```

Note that regardless of the name of your source code file, the name of the output file is always `a.out`. You may actually know what an ordinary `a.out` file is, but this isn't one of them. It is an `elf` file, despite its name. If you want to create an output file with a different name, you have to use the –`o` command line option. The following command creates an output file with name `hello`.

```
gcc hello.c -o hello
```

As you may have noted, the above commands do both the compiling and linking processes in a single step. If you don't want to link the program, then simple compilation of `hello.c` file can be done with the following command. The output file will be `hello.o` and will contain object code.

```
gcc –c hello.c
```

Note that both –`c` and –`o` command line switches can be used simultaneously. The following command compiles `hello.c` and produces an output file `test.o` which is not yet linked.

```
gcc –c hello.c  -o test.o
```

Usually when you compile many files in a project, you don't create `a.out` files. Either you compile many files into object files and then link them together into an application or make executables with the same name as the source code file.

3.3.2 Default File Types

GCC can recognize an input file by the last part of its name, sometimes called an extension. Table 3-1 shows file types and the extensions used with them. Depending upon a particular extension, `gcc` takes appropriate action to build the output file.

Table 3-1 File types used with GCC

File Extension	File Type
.c	C source code file.
.cc	C++ source code file.
.cp	C++ source code file.
.cxx	C++ source code file.
.cpp	C++ source code file.
.c+	C++ source code file.
.C	C++ source code file.
.m	Objective C source code file.
.F	Fortran source code file.
.fpp	Fortran source code file.
.FPP	Fortran source code file.
.h	C header file.
.i	C source code file. GCC does not preprocess it.
.ii	C++ source code file. GCC does not preprocess it.
.mi	Objective C source code file. GCC does not preprocess it.
.f	Fortran source code file. GCC does not preprocess it.
.for	Fortran source code file. GCC does not preprocess it.
.FOR	Fortran source code file. GCC does not preprocess it.
.s	Assembler code. GCC does not preprocess it.
.S	Assembler file.

This means that if you use command `gcc hello.c`, GCC will consider `hello.c` as a C program and will invoke appropriate helper programs to build the output. However, if you use `gcc hello.cpp` command, GCC will consider `hello.cpp` as a C++ program and will compile it accordingly. You can also select a language type with a particular file using –x command line option. Table 3-2 lists languages that can be selected with this option.

Table 3-2 Selecting languages with –x option.

Option	Language
-x c (lowercase c)	C language selection
-x c++	C++ file
-x objective-c	Objective C
-x assembler	Assembler file
-x f77	Fortran file
-x java	Java language file

Note that –x option applies to all the files that follow until you turn it off using the –x `none` option on the command line. This is especially important when you use the GNU `make` utility discussed in Chapter 5.

By default, the object file created by GCC has `.o` extension, replacing the original extension of the file. You can create output file with a particular name using –o command line option. The following command creates `test.o` file from `hello.c`.

```
gcc -c hello.c -o test.o
```

3.3.3 Compiling to Intermediate Levels

The compilation process involves many steps like preprocessing, assembling and linking. By default GCC carries out all of these processes and generates executable code as you have seen earlier. However, you can force GCC not to do all of these steps. For example, using the –c command line option, the `gcc` command will only compile a source code file and will not generate executable object code. As you already know, the following command compiles file `hello.c` and creates an object file `hello.o`.

```
gcc -c hello.c
```

If you look at the type of the newly created object file using the `file` command, you can see that this is not a linked file. This is shown in the following command output.

```
[root@conformix chap-03]# file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1,
not stripped
[root@conformix chap-03]#
```

3.3.3.1 Creating Assembler Code

Using the –S (uppercase S) command line option, you can stop the GCC compiler just before the assembler process. The output is an assembler file with a `.s` extension. The following command creates an output file `hello.s` from source code file `hello.c`.

```
gcc -S hello.c
```

If you look at the output file, you can see the assembler code. Contents of the input file are as follows:

```
#include <stdio.h>
main()
{
  printf ("Hello world\n");
}
```

The output assembler code is shown below:

```
[root@conformix chap-03]# cat hello.s
        .file   "hello.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "Hello world\n"
.text
        .align 4
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1
2.96-81)"
[root@conformix chap-03]#
```

This assembler code may be used with some assembler, like GNU as. Here *as* is not word "as" but name of the GNU Assembler which is often written as *GNU as as*, later on. It can also be assembled and linked to create and execute. Please note that for the above command, the compiler that is included in RedHat distribution was used.

3.3.4 Compilation with Debug Support

If you want to debug a program after compiling with `gcc`, you have to include debug information in the compiled program. The debug information is included in object file using the –g command line switch with `gcc`. The following command creates the `hello.o` file that contains debug information.

```
[rr@conformix 4]$ gcc -g -c hello.c
[rr@conformix 4]$
```

Note that when you compile a program with debug information, the size may be quite large as compared to a file compiled without debug information. In the example program of `hello.c`, the size of `hello.o` is 908 bytes when compiled without debug information. The size of `hello.o` is 10780 bytes when it is compiled with debug information.

You can use multiple debug levels with –g option. The default debug level is 2 which is equivalent to using –g2 command line option. If you use –g3 command line option, information about macros is also included which makes it easier to debug macros.

NOTE You can use the debug option with optimization options. Optimization options are discussed later in this chapter.

With the –a option on the command line, you can also include some profiling information in the object code.

You can also use some command line switches to provide extra information. For example, one useful thing is to print out a list of directories that the `gcc` command searches to find files. The following command will print all directories that `gcc` uses to search libraries, programs and so on.

```
[rr@conformix 4]$ gcc -print-search-dirs hello.c -o hello
install: /usr/lib/gcc-lib/i386-redhat-linux/2.96/
programs: =/usr/lib/gcc-lib/i386-redhat-linux/2.96/:/usr/lib/
gcc-lib/i386-redhat-linux/2.96/:/usr/lib/gcc-lib/i386-redhat-
linux/:/usr/lib/gcc/i386-redhat-linux/2.96/:/usr/lib/gcc/i386-
redhat-linux/:/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../
../../i386-redhat-linux/bin/i386-redhat-linux/2.96/:/usr/lib/
gcc-lib/i386-redhat-linux/2.96/../../../../i386-redhat-linux/
bin/
libraries: =/usr/lib/gcc-lib/i386-redhat-linux/2.96/:/usr/lib/
gcc/i386-redhat-linux/2.96/:/usr/lib/gcc-lib/i386-redhat-
linux/2.96/../../../../i386-redhat-linux/lib/i386-redhat-
linux/2.96/:/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../
../i386-redhat-linux/lib/:/usr/lib/gcc-lib/i386-redhat-linux/
2.96/../../../i386-redhat-linux/2.96/:/usr/lib/gcc-lib/i386-
redhat-linux/2.96/../../../:/lib/i386-redhat-linux/2.96/:/lib/
:/usr/lib/i386-redhat-linux/2.96/:/usr/lib/
[rr@conformix 4]$
```

Here I have used GCC version 2.96 that came with RedHat Linux 7.1 and you can see directories and references to this version information also.

You can also find the amount of time taken by each process during compilation. The following command displays time taken during each step of building the output file.

