1 Getting Started

1.1 Compiling with GCC

A compiler turns human-readable source code into machine-readable object code that can actually run.The compilers of choice on Linux systems are all part of the GNU Compiler Collection, usually known as GCC. GCC also compiles Fortran (under the auspices of g77). Front-ends for Pascal, Modula-3, Ada 9X, and other languages are in various stages of development.The compilation process includes up to four stages:

- Preprocessing
- Compilation Proper
- Assembly
- Linking

You can stop the process after any of these stages to examine the compiler's output at that stage.

GCC includes over 30 individual warnings and three "catch-all" warning levels. GCC is also a cross-compiler, so you can develop code on one processor architecture that will be run on another. Finally, GCC sports a long list of extensions to C and C++. Most of these extensions enhance performance, assist the compiler's efforts at code optimization, or make your job as a programmer easier. The price is portability, however.
This program will compute the reciprocal of an integer. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "reciprocal.hpp"
int main (int argc, char **argv)
{
  int i;
  i = atoi (argv[1]);
  printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
  return 0;
}
```

reciprocal.cpp

```
#include <cassert>
#include "reciprocal.hpp"
double reciprocal (int i) {
```

```
// I should be non-zero.
assert (i != 0);
return 1.0/i;
}
```

There is also one header file called reciprocal.hpp

```
#ifdef __cplusplus
extern "C" {
#endif
extern  double reciprocal (int i);
#ifdef __cplusplus
}
#endif
```

The first step is to turn the C and C++ source code into object code.

1.1.1 Compiling a Single Source File

To compile a C source file, you use the **-c** option.

```
% gcc -c main.c
```

To tell GCC to stop compilation after preprocessing, use GCC's -E option:

```
% gcc -E main.c -o main.pp
```

Examine **main.pp** and you can see the contents of *stdio.h, stdlib.h* and *reciprocal.hpp* have indeed been inserted into the file, along with other preprocessing tokens. The next step is to compile **main.pp** to object code. Use GCC's -c option to accomplish this:

```
%gcc -x cpp-output -c main.pp -o main.o
```

The most common extensions and their interpretation are listed in Table 1.

The resulting object file is named *main.o*. The C++ compiler is called **g++**.

```
% g++ -c reciprocal.cpp
```

The **-c** option tells g++ to compile the program to an object file only; without it, g++ will attempt to link the program to produce an executable. After you've typed this command, you'll have an object file called *reciprocal.o*.

The **-I** option is used to tell GCC where to search for header files. By default, GCC looks in the current directory and in the directories where headers for the standard libraries are installed. If you need to include header files from somewhere else, you'll need the -I option.

Table 1: How GCC interprets filename extensions.

Extension	Type
.c	C language source code
.C, .cc	C++ language source code
.i	Preprocessed C source code
.ii	Preprocessed C++ source code
.S, .s	Assembly language source code
.o	Compiled object code
.a, .so	Compiled library code

```
% g++ -c -I ../include reciprocal.cpp
```

If you don't want the overhead of the assertion check present in *reciprocal.cpp*; that's only there to help you debug the program.You turn off the check by defining the macro **NDEBUG**.

```
% g++ -c -D NDEBUG reciprocal.cpp
```

If you had wanted to define NDEBUG to some particular value, you could have done something like this:

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

If you are really building production code, you probably want to have GCC optimize the code so that it runs as quickly as possible.You can do this by using the **-O2** command-line option. Static variables may vanish or loops may be unrolled, so that the optimized program does not correspond line-for-line with the original source code. (GCC has several different levels of optimization; the second level is appropriate for most programs.)

