

An Introduction to Software Development under GNU/Linux

by Amr Ramadan, June 2006

Moving to GNU/Linux

When I first moved to GNU/Linux from Microsoft Windows, the transition was a little bit hard because I was not very much expecting that the fundamentals of how I work on this new operating system would differ. I was expecting that there would be a learning curve, but like in learning how to use Borland C++ after using Microsoft Visual Studio. One of the first steps I made was to look for an IDE to replace Visual Studio. I didn't find any that were a perfect replacement. And that was when I realized that moving to GNU/Linux would not be just about changing an operating system on top of which applications run, but rather, it will be about changing the fundamentals of how I work.

In this tutorial, I will guide you through your transition to developing C++ applications under GNU/Linux. This will go from choosing the toolchain, to using it to compile and debug applications and libraries. Interestingly, not only this information applies to C, but it even applies to other languages like Pascal and Objective-C. This is true because, for example, the compiler and debugger suite we would be using, can both be used to compile and debug applications in several languages, and hence, what you'd learn once for C++ can be applied on other languages, if the need aroused.

Toolchain

The GNU Compiler Collection (gcc) is the standard mean for compiling applications under GNU/Linux. gcc is used to compile the Linux kernel, and most of the open-source applications you'd come across while using a Linux-based distribution. The standard release of gcc supports compiling applications written in Ada, C, C++, Fortran, Java, Objective-C and Objective-C++, and additional frontends can be installed for compiling applications in other languages. gcc can target several computer architectures including x86, IA-64, MIPS and PowerPC.

Along with gcc, the GNU Debugger (gdb) is the standard debugger you'd be using under GNU/Linux. gdb, being a command-line debugger, will certainly look counter-productive once you come across it. There are no menus to go through or buttons to use to add watches, all you would have is a console through which you would type commands that you were accustomed to using your mouse to do them in the days of the Visual Studio Debugger. However, and before you'd run away or run to find a GUI frontend for gdb, it is quite noteworthy to mention that getting used to gdb is something that would certainly pay off.

Picking an IDE should not be a target at this early point. I would not get into why is that now, but by the end of this tutorial, we would open up this point again, and it will be clear why sticking to command-line tools is much more convenient.

Developing Your First Application.

Our first application would be a simple "Hello, world!". The purpose of this simple application is to give you a quick look on how to use gcc and gdb, and how to use their manual pages.

First, open a terminal (command-line) window, type `"vi hello.cpp"` (without quotes), and hit enter. This will start the `vi` program, which we will use for writing and editing source code. Once open, you will find at the bottom of the screen text that tells you that `vi` is editing a "New File" called

"hello.cpp". Since we did not change the directory when we opened the console, the file will be created in your home directory.

To start writing the source code of the application, hit "i" in your keyboard to go to the Insert mode and type the code. Once done, hit the "Escape" key in your keyboard, and write ":wq" (without quotes) and hit the "Enter" key, to ask vi to write the changes and quit the editor. A colon is used to specify a command to vi. If we were just to write ":w", then vi would have saved the changes to the file without quitting, thus allowing us to continue editing.

```
/*
 * hello.cpp
 */

#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Now that we have written our first application, it is time to compile it. We use g++ to first generate the machine code of our application, and then use g++ to link this machine code with the system libraries into an executable binary. We will then see how to merge those two steps into one. g++ is the frontend of the GNU Compiler Collection used to compile C++ code.

First, we compile "hello.cpp" into an object file. The following command generates a "hello.o" file.

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

Then, we link this object file with the system libraries to generate an executable binary. The following command generates a "hello" output file.

```
g++ hello.o -o hello
```

To run this program, type:

```
./hello
```

The initial dot means the current directory. If you were to type "hello" only and hit enter, then the console would look for a binary called "hello" in the paths defined in the PATH environment variable. Using the dot, we ask for executing the binary program "hello" which resides in this current directory. An equivalent way would be to type the full path to the binary file.

The above two steps we used to compile the application could have been made into one, as follows:

```
g++ hello.cpp -o hello
```

g++ has a myriad of options which you can use to specify such things as including symbolic information for debugging, setting the warning and error levels, setting the optimization options, specifying paths in which the compiler would look for header files, libraries, and many other options. The Manual Pages is your first reference for help on any application. You get the Manual Pages for g++ by:

```
man g++
```

Some of these options are:

```
-g      Produce debugging information in the operating system's native format.
-Wall   Turns on all optional warnings.
-l      Search the library named library when linking.
-L      Add directory dir to the list of directories to be searched for -l.
-I      Add the directory dir to be searched for header files.
```

When viewing the manual pages, you would at least use the "Up", "Down", "Page Up" and "Page Down" keys to move through the pages, "/" to search, and "q" to quit.

We will now use "gdb" to debug our little application. To start debugging, run gdb passing to it the name of the binary which must have been created with the "-g" switch shown above.

```
gdb hello
```

This will start gdb setting the file to debug being "hello". Add a breakpoint to the main function by using the "break" command, or its short version "b".