```
[rr@conformix 4]$ gcc -time hello.c -o hello
# cpp0 0.06 0.00
# cc1 0.08 0.01
# as 0.02 0.00
# collect2 0.12 0.03
[rr@conformix 4]$
```

It is also evident from the output of the above command that GCC has used four other programs (cpp0, cc1, as and collect2) during the compilation process.

3.3.5 Compilation with Optimization

The first objective of a compiler is to generate output code swiftly. The compiler does not do any code optimization to make the compile time short. However you can instruct `gcc` to compile code with code optimization. This is done using `–O` (uppercase O, not zero) on the command line. Different optimization levels can be designated by using a number suffix with this option. For example, `-O2` will do code optimization at level 2. If you specifically don't want to do any code optimization, you can use zero with option as –O0.

So what does optimization mean? Consider the following C source code file `sum.c` that calculates the sum of two numbers and prints the result. Of course this is not the best code for this purpose and it is used only to demonstrate a point.

```
 1  #include <stdio.h>
 2  main ()
 3  {
 4    int a, b, sum;
 5
 6    a=4;
 7    b=3;
 8    sum = a+b;
 9
10    printf("The sum is: %d\n", sum);
11  }
```

If you compile this program without any optimization, the compiler will generate code for all lines starting from line number 6 to line number 10. This can be verified by loading the file in a debugger and tracing through it. However, if you optimize the compilation process, lines 6 to 10 can be replaced by a single line as shown below. This can be done without affecting the output of the program.

```
printf("The sum is: 7\n", );
```

This is because the compiler can easily determine that all of the variables are static and there is no need to assign values and then calculate the sum at the run time. All of this can be done at the compile time. You can also verify this fact in a debugger. The optimized code will skip over assignment lines (lines 6 to 8) and will directly jump to the `printf` statement when you step through.

However in the following code, the compiler can't make such decisions because the numbers `a` and `b` are entered interactively.

```
 1  #include <stdio.h>
 2  main ()
 3  {
 4    int a, b, sum;
 5
 6    printf("Enter first number: ");
 7    scanf("%d", &a);
 8    printf("Enter second number: ");
 9    scanf("%d", &b);
10
11    sum = a+b;
12
13    printf("The sum is: %d\n", sum);
14  }
```

If you compile this code with different levels of optimization (e.g., –O1 and –O2), and then trace it through a debugger, you will see a difference in execution sequence because of the way the compiler makes decisions at the compile time.

It may be mentioned that optimization is not always beneficial. For example, code optimization changes timings or clock cycles when the code is executed. This especially may create some problems on embedded systems if you have debugged your code by compiling without optimization. The rule of thumb is that you should create optimized code instead of relying on the compiler to make optimization for you.

For a detailed list of optimization options, please see all options starting with `–f` command line option. However options starting with `–O` are the most commonly used in the optimization process.

3.3.6 Static and Dynamic Linking

A compiler can generate static or dynamic code depending upon how you proceed with the linking process. If you create static object code, the output files are larger but they can be used as stand-alone binaries. This means that you can copy an executable file to another system and it does not depend on shared libraries when it is executed. On the other hand, if you chose dynamic linking, the final executable code is much smaller but it depends heavily upon shared libraries. If you copy the final executable program to another system, you have to make sure that the shared libraries are also present on the system where your application is executed. Please note that version inconsistencies in dynamic libraries can also cause problems.

To create static binaries, you have to use –static command line option with gcc. To created dynamically linked output binary files, use –shared on the command line.

For example, if we compile the hello.c program used earlier in this chapter with shared libraries, size of the output executable file is 13644 bytes (this can be further reduced using the strip utility discussed later in Chapter 7 of this book). However, if you compile it statically, the size of the output binary file is 1625261 bytes, which is very large compared to the shared binary. Note that this size can also be reduced using the strip utility.

To identify the dependencies of a dynamically linked binary file, you can use the ldd command. The following command shows that linked output file hello depends upon two dynamic libraries.

```
[rr@conformix 4]$ ldd hello
  libc.so.6 => /lib/i686/libc.so.6 (0x4002c000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[rr@conformix 4]$
```

If you copy hello to some other host, you also need to make sure that libc.so.6 and ld-linux.so.2 exist on the target system.

On most of the Linux systems, dynamic linking is done by default.

3.3.7 Compiling Source Code for Other Languages

As mentioned earlier, the GCC set of compilers supports many languages. It can be used to compile programs other than C language. Following is an introduction to compiling programs from other languages.

3.3.7.1 Compiling C++ Code

C++ source code files have suffixes such as .C, .cpp, .cc, .c++, .cxx or .cp. The gcc compiler recognizes these extensions and can compile C++ code. However you can also use g++ or c++ compilers, which are part of the GCC compilers family and are installed with it. These programs invoke gcc with appropriate options to compile C++ code and location of class files. Using these programs, you can also compile C++ source code files that don't have the standard suffixes listed earlier.

3.3.7.2 Compiling Objective C Code

Objective files have suffixes such as .m and gcc recognizes Objective C files with that suffix. When you compile Objective C code, you have to pass an option to the linker. This is done using –lobjc. By this option, the linker uses Objective C libraries during the linking process. Consider the following sample Objective C code (stored in hello.m file) to print "Hello World" on the standard output.