```
% g++ -c -O2 reciprocal.cpp
```

Note that compiling with optimization can make your program more difficult to debug with a debugger. Also, in certain instances, compiling with optimization can uncover bugs in your program that did not manifest themselves previously.

1.1.2 Optimization Options

Code optimization is an attempt to improve performance. The trade-off is lengthened compile times and increased memory usage during compilation.

- The bare *-O* option tells GCC to reduce both code size and execution time.
- It is equivalent to *-O1*. The types of optimization performed at this level depend on the target processor, but always include at least thread jumps and deferred stack pops.
  - Thread jump optimizations attempt to reduce the number of jump operations,
  - deferred stack pops occur when the compiler lets arguments accumulate on the stack as functions return and then pops them simultaneously, rather than popping the arguments piecemeal as each called function returns.
- *-O2* level optimizations include all first-level optimization plus additional tweaks that involve processor instruction scheduling. At this level, the compiler takes care to make sure the processor has instructions to execute while waiting for the results of other instructions or data latency from cache or main memory. The implementation is highly processor-specific.
- *-O3* options include all *O2* optimizations, loop unrolling, and other processor-specific features. Depending on the amount of low-level knowledge you have about a given CPU family, you can use the -fflag option to request specific optimizations you want performed. Three of these flags bear consideration: *-ffastmath, -finline-functions,* and *-funroll-loops*.
  - *-ffastmath* generates floating-point math optimizations that increase speed, but violate IEEE and/or ANSI standards.
  - *-finline-functions* expands all "simple" functions in place, much like preprocessor macro replacements. Of course, the compiler decides what constitutes a simple function.
  - *-funroll-loops* instructs GCC to unroll all loops that have a fixed number of iterations that can be determined at compile time.
- Inlining and loop unrolling can greatly improve a program's execution speed because they avoid the overhead of function calls and variable lookups, but the cost is usually a large increase in the size of the binary or object files.
- You will have to experiment to see if the increased speed is worth the increased file size. See the GCC info pages for more details on processor flag.

1.1.3 Linking Object Files

You should always use g++ to link a program that contains C++ code, even if it also contains C code. If your program contains only C code, you should use gcc instead. Because this program contains both C and C++, you should use g++, like this:

```
% g++ -o reciprocal main.o reciprocal.o
% ./reciprocal 7
The reciprocal of 7 is 0.142857
```

As you can see, g++ has automatically linked in the standard C runtime library containing the implementation of **printf**. If you had needed to link in another library (such as a graphical user interface toolkit), you would have specified the library with the **-l** option. In Linux, library names almost always start with **lib**. For example, the Pluggable Authentication Module (PAM) library is called **libpam.a**.

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

The compiler automatically adds the **lib** prefix and the **.a** suffix.

As with header files, the linker looks for libraries in some standard places, including the **/lib** and **/usr/lib** directories that contain the standard system libraries. If you want the linker to search other directories as well, you should use the **-L** option. You can use this line to instruct the linker to look for libraries in the /usr/local/lib/pam directory before looking in the usual places:

```
% g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
```

Although you don't have to use the **-I** option to get the preprocessor to search the current directory, you do have to use the -L option to get the linker to search the current directory. In particular, you could use the following to instruct the linker to find the test library in the current directory:

```
% gcc -o app app.o -L. -ltest
```

By default, gcc uses shared libraries, so if you must link against static libraries, you have to use the *-static* option. This means that only static libraries will be used. The following example creates an executable linked against the static ncurses.

```
% gcc cursesapp.c -lncurses -static
```

- When you link against static libraries, the resulting binary is much larger than using shared libraries.

- Why use a static library, then? One common reason is to guarantee that users can run your program in the case of shared libraries, the code your program needs to run is linked dynamically at runtime, rather than statically at compile time.
- If the shared library your program requires is not installed on the user's system, she will get errors and not be able to run your program.

1.1.4 Common Command-line Options

The list of command-line options gcc accepts runs to several pages, so we will only look at the most common ones in Table 2.

1.1.5 Error Checking and Warnings

GCC boasts a whole class of *error-checking*, *warning-generating*, *command-line options*. These include *-ansi, -pedantic, -pedantic- errors,* and *-Wall*. To begin with,

- -pedantic tells GCC to issue all warnings demanded by strict ANSI/ISO standard C. Any program using forbidden extensions, such as those supported by GCC, will be rejected.
- -pedantic-errors behaves similarly, except that it emits errors rather than warnings.
- -ansi, finally, turns off GNU extensions that do not comply with the standard.

None of these options, however, guarantee that your code, when compiled without error using any or all of these options, is 100 percent ANSI/ISO-compliant.

Consider pedant.c , an example of very bad programming form. It declares **main()** as returning **void**, when in fact **main()** returns **int**, and it uses the GNU extension **long long** to declare a 64-bit integer.