```
#include "objc/Object.h"

@interface HelloWorld : Object
{
```

```
  STR msg;
}

+ new;
- print;
- setMessage: (STR) str;

@end

@implementation HelloWorld

+ new
{
  self = [super new];
  [self setMessage : ""];
  return self;
}

- print
{
  printf("%s\n", msg);
  return self;
}

- setMessage: (STR) str
{
  msg = str;
  return self;
}

@end


int main(int argc, char**argv) {
  id msg;

  msg = [HelloWorld new];

  [msg setMessage: "Hello World"] ;
  [msg print];
  return 0;
}
```

You can compile and link it using the `gcc hello.m -lobjc` command. The output is again `a.out` file that can be executed on the command line.

This is sort of a long "Hello World" program. There are much shorter Objective C "Hello World" programs available on the Internet.

3.3.7.3 Compiling Java Code

Information about the GCC Java compiler `gcj` is available at http://gcc.gnu.org/java/. Before you can build Java programs, you also need to have `libgcj` installed. With old compilers, you had to install it separately from source code. When you build and install new versions of GCC, `libgcj` is installed automatically. If you are still using an old compiler and want to have `libgcj` installed, information is available at http://gcc.gnu.org/java/libgcj2.html. Briefly the process is as follows:

Download `libgcj` from ftp://sourceware.cygnus.com/pub/java/ or another web site on the Internet. Untar it in `/opt` directory and a new directory will be created under `/opt` which will contain source code for `libgcj`. Create a directory `/opt/libgcj-build` and move into this directory. After that you have to perform the following sequence of commands:

- ../libgcj/configure
- make
- make install

Note that your new compiler must be in `PATH` before you build `libgcj`.

Now let us see how to compile a Java program. Consider the following simple Java program that prints the message "Hello World". The source code filename is `hello.java`.

```
class HelloWorld {
  public static void main (String args[]) {
    System.out.print("Hello World ");
  }
}
```

Traditionally you have to invoke the `javac` program to build a byte code. After that you have to run the byte code using the `java` program on Linux. However if you use `gcj`, you can create a binary output file `hello` using the following command:

```
gcj –main=HelloWorld –o hello hello.java
```

The output file is `hello`, which is an executable binary. The `–main` switch is used for the entry point when the program is executed.

The compiler uses some information to build Java code. This information includes reading the `gcj` specification file and libraries. The following command displays this information.

```
[rr@conformix 4]$ gcj -v
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/specs
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/../../../libgcj.spec
rename spec lib to liborig
rename spec startfile to startfileorig
```

```
Configured with: ../gcc-3.0.4/configure --prefix=/opt/gcc-
3.0.4 --enable-threads=posix
Thread model: posix
gcc version 3.0.4
[rr@conformix 4]$
```

The compilation of Java programs is completed in many steps. Let us compile the `hello.java` program to build a statically linked `hello` output binary using the following command. The –v switch shows all of the steps during this process.

```
[rr@conformix 4]$ gcj hello.java --main=HelloWorld  -o hello -
static -v
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/specs
Reading specs from /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-
gnu/3.0.4/../../../libgcj.spec
rename spec lib to liborig
rename spec startfile to startfileorig
Configured with: ../gcc-3.0.4/configure --prefix=/opt/gcc-
3.0.4 --enable-threads=posix
Thread model: posix
gcc version 3.0.4
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/jc1
hello.java -fuse-divide-subroutine -fuse-boehm-gc -fnon-call-
exceptions -quiet -dumpbase hello.java -g1 -version -o /tmp/
ccHj5WMY.s
GNU Java version 3.0.4 (i686-pc-linux-gnu)
  compiled by GNU C version 3.0.4.
 as --traditional-format -V -Qy -o /tmp/cchm92Nc.o /tmp/
ccHj5WMY.s
GNU assembler version 2.10.91 (i386-redhat-linux) using BFD
version 2.10.91.0.2
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/jvgenmain
HelloWorldmain /tmp/ccTlFcXz.i
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/cc1 /tmp/
ccTlFcXz.i -quiet -dumpbase HelloWorldmain.c -g1 -version -
fdollars-in-identifiers -o /tmp/ccHj5WMY.s
GNU CPP version 3.0.4 (cpplib) (i386 Linux/ELF)
GNU C version 3.0.4 (i686-pc-linux-gnu)
  compiled by GNU C version 3.0.4.
 as --traditional-format -V -Qy -o /tmp/ccBgJjpa.o /tmp/
ccHj5WMY.s
GNU assembler version 2.10.91 (i386-redhat-linux) using BFD
version 2.10.91.0.2
 /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/collect2 -
m elf_i386 -static -o hello /usr/lib/crt1.o /usr/lib/crti.o /
opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/crtbegin.o -
L/opt/gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4 -L/opt/
```

```
gcc-3.0.4/lib/gcc-lib/i686-pc-linux-gnu/3.0.4/../../.. /tmp/
ccBgJjpa.o /tmp/cchm92Nc.o -lgcc -lgcj -lm -lgcjgc -lpthread -
lzgcj -ldl -lgcc -lc -lgcc /opt/gcc-3.0.4/lib/gcc-lib/i686-pc-
linux-gnu/3.0.4/crtend.o /usr/lib/crtn.o
[rr@conformix 4]$
```

As you can see, different programs have been executed to get the output binary file. These programs include:

- The `jcl` program
- GNU assembler as. Again as is the name of the assembler. `as`
- The `jvgenmain` program
- The `cc1` compiler
- The `collect2` program

You can also see various command line switches used with these programs.

3.3.8 Summary of gcc Options

Hundreds of options can be used with `gcc` on command line. Explanation of all of these options is beyond the scope of this book. However, following is a summary list of these options as displayed by `gcc` man page (using `man gcc` command). Options are grouped in different sections which will be helpful for you if you are looking for options related to a particular task.

3.3.8.1 Overall Options

```
-c
-S
-E
-o file
-pipe
-pass-exit-codes
-x language
-v
--target-help
--help
```

3.3.8.2 C Language Options

```
-ansi
-std=standard
-aux-info filename
-fno-asm
-fno-builtin
-fhosted
-ffree-standing
-trigraphs
-traditional
```

```
-traditional-cpp
-fallow-single-precision
-fcond-mismatch
-fsigned-bitfields
-fsigned-char
-funsigned-bitfields
-funsigned-char

-fwritable-strings
-fshort-wchar
```

3.3.8.3 C++ Language Options

```
-fno-access-control
-fcheck-new
-fconserve-space
-fno-const-strings
-fdollars-in-identifiers
-fno-elide-constructors
-fno-enforce-eh-specs
-fexternal-templates

-falt-external-templates
-ffor-scope
-fno-for-scope
-fno-gnu-keywords
-fno-implicit-templates
-fno-implicit-inline-templates
-fno-implement-inlines
-fms-extensions
-fno-nonansi-builtins
-fno-operator-names
-fno-optional-diags
-fpermissive
-frepo
-fno-rtti
-fstats
-ftemplate-depth-n
-fuse-cxa-atexit
-fno-weak
-nostdinc++
-fno-default-inline
-Wctor-dtor-privacy
-Wnon-virtual-dtor
-Wreorder
-Weffc++
-Wno-deprecated
-Wno-non-template-friend
```

```
-Wold-style-cast
-Woverloaded-virtual
-Wno-pmf-conversions
-Wsign-promo
-Wsynth
```

3.3.8.4 Objective-C Language Options

```
-fconstant-string-class=class-name
-fgnu-runtime
-fnext-runtime
-gen-decls
-Wno-protocol
-Wselector
```

3.3.8.5 Language Independent Options

```
-fmessage-length=n
-fdiagnostics-show-location=[once|every-line]
```

3.3.8.6 Warning Options

```
-fsyntax-only
-pedantic
-pedantic-errors
-w
-W
-Wall
-Waggregate-return
-Wcast-align
-Wcast-qual
-Wchar-subscripts
-Wcomment
-Wconversion
-Wdisabled-optimization

-Werror
-Wfloat-equal
-Wformat
-Wformat=2
-Wformat-nonliteral
-Wformat-security
-Wid-clash-len
-Wimplicit
-Wimplicit-int
-Wimplicit-function-declaration
-Werror-implicit-function-declaration
-Wimport
-Winline
-Wlarger-than-len
```

```
-Wlong-long
-Wmain
-Wmissing-braces
-Wmissing-declarations
-Wmissing-format-attribute
-Wmissing-noreturn
-Wmultichar
-Wno-format-extra-args
-Wno-format-y2k
-Wno-import
-Wpacked
-Wpadded
-Wparentheses
-Wpointer-arith
-Wredundant-decls
-Wreturn-type
-Wsequence-point
-Wshadow
-Wsign-compare
-Wswitch
-Wsystem-headers
-Wtrigraphs
-Wundef
-Wuninitialized
-Wunknown-pragmas
-Wunreachable-code
-Wunused
-Wunused-function
-Wunused-label
-Wunused-parameter
-Wunused-value
-Wunused-variable
-Wwrite-strings
```

3.3.8.7 C-only Warning Options

```
-Wbad-function-cast
-Wmissing-prototypes
-Wnested-externs
-Wstrict-prototypes
-Wtraditional
```

3.3.8.8 Debugging Options

```
-a
-ax
-dletters
-dumpspecs
-dumpmachine
```

```
-dumpversion
-fdump-unnumbered
-fdump-translation-unit[-n]
-fdump-class-hierarchy[-n]
-fdump-ast-original[-n]
-fdump-ast-optimized[-n]
-fmem-report
-fpretend-float
-fprofile-arcs
-ftest-coverage

-ftime-report -g
-glevel
-gcoff
-gdwarf
-gdwarf-1
-gdwarf-1+
-gdwarf-2
-ggdb
-gstabs
-gstabs+
-gxcoff
-gxcoff+
-p
-pg
-print-file-name=library
-print-libgcc-file-name
-print-multi-directory
-print-multi-lib
-print-prog-name=program
-print-search-dirs
-Q
-save-temps
-time
```

3.3.8.9 Optimization Options

```
-falign-functions=n
-falign-jumps=n
-falign-labels=n
-falign-loops=n
-fbranch-probabilities
-fcaller-saves
-fcse-follow-jumps
-fcse-skip-blocks
-fdata-sections
-fdce -fdelayed-branch
-fdelete-null-pointer-checks
```

```
-fexpensive-optimizations

-ffast-math
-ffloat-store
-fforce-addr
-fforce-mem
-ffunction-sections

-fgcse
-finline-functions
-finline-limit=n
-fkeep-inline-functions

-fkeep-static-consts
-fmove-all-movables
-fno-default-inline
-fno-defer-pop
-fno-function-cse
-fno-guess-branch-probability
-fno-inline
-fno-math-errno
-fno-peephole
-fno-peephole2
-fomit-frame-pointer
-foptimize-register-move

-foptimize-sibling-calls
-freduce-all-givs
-fregmove
-frename-registers

-frerun-cse-after-loop
-frerun-loop-opt
-fschedule-insns
-fschedule-insns2

-fsingle-precision-constant
-fssa
-fstrength-reduce
-fstrict-aliasing
-fthread-jumps
-ftrapv
-funroll-all-loops
-funroll-loops --param name=value -O
-O0
-O1
-O2
-O3
-Os
```

3.3.8.10 Preprocessor Options

```
-$
-Aquestion=answer
-A-question[=answer]
-C
-dD
-dI
-dM
-dN
-Dmacro[=defn]
-E
-H
-idirafter dir
-include file
-imacros file
-iprefix file
-iwithprefix dir
-iwithprefixbefore dir
-isystem dir
-M
-MM
-MF
-MG
-MP
-MQ
-MT
-nostdinc
-P
-remap
-trigraphs
-undef
-Umacro
-Wp,option
```

3.3.8.11 Assembler Option

```
-Wa,option
```

3.3.8.12 Linker Options

```
 object-file-name
-llibrary
-nostartfiles
-nodefaultlibs
-nostdlib
-s
-static
-static-libgcc
-shared
```

```
-shared-libgcc
-symbolic
-Wl,option
-Xlinker option
 -u symbol
```

3.3.8.13 Directory Options

```
-Bprefix
-Idir
-I-
-Ldir
-specs=file
```

3.3.8.14 Target Options

```
-b machine
-V version
```

3.3.8.15 Machine Dependent Options

M680x0 Options

```
-m68000
-m68020
-m68020-40
-m68020-60
-m68030
-m68040
-m68060
-mcpu32
-m5200
-m68881
-mbitfield
-mc68000
-mc68020
-mfpa
-mnobitfield
-mrtd
-mshort
-msoft-float
-mpcrel
-malign-int
-mstrict-align
```

M68hc1x Options

```
-m6811
-m6812
-m68hc11
-m68hc12
-mauto-incdec
```

```
-mshort
-msoft-reg-count=count
```

VAX Options

```
-mg
-mgnu
-munix
```

SPARC Options

```
-mcpu=cpu-type
-mtune=cpu-type
-mcmodel=code-model
-m32
-m64
-mapp-regs
-mbroken-saverestore

-mcypress
-mepilogue
-mfaster-structs
-mflat
-mfpu
-mhard-float
-mhard-quad-float
-mimpure-text
-mlive-g0
-mno-app-regs
-mno-epilogue
-mno-faster-structs
-mno-flat
-mno-fpu
-mno-impure-text
-mno-stack-bias
-mno-unaligned-doubles
-msoft-float
-msoft-quad-float
-msparclite
-mstack-bias
-msupersparc
-munaligned-doubles
-mv8
```

Convex Options

```
-mc1
-mc2
-mc32
-mc34
```

```
-mc38
-margcount
-mnoargcount
-mlong32
-mlong64
-mvolatile-cache
-mvolatile-nocache
```

AMD29K Options