```
// pedant.c - use -ansi, -pedantic or -pedantic-errors
#include <stdio.h>
 void main(void)
{
        long long int i = 01;
        fprintf(stdout, "This is a non-conforming C program\n");
}
```

Using

Table 2: GCC command-line options.

Option	Description
-o	FILE Specify the output filename; not necessary when compiling to object code. If FILE is not specified, the default name is a.out.
-c	Compile without linking.
-DFOO=BAR	Define a preprocessor macro named FOO with a value of BAR on the command-line.
-IDIRNAME	Prepend DIRNAME to the list of directories searched for include files.
-LDIRNAME	Prepend DIRNAME to the list of directories searched for library files. By default, gcc links against shared libraries.
-static	Link against static libraries.
-lFOO	Link against libFOO.
-g	Include standard debugging information in the binary.
-ggdb	Include lots of debugging information in the binary that only the GNU debugger, gdb, can understand.
-O	Optimize the compiled code.
-ON	Specify an optimization level N, $0 <= N <= 3$.
-ansi	Support the ANSI/ISO C standard, turning off GNU extensions that conflict with the standard (this option does not guarantee ANSI-compliant code).
-pedantic	Emit all warnings required by the ANSI/ISO C standard.
-pedantic-errors	Emit all errors required by the ANSI/ISO C standard.
-traditional	Support the Kernighan and Ritchie C language syntax (such as the old-style function definition syntax). If you don't understand what this means, don't worry about it.
-w	Suppress all warning messages. In my opinion, using this switch is a very bad idea!
-Wall	Emit all generally useful warnings that gcc can provide. Specific warnings can also be flagged using -Wwarning.
-werror	Convert all warnings into errors, which will stop the compilation.
-MM	Output a make-compatible dependency list.
-v	Show the commands used in each step of compilation.

```
%gcc pedant.c -o pedant
```

this code compiles without complaint.

- First, try to compile it using **-ansi**:

  ```
  %gcc -ansi pedant.c -o pedant
  ```

  Again, no complaint. The lesson here is that **-ansi** forces GCC to emit the diagnostic messages required by the standard. It does not insure that your code is ANSI C compliant. The program compiled despite the deliberately incorrect declaration of main().

- Now, **-pedantic**:

  ```
  %gcc -pedantic pedant.c -o pedant
  pedant.c: In function ‘main’:
  pedant.c:9: warning: ANSI C does not support ‘long long’
  ```

  The code compiles, despite the emitted warning.

- With **-pedantic- errors**, however, it does not compile. GCC stops after emitting the error diagnostic:

  ```
  %gcc -pedantic-errors pedant.c -o pedant
  pedant.c: In function ‘main’:
  pedant.c:9: ANSI C does not support ‘long long’
  ```

To reiterate, the **-ansi, -pedantic,** and **-pedantic-errors** compiler options do not insure ANSI/ISO-compliant code. They merely help you along the road.

Some users try to use **-pedantic** to check programs for strict ANSI C conformance.

1.2 Automating the Process with GNU Make

- The basic idea behind **make** is simple.You tell **make** what targets you want to build and then give rules explaining how to build them. You also specify dependencies that indicate when a particular target should be rebuilt.

- **make** minimizes rebuild times because it is smart enough to determine which files have changed, and thus only rebuilds files whose components have changed.

- **make** maintains a database of dependency information for your projects and so can verify that all of the files necessary for building a program are available each time you start a build.
- A makefile is a text file database containing rules that tell make what to build and how to build it. A rule consists of the following:
  - A target, the "thing" make ultimately tries to create
  - A list of one or more dependencies, usually files, required to build the target
  - A list of commands to execute in order to create the target from the specified dependencies
- When invoked, GNU make looks for a file named GNUmakefile, makefile, or Makefile, in that order. For some reason, most Linux programmers use the last form, Makefile. Makefile rules have the general form

```
target : dependency dependency [...]
command
command
[...]
```

  - **target** is generally the file, such as a binary or object file, that you want created.
  - **dependency** is a list of one or more files required as input in order to create target.
  - The **command**s are the steps, such as compiler invocations, necessary to create target.
  - Unless specified otherwise, make does all of its work in the current working directory.
- In addition to the obvious targets, there should always be a **clean** target. This target removes all the generated object files and programs so that you can start fresh. The rule for this target uses the **rm** command to remove the files. Here is what **Makefile** contains:

```
reciprocal: main.o reciprocal.o
        g++ $(CFLAGS) -o reciprocal main.o reciprocal.o
main.o: main.c reciprocal.hpp
        gcc $(CFLAGS) -c main.c
reciprocal.o: reciprocal.cpp reciprocal.hpp
```

```
        g++ $(CFLAGS) -c reciprocal.cpp
clean:
        rm -f *.o reciprocal
```

This makefile has four rules.

- The first target, **reciprocal**, is called the default target, this is the file that make tries to create. **reciprocal** has two dependencies, **main.o** and **reciprocal.o**; these two files must exist in order to build editor.
- Following line is the command that make will execute to create editor.This command builds an executable named **reciprocal** from the two object files.
- The next two rules tell make how to build the individual object files.

- The line with the rule on it must start with a Tab character, or **make** will get confused.
- The $(CFLAGS) is a **make** variable.You can define this variable either in the **Makefile** itself or on the command line. GNU make will substitute the value of the variable when it executes the rule. So, for example, to recompile with optimization enabled, you would do this:

```
% make clean
rm -f *.o reciprocal
% make CFLAGS=-O2
```

- How does make know when to rebuild a file?
  - If a specified target does not exist in a place where make can find it, make (re)builds it.
  - If the target does exist, make compares the timestamp on the target to the timestamp of the dependencies.
  - If one or more of the dependencies is newer than the target, make rebuilds the target, assuming that the newer dependency implies some code change that must be incorporated into the target.

1.3 Debugging with GNU Debugger (GDB)

- The debugger is the program that you use to figure out why your program is not behaving the way you think it should. You can use GDB to step through your code, set breakpoints, and examine the value of local variables.
- The *-g* option can be qualified with a 1, 2, or 3 to specify how much debugging information to include.
  - The default level is 2 (*-g2*), which includes extensive symbol tables, line numbers, and information about local and external variables.
  - Level 3 debugging information includes all of the level 2 information and all of the macro definitions present.
  - Level 1 generates just enough information to create backtracks and stack dumps. It does not generate debugging information for local variables or line numbers.
- Additional debugging options include the *-p* and *-pg* options, which embed profiling information into the binary.
  - This information is useful for tracking down performance bottlenecks in your code.
  - *-p* adds profiling symbols that the prof program can read,
  - *-pg* adds symbols that the GNU project's prof incarnation, **gprof**, can interpret.
  - The *-a* option generates counts of how many times blocks of code (such as functions) are entered.
  - *-save-temps* saves the intermediate files, such as the object and assembler files, generated during compilation.

1.3.1 Compiling with Debugging Information

To use GDB, you will have to compile with debugging information enabled. Do this by adding the **-g** switch on the compilation command line. If you are using a **Makefile** as described previously, you can just set **CFLAGS** equal to **-g** when you run make, as shown here:

```
% make CFLAGS=-g
```

When you compile with **-g**, the compiler includes extra information in the object files and executables. The debugger uses this information to figure out which addresses correspond to which lines in which source files, how to print out local variables, and so forth.

1.3.2 Running GDB

You can start up gdb by typing:

```
% gdb reciprocal
```

When **gdb** starts up, you should see the GDB prompt:

```
(gdb)
```

The first step is to run your program inside the debugger. Just enter the command **run** and any program arguments.

```
(gdb) run
Starting program: reciprocal

Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287 strtol.c: No such file or directory.
(gdb)
```

The problem is that there is no error–checking code in **main**. The program expects one argument, but in this case the program was run with no arguments. The **SIGSEGV** message indicates a program crash. GDB knows that the actual crash happened in a function called __strtol_internal. That function is in the standard library, and the source is not installed, which explains the "No such file or directory" message.You can see the stack by using the where command:

```
(gdb) where
#0 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
#1 0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

You can see from this display that **main** called the **atoi** function with a **NULL** pointer, which is the source of the trouble. You can go up two levels in the stack until you reach **main** by using the up command:

```
(gdb) up 2
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8 i = atoi (argv[1]);
```

You can view the value of variables using the **print** command:

```
(gdb) print argv[1]
$2 = 0x0
```

That confirms that the problem is indeed a **NULL** pointer passed into **atoi**. You can set a breakpoint by using the **break** command:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

This command sets a breakpoint on the first line of **main**. Now try rerunning the program with an argument, like this:

```
(gdb) run 7
Starting program: reciprocal 7

Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8 i = atoi (argv[1]);
```

You can see that the debugger has stopped at the breakpoint. You can step over the call to **atoi** using the **next** command:

```
(gdb) next
9 printf ("The reciprocal of %d is %g\n", i, reciprocal (i));
```

If you want to see what is going on inside reciprocal, use the **step** command like this:

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6 assert (i != 0);
```

You are now in the body of the reciprocal function.

1.4 Finding More Information

1.4.1 Man Pages

Linux distributions include man pages for most *standard commands*, *system calls*, and *standard library functions*.The **man** pages are divided into numbered subsections; for programmers, the most important are these:

```
(1) User commands
(2) System calls
(3) Standard library functions
(8) System/administrative commands
```

The numbers denote **man** page subsections. Linux's man pages come installed on your system; use the **man** command to access them. To look up a *man* page, simply invoke ***man name***, where ***name*** is a command or function name.

```
% man sleep
% man 3 sleep
```

If you're not sure which command or function you want, you can perform a keyword search on the summary lines, using ***man -k keyword***.

1.4.2 Header Files

You can learn a lot about the system functions that are available and how to use them by looking at the system header files. These reside in **/usr/include** and **/usr/include/sys**. These header files make good reading for inquiring minds. Don't include them directly in your programs, though; always use the header files in **/usr/include** or as mentioned in the man page for the function you're using. The source code for the Linux kernel itself is usually stored under **/usr/src/linux**.