```
-m29000
-m29050
-mbw
-mnbw
-mdw
-mndw
-mlarge
-mnormal
-msmall
-mkernel-registers

-mno-reuse-arg-regs
-mno-stack-check
-mno-storem-bug
-mreuse-arg-regs
-msoft-float
-mstack-check
-mstorem-bug
-muser-registers
```

ARM Options

```
-mapcs-frame
-mno-apcs-frame
-mapcs-26
-mapcs-32
-mapcs-stack-check
-mno-apcs-stack-check
-mapcs-float
-mno-apcs-float
-mapcs-reentrant
-mno-apcs-reentrant
-msched-prolog
-mno-sched-prolog
-mlittle-endian
-mbig-endian
-mwords-little-endian
-malignment-traps
-mno-alignment-traps
-msoft-float
```

```
-mhard-float
-mfpe
-mthumb-interwork
-mno-thumb-interwork
-mcpu=name
-march=name
-mfpe=name
-mstructure-size-boundary=n
-mbsd
-mxopen
-mno-symrename
-mabort-on-noreturn
-mlong-calls
-mno-long-calls
-msingle-pic-base
-mno-single-pic-base
-mpic-register=reg
-mnop-fun-dllimport
-mpoke-function-name
-mthumb
-marm
-mtpcs-frame
-mtpcs-leaf-frame
-mcaller-super-interworking
-mcallee-super-interworking
```

MN10200 Options

```
-mrelax
```

MN10300 Options

```
-mmult-bug
-mno-mult-bug
-mam33
-mno-am33
-mno-crt0
-mrelax
```

M32R/D Options

```
-mcode-model=model-type
-msdata=sdata-type -G num
```

M88K Options

```
-m88000
-m88100
-m88110
-mbig-pic
-mcheck-zero-division
```

```
-mhandle-large-shift
-midentify-revision
-mno-check-zero-division
-mno-ocs-debug-info
-mno-ocs-frame-position
-mno-optimize-arg-area
-mno-serialize-volatile
-mno-underscores
-mocs-debug-info
-mocs-frame-position
-moptimize-arg-area
-mserialize-volatile

-mshort-data-num
-msvr3
-msvr4
-mtrap-large-shift
-muse-div-instruction
-mversion-03.00
-mwarn-passed-structs
```

RS/6000 and PowerPC Options

```
-mcpu=cpu-type
-mtune=cpu-type
-mpower
-mno-power
-mpower2
-mno-power2
-mpowerpc

-mpowerpc64
-mno-powerpc
-mpowerpc-gpopt
-mno-powerpc-gpopt
-mpowerpc-gfxopt

-mno-powerpc-gfxopt
-mnew-mnemonics
-mold-mnemonics
-mfull-toc
-mminimal-toc

-mno-fop-in-toc
-mno-sum-in-toc
-m64
-m32
-mxl-call
-mno-xl-call
```

```
-mthreads
-mpe
-msoft-float
-mhard-float
-mmultiple
-mno-multiple
-mstring
-mno-string
-mupdate
-mno-update
-mfused-madd
-mno-fused-madd
-mbit-align
-mno-bit-align
-mstrict-align
-mno-strict-align
-mrelocatable
-mno-relocatable
-mrelocatable-lib
-mno-relocatable-lib
-mtoc
-mno-toc
-mlittle
-mlittle-endian
-mbig
-mbig-endian
-mcall-aix
-mcall-sysv
-mcall-netbsd
-mprototype
-mno-prototype-

-msim
-mmvme
-mads
-myellowknife
-memb
-msdata
-msdata=opt
-mvxworks –G num
```

RT Options

```
-mcall-lib-mul
-mfp-arg-in-fpregs
-mfp-arg-in-gregs
-mfull-fp-blocks
-mhc-struct-return
-min-line-mul
```

```
-mminimum-fp-blocks
-mnohc-struct-return
```

MIPS Options

```
-mabicalls
-mcpu=cpu-type
-membedded-data
-muninit-const-in-rodata
-membedded-pic-

-mfp32
-mfp64
-mgas
-mgp32
-mgp64
-mgpopt
-mhalf-pic
-mhard-float
-mint64
-mips1
-mips2
-mips3
-mips4
-mlong64
-mlong32
-mlong-calls
-mmemcpy
-mmips-as
-mmips-tfile
-mno-abicalls
-mno-embedded-data
-mno-uninit-const-in-rodata
-mno-embedded-pic
-mno-gpopt
-mno-long-calls
-mno-memcpy
-mno-mips-tfile
-mno-rnames
-mno-stats
-mrnames
-msoft-float
-m4650
-msingle-float
-mmad
-mstats
-EL
-EB
-G num
```

```
-nocpp
-mabi=32
-mabi=n32
-mabi=64
-mabi=eabi -mfix7000
-mno-crt0
```

i386 Options

```
-mcpu=cpu-type
-march=cpu-type
-mintel-syntax
-mieee-fp
-mno-fancy-math-387
-mno-fp-ret-in-387
-msoft-float
-msvr3-shlib
-mno-wide-multiply
-mrtd
-malign-double
-mreg-alloc=list
-mregparm=num
-malign-jumps=num
-malign-loops=num
-malign-functions=num
-mpreferred-stack-boundary=num
-mthreads
-mno-align-stringops
-minline-all-stringops
-mpush-args
-maccumulate-outgoing-args

-m128bit-long-double
-m96bit-long-double
-momit-leaf-frame-pointer
```

HPPA Options

```
-march=architecture-type
-mbig-switch
-mdisable-fpregs
-mdisable-indexing
-mfast-indirect-calls
-mgas
-mjump-in-delay
-mlong-load-store
-mno-big-switch
-mno-disable-fpregs
-mno-disable-indexing
-mno-fast-indirect-calls
```

```
-mno-gas
-mno-jump-in-delay
-mno-long-load-store
-mno-portable-runtime
-mno-soft-float
-mno-space-regs
-msoft-float
-mpa-risc-1-0
-mpa-risc-1-1
-mpa-risc-2-0
-mportable-runtime
-mschedule=cpu-type
-mspace-regs
```

Intel 960 Options

```
-mcpu-type
-masm-compat
-mclean-linkage
-mcode-align
-mcomplex-addr
-mleaf-procedures
-mic-compat
-mic2.0-compat
-mic3.0-compat
-mintel-asm
-mno-clean-linkage
-mno-code-align
-mno-complex-addr
-mno-leaf-procedures
-mno-old-align
-mno-strict-align
-mno-tail-call
-mnumerics
-mold-align
-msoft-float
-mstrict-align
-mtail-call
```

DEC Alpha Options

```
-mfp-regs
-mno-fp-regs
-mno-soft-float
-msoft-float
-malpha-as
-mgas
-mieee
-mieee-with-inexact
-mieee-conformant
```

```
-mfp-trap-mode=mode
-mfp-rounding-mode=mode

-mtrap-precision=mode
-mbuild-constants
-mcpu=cpu-type
-mbwx
-mno-bwx
-mcix
-mno-cix
-mmax
-mno-max
-mmemory-latency=time
```

Clipper Options

```
-mc300
-mc400
```

H8/300 Options

```
-mrelax
-mh
-ms
-mint32
-malign-300
```

SH Options

```
-m1
-m2
-m3
-m3e
-m4-nofpu
-m4-single-only
-m4-single
-m4
-mb
-ml
-mdalign
-mrelax
-mbigtable
-mfmovd
-mhitachi
-mnomacsave
-mieee
-misize
-mpadstruct
-mspace
-mprefergot
-musermode
```

System V Options

```
-Qy
-Qn
-YP,paths
-Ym,dir
```

ARC Options

```
-EB
-EL
-mmangle-cpu
-mcpu=cpu
```

-mtext=text-section

-mdata=data-section

-mrodata=readonly-data-section

TMS320C3x/C4x Options

```
-mcpu=cpu
-mbig
-msmall
-mregparm
-mmemparm
-mfast-fix
-mmpyi
-mbk
-mti
-mdp-isr-reload -mrpts=count
-mrptb
-mdb
-mloop-unsigned
-mparallel-insns
-mparallel-mpy
-mpreserve-float
```

V850 Options

```
-mlong-calls
-mno-long-calls
-mep
-mno-ep
-mprolog-function
-mno-prolog-function

-mspace
-mtda=n
-msda=n
-mzda=n
-mv850
-mbig-switch
```

NS32K Options

```
-m32032
-m32332
-m32532
-m32081
-m32381
-mmult-add
-mnomult-add
-msoft-float
-mrtd
-mnortd
-mregparam
-mnoregparam
-msb
-mnosb
-mbitfield
-mnobitfield
-mhimem
-mnohimem
```

AVR Options

```
-mmcu=mcu
-msize
-minit-stack=n
-mno-interrupts
-mcall-prologues
-mno-table-jump

-mtiny-stack
```

MCore Options

```
-mhardlit
-mno-hardlit
-mdiv
-mno-div
-mrelax-immediates
-mno-relax-immediates
-mwide-bitfields
-mno-wide-bitfields
-m4byte-functions
-mno-4byte-functions
-mcallgraph-data
-mno-callgraph-data
-mslow-bytes
-mno-slow-bytes
-mno-lsim
-mlittle-endian
```

```
-mbig-endian
-m210
-m340
-mstack-increment
```

IA-64 Options

```
-mbig-endian
-mlittle-endian
-mgnu-as
-mgnu-ld
-mno-pic
-mvolatile-asm-stop
-mb-step
-mregister-names
-mno-sdata
-mconstant-gp
-mauto-pic
-minline-divide-min-latency

-minline-divide-max-throughput
-mno-dwarf2-asm
-mfixed-range=register-range
```

S/390 and zSeries Options

```
-mhard-float
-msoft-float
-mbackchain
-mno-backchain
-msmall-exec
-mno-small-exec
-mmvcle
-mno-mvcle
-m64
-m31
-mdebug
-mno-debug
```

Xtensa Options

```
-mbig-endian
-mlittle-endian
-mdensity
-mno-density
-mmac16
-mno-mac16
-mmul16
-mno-mul16
-mmul32
```

```
-mno-mul32
-mnsa
-mno-nsa
-mminmax
-mno-minmax
-msext
-mno-sext
-mbooleans
-mno-booleans
-mhard-float
-msoft-float
-mfused-madd
-mno-fused-madd
-mserialize-volatile
-mno-serialize-volatile
-mtext-section-literals
-mno-text-section-literals
-mtarget-align
-mno-target-align
-mlongcalls
-mno-long-calls
```

3.3.8.16 Code Generation Options

```
-fcall-saved-reg
-fcall-used-reg
-ffixed-reg
-fexceptions
-fnon-call-exceptions
-funwind-tables
-finhibit-size-directive
-finstrument-functions
-fcheck-memory-usage

-fprefix-function-name
-fno-common
-fno-ident
-fno-gnu-linker
-fpcc-struct-return
-fpic
-fPIC
-freg-struct-return
-fshared-data
-fshort-enums
-fshort-double
-fvolatile
-fvolatile-global
-fvolatile-static
```

```
-fverbose-asm
-fpack-struct
-fstack-check
-fstack-limit-register=reg
-fstack-limit-symbol=sym
-fargument-alias
-fargument-noalias
-fargument-noalias-global

-fleading-underscore
```

3.4 Linking a program

As mentioned earlier, building an executable binary file from a source code file is a multi-stage process.

The `gcc` compiler usually carries all these steps for you by default. However you can stop `gcc` at any of these stages for various reasons. You have already seen how to create assembler code from a C source code file. In most software projects, output binaries are built from many source code files. For this purpose you have to compile these source code files to object files one by one before you can generate the output binary file. After creating these object files, you can link these files using `gcc` to the final executable.

The `gcc` compiler gets help from different programs during the four steps listed earlier in this chapter. During the linking part, the `ld` program is invoked with appropriate arguments to link one or many object files together with libraries to generate an executable. Command line switches used with `ld` can be found using manual pages.

The linker is a part of GNU binary utilities package also known as binutils. You can download and install the latest version of GNU linker if you need to do so.

3.5 Assembling a Program

GNU assembler is part of GNU binary utilities package. To get the latest binutils packages, you can download it from ftp://ftp.gnu.org/gnu/binutils/. After downloading you have to use the `tar` command to extract source code files. The latest version at the time of writing this chapter is 2.12 and the `tar` command will create a directory `binutils-2.12` automatically. I have extracted it in `/opt` directory, so the source code directory is `/opt/binutils-2.12`. I created another directory `/opt/binutils-build` to build the package. Running the `configure` script is the first step towards building binary utilities package. The following sequence of commands does all of these steps.

```
[root@conformix /opt]# cd /opt
[root@conformix /opt]# tar zxvf binutils-2.12.tar.gz
[root@conformix /opt]# mkdir binutils-build
[root@conformix /opt]# cd binutils-build/
[root@conformix binutils-build]# ../binutils-2.12/configure --
prefix=/opt/gcc-3.0.4 --enable-shared
```

```
[root@conformix /opt]# make LDFLAGS=-all-static tooldir=/opt/
gcc-3.0.4
[root@conformix /opt]# make tooldir=/opt/gcc-3.0.4 install
```

Note that since we have used `/opt/gcc/3.0.4` as a prefix and `tooldir` as the destination for installation, the binary utilities will be installed in `/opt/gcc-3.0.4/bin` directory. As you may have noted, I have installed all software under `/opt/gcc-3.0.4` so that I can use one directory for all of my tools.

3.6 Handling Warning and Error messages

The GNU compiler generates two types of messages when compiling source code. Warning messages are non-critical messages and the compiler can build the output code even if there are warning messages. Error messages are fatal messages and compiler is not able to generate output files when an error occurs. By default, the compiler does not display many warning messages. You may like to see these messages, for example, when you want to get a release of a software package. Usually the release version of software should be compiled without any warning message. Using options that start with `–W`, you can control which types of messages should be displayed. It is a good idea to always use `–Wall` option in your Makefiles. The reason is that warnings often signify problems in code that you will want to see. Please see `gcc` man pages to get a feel of options controlling warning messages.

3.7 Include files

During the compilation process, the compiler should know where include files will be located. This can be done in different ways. The compiler looks into some default locations when searching for include files and you have already seen in this chapter how to display that information. You can also use the `–I` command line switch with the `gcc` command to set additional paths to locate include files. The third method is the use of environment variables. Please see the environment variables section earlier in this chapter for more information.

3.8 Creating Libraries

Libraries are files containing commonly used functions needed for all programs. For example, `printf()` is a function used in most of the C programs. When the program is linked, the linker must know where the code for the `printf()` function is located. Knowledge of location of library files is a must at this stage. The location can be passed to the linker using command line options or environment variables. In large software projects, this is done inside Makefiles. The environment variable that shows the location of library files is `LD_LIBRARY_PATH`. Please see the list of command line options in this chapter or use manual pages to find out which options are appropriate.

3.9 Standard Libraries

In addition to the compiler, assembler, linker and other tools, you also need standard library files. All Linux distributions come with the GNU C library which is installed as `glibc`. However, when you are building your development environment, you may need to get the library in source code format and install the latest version. You may also need the library in source code format in order to cross compile it when you are building a cross-compilation environment. The standard GNU C library provides functions for the following major areas.

- Memory management
- Error handling
- Character handling
- Strings and arrays
- Locales and international specific things, like character handling and date and time specific functions
- Searching
- Sorting
- Pattern matching
- Input and output
- File system support
- Pipes
- Fifos
- Sockets
- Terminal I/O
- Mathematics functions
- Date and time functions
- Signal and exception handling
- Process management
- Job control
- User and groups
- System related tasks

Information about `glibc` is available on http://www.gnu.org/software/libc/libc.html. You can download it from ftp://ftp.gnu.org/gnu/glibc/ where you can find the latest as well as old versions. The current version at the time of writing this book is 2.2.5.

After downloading the library, you have to untar it. For the sake of this book, I untarred the library file in `/opt` directory. The following `tar` command extracts library files in `/opt/glibc-2.2.5` directory.

```
tar zxvf glibc-2.2.5.tar.gz
```

Now you have to extract any other components or add-ons of the library. I have added one component, Linux threads. File `glibc-linuxthreads-2.2.5.tar.gz` can also be

downloaded from the same location from which you downloaded the main `glibc-2.2.5.tar.gz` file. The following commands are used to extract files from these two archives:

```
cd glibc-2.2.5
tar zxvf ../ glibc-linuxthreads-2.2.5.tar.gz
```

In old versions of `glibc`, you had to install add-ons like locale data and crypt library. These are included in the latest version of `glibc`.

Now you have to configure the library after creating a build directory. Basically this is the same process that you have done while installing GCC and binutils. This is done using the following set of commands:

```
[root@conformix glibc-2.2.5]# mkdir build
[root@conformix glibc-2.2.5]# cd build
[root@conformix build]# ../configure --enable-add-
ons=linuxthreads --prefix=/opt/gcc-3.0.4
```

The actual compilation and testing is done using the following two commands:

```
make
make check
```

The final install command is the following:

```
make install
```

This command will install components of GNU C library under `/opt/gcc-3.0.4/lib` because we had chosen `/opt/gcc/3.0.4` as our prefix when we ran the `configure` script. Now you can set the appropriate paths in your Makefiles to use the new library.

3.10 Compiling Pascal Programs

Pascal programs can be converted to C language programs and then compiled in the usual way. The `p2c` command on Linux does this for you. Consider the following Pascal program `hello.pas`.

```
(* Program to demonstrate Pascal compilation *)
program Hello ;
begin
   writeln ('Hello world')
end.
```

The following command will create a file `hello.c` which is the equivalent C version of the `hello.pas` program.

```
[rr@conformix 4]$ p2c hello.pas
Hello

Translation completed.
[rr@conformix 4]$
```

The output `hello.c` file is shown below:

```
/* Output from p2c 1.21alpha-07.Dec.93, the Pascal-to-C
translator */
/* From input file "hello.pas" */


/* Program to demonstrate Pascal compilation */

#include <p2c/p2c.h>


main(argc, argv)
int argc;
Char *argv[];
{
  PASCAL_MAIN(argc, argv);
  printf("Hello world\n");
  exit(EXIT_SUCCESS);
}




/* End. */
```

You may need some libraries with `p2c` program and C compiler to successfully complete this process.

Many Pascal compilers are also available in the open source world. These compilers can be used to generate executable code from Pascal source directly without any help from GCC.

3.10.1 Using Free Pascal (fpc)

The Free Pascal Project has created a 32-bit Pascal compiler which is available for Linux, among other operating systems. You can download it from http://www.freepascal.org. This software is licensed under GPL like other open source projects. At the time of writing this book version 1.0.4 is available for download. After download, you can install the compiler as follows on a RedHat system:

```
[root@conformix rr]# rpm --install fpc-1.0.4.i386.rpm
Write permission in /etc.
Found libgcc.a in /usr/lib/gcc-lib/i386-redhat-linux/2.96
Writing sample configuration file to /etc/ppc386.cfg
[root@conformix rr]#
```

The best thing about this compiler is that you don't need to convert source code files to C language and then compile these. The fpc compiler creates binary executable files. The following command creates an executable file `hello` from `hello.pas` program listed earlier.

```
[rr@conformix 4]$ fpc hello.pas
Free Pascal Compiler version 1.0.4 [2000/12/18] for i386
Copyright (c) 1993-2000 by Florian Klaempfl
Target OS: Linux for i386
Compiling hello.pas
Assembling hello
Linking hello
6 Lines compiled, 0.1 sec
[rr@conformix 4]$
```

The output `hello` program can be executed like any other program.

3.10.2 Using GNU Pascal

A GNU Pascal compiler is available from http://agnes.dida.physik.uni-essen.de/~gnu-pascal/ for download and can be installed on many platforms like `gcc`. The web site contains all manuals and downloadable files for this compiler. This compiler also creates executable files from Pascal source code.

3.11 Compiling Fortran Programs

The GCC family of compilers also includes `g77` which acts as a front end to invoke `gcc`. The `gcc` compiler is used to compile Fortran programs. Consider the following simple `hello.for` Fortran program that write "Hello world" on standard output.

```
c This program is to display a string and then end
c Written to demonstrate Fortran program compilation
c
c Rafeeq Rehman, March 18, 2002

      program hello
      write (*,*) 'Hello world'
      stop
      end
```

Note that statements in Fortran start from column 7. All lines starting with c in column 1 are comment lines. You can compile this program using the following command:

```
g77 hello.for
```

The `g77` compiler, like `gcc`, creates `a.out` as a default output executable. You can create output file with a different name using `–o` command line option. The following command creates "kaka" as output binary executable file.

```
g77 hello.for –o kaka
```

The `g77` program uses many programs to generate output binary. You can display version information by using `g77 –v` command. Output of this command is shown below.

```
[rr@conformix 4]$ g77 -v
g77 version 2.96 20000731 (Red Hat Linux 7.1 2.96-81) (from
FSF-g77 version 0.5.26 20000731 (Red Hat Linux 7.1 2.96-81))
Driving: g77 -v -c -xf77-version /dev/null -xnone
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/
specs
gcc version 2.96 20000731 (Red Hat Linux 7.1 2.96-81)
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/tradcpp0 -lang-
fortran -v -D__GNUC__=2 -D__GNUC_MINOR__=96 -
D__GNUC_PATCHLEVEL__=0 -D__ELF__ -Dunix -Dlinux -D__ELF__ -
D__unix__ -D__linux__ -D__unix -D__linux -Asystem(posix) -
Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -
D__tune_i386__ /dev/null /dev/null
GNU traditional CPP version 2.96 20000731 (Red Hat Linux 7.1
2.96-81)
 /usr/lib/gcc-lib/i386-redhat-linux/2.96/f771 -fnull-version -
quiet -dumpbase g77-version.f -version -fversion -o /tmp/
ccr29cz4.s /dev/null
GNU F77 version 2.96 20000731 (Red Hat Linux 7.1 2.96-81)
(i386-redhat-linux) compiled by GNU C version 2.96 20000731
(Red Hat Linux 7.1 2.96-81).
GNU Fortran Front End version 0.5.26 20000731 (Red Hat Linux
7.1 2.96-81)
 as -V -Qy -o /tmp/ccqJAAN9.o /tmp/ccr29cz4.s
GNU assembler version 2.10.91 (i386-redhat-linux) using BFD
version 2.10.91.0.2
 ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o /tmp/
ccijxxjb /tmp/ccqJAAN9.o /usr/lib/gcc-lib/i386-redhat-linux/
2.96/../../../crt1.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/
../../../crti.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/
crtbegin.o -L/usr/lib/gcc-lib/i386-redhat-linux/2.96 -L/usr/
lib/gcc-lib/i386-redhat-linux/2.96/../../.. -lg2c -lm -lgcc -
lc -lgcc /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtend.o /
usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crtn.o
 /tmp/ccijxxjb
__G77_LIBF77_VERSION__: 0.5.26 20000731 (prerelease)
@(#)LIBF77 VERSION 19991115
__G77_LIBI77_VERSION__: 0.5.26 20000731 (prerelease)
@(#) LIBI77 VERSION pjw,dmg-mods 19991115
__G77_LIBU77_VERSION__: 0.5.26 20000731 (prerelease)
@(#) LIBU77 VERSION 19980709
[rr@conformix 4]$
```

As you can see, it takes the help of `gcc` to compile programs. Note that Fortran programs have `.for` or `.f` at the end of the name of the program file.

As with `gcc`, you can create intermediate assembled or object code with `g77`. See manual pages of `g77` for more information.

3.12 Other Compilers

A major benefit of using Linux as a development platform is that you have so many tools and compilers available in the open source community that you can develop in virtually any standard language. This section provides short information about a few more languages that can be easily integrated into the Linux development platform.

3.12.1 Smalltalk

Smalltalk is an object oriented language and a GNU version of Smalltalk can be downloaded from ftp://ftp.gnu.org/gnu/smalltalk/. The latest version at the time of writing this book is 1.95.9. The compilation and installation is quite easy and is similar to other GNU tools. You can also go to www.smalltalk.org for more information.

3.12.2 Oberon

Oberon is the successor to the Pascal and Modula languages and it is an object-oriented language. It is mostly used in educational environments for teaching programming language concepts. For detailed information, please refer to http://www.oberon.ethz.ch/.

3.12.3 Ruby

Ruby is an interpreted language. Information about this language can be found at http://www.ruby-lang.org/en/.

3.13 References and Resources

1. GNU web site at http://www.gnu.org/
2. Languages supported by GCC at http://gcc.gnu.org/onlinedocs/gcc-3.0.2/gcc_2.html
3. Status of C99 support on GCC at http://gcc.gnu.org/gcc-3.0/c99status.html
4. Object Oriented Programming and Objective C at http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/
5. Object Programming and Objective Language at http://www.toodarkpark.org/computers/objc/
6. The GNU C Library at http://www.delorie.com/gnu/docs/glibc/libc_toc.html
7. Using as at http://www.objsw.com/docs/as_toc.html
8. Using and porting GNU cc at http://www.delorie.com/gnu/docs/gcc/gcc_toc.html
9. GNU C++ Library at http://www.delorie.com/gnu/docs/libg++/libg++_toc.html
10. NetBSD Library source code at http://www.ajk.tele.fi/libc/code.html
11. Free Pascal project at http://www.freepascal.org
12. GNU Pascal at http://agnes.dida.physik.uni-essen.de/~gnu-pascal/
13. GNU Smalltalk download site at ftp://ftp.gnu.org/gnu/smalltalk/
14. Smalltalk information at http://www.smalltalk.org

15. Bob Nevlen, Linux Assembly Language Programming ISBN 0-13-0877940-1, Prentice Hall PTR 2000.

16. Oberon Home page at http://www.oberon.ethz.ch/

17. Ruby Home Page at http://www.ruby-lang.org/en/