```
b main
```

Now run the application by using the "run" command, or its short version "r". gdb will now start executing the program and will stop at the breakpoint we set at the main function. Type "next" or "n" (without quotes) to make a single step through the code, and you will see "Hello, world!" printed. Make another single step and the program will exit. Quit gdb by issuing the "quit" or "q" command.

Besides single stepping, you can of course step into function calls using "step" or "s", continue running the program after stopping with "continue" or "c", display the program stack with "backtrace" or "bt", and print the value of a variable with "print" or "p".

Consult the manual pages of gdb (using "man gdb") for additional information on gdb.

Dividing Your Application into Multiple Source Files

Let us split our little application into multiple source files. Simply, what we would do is to define a Person class in "person.h", implement it in "person.cpp", and use it in "main.cpp". Then, we would see how we could compile these multiple sources into one binary, and how to debug such application. Here is the source code of these files:

```
/*
 * person.h
 */

class Person
{
public:
    Person() {};
    ~Person() {};

    void Speak(char* sentence);
};
```

```

/*
 * person.cpp
 */

#include "person.h"
#include <iostream>

void Person::Speak(char* sentence)
{
    std::cout << sentence << std::endl;
}

```

```

/*
 * main.cpp
 */

#include "person.h"

int main()
{
    Person person;
    person.Speak("Hello, World!");

    return 0;
}

```

Like with our first application, we can either compile the source files together, or we can separately compile each source to an object file and then link the output files. The second method is advantageous, specially if we have a large number of files, because we can then only compile the changed files. This is important, because if we have 100 files, and we only changed 1, we practically do not need to compile the other 99 files. If we can only compile that one changed file, and then link it with the already compiled 99 files, then we could save a considerable amount of time.

Here is how we can compile the above program:

```

g++ -c person.cpp
g++ -c main.cpp
g++ person.o main.o -o program

```

This will create two object files, and link them into an executable named "program". Run the program just like before, prepending its name with "./":

```

./program

```

As we said before, the above method could speed up the building process of your project; when you change only a small set of files, only those files need to be recompiled. On the contrary, if we were to compile this program like this:

```

g++ main.cpp person.cpp -o program

```

Then each time we execute the above command, the compilation and linking process will be executed in its entirety, wasting possible valuable time.

Next we try to debug the application. Compile it with debug symbols:

```
g++ -g main.cpp person.cpp -o program
```

Then start gdb.

```
gdb program
```

Add a breakpoint.

```
break person.cpp:10
```

This adds a breakpoint at line 10 in person.cpp

Of the two methods of compiling several source files, the first is comparably too complicated. With a little over a couple of files, it will become almost impossible to keep track of which files were changed and which were not. This is where Makefiles come in.

Automating the Build Process with Makefiles

We will add another program to our toolchain: Make. Make is a program used to automate the process of generating a file (or a set files, called targets) from one or more files. Make takes care of compiling files only if their source code is newer than the already built object code for that file, or if one of the files they depend upon has been changed. To use Make, you create a file named "Makefile" in your source tree, and write in that file rules that describe *targets* and their *dependencies*. Let us go through a Makefile to compile the above application:

```
#
# Makefile
#
# Note: comments are prepended by a hash.
#
CXXFLAGS=-Wall

all: program

program: main.o person.o
    g++ main.o person.o -o program

main.o: main.cpp
    g++ $(CXXFLAGS) -c main.cpp

person.o: person.cpp
    g++ $(CXXFLAGS) -c person.cpp

clean:
    rm -rf *.o program
```

A simple Makefile consists of several targets, in the following form:

```
target: dependencies
TAB command
```

`target` is the name of a target. It is followed by a colon and a list of space-separated dependencies. The next line must be prepended with a TAB (and not several spaces), and then the command to execute for that target.

The first target we specified in the above example is "all", which depends on another target "program". The "all" target is the default target. When you run Make without specifying a target, it will look for the "all" target.

The "program" target depends on two dependencies, which means that it will not be executed until those two targets it depends on are. Each of those latter targets produce the object files, which are then used by the "program" target to create the final executable.

To build your application, in the folder in which you created the Makefile, type:

```
make
```

Now Make will read the "Makefile", and start creating the "all" target. For each of the targets, it will execute its command only if its dependencies have changed. For example, it will only execute the command "g++ -c main.cpp" if the file "main.cpp" is newer than the object file "main.o". This should take a huge weight off you; you only have to create the Makefile once and all you need to build your project is run the `make` program.

"CXXFLAGS" defines a variable with that name, and sets its value to "-Wall". We evaluate the value of that variable in the Makefile with `$(CXXFLAGS)`, and we can override this value from the command-line. To create a debug version of our program:

```
make CXXFLAGS=-g
```

This will override the defined value of "-Wall" in the Makefile with "-g".

If you tried to build your project with `make`, and then promptly tried the above command to build a debug version, you will notice that `make` outputs the following:

```
make: Nothing to be done for 'all'
```

This is because none of the source files have changed. Even though you want to create a different binary, all that `make` knows is that since no dependencies have changed, it should not generate another binary. To solve this issue, we have to clean the source tree from the files `Make` generated so that we could build the code again with the "-g" flag.

The last target, "clean", is used to clean the source tree from the files created by `make`. In its command, we recursively remove (the `-r` switch) all objects files and the built program, and also ignore all non-existent files and avoid confirmations to deleting (the `-f` switch). To clean your source tree, simply type while in it:

```
make clean
```

This executes only the "clean" target.

Makefiles definitely ease the development, but it also becomes difficult to think about the complexity of a Makefile for a project that consists of hundreds or thousands of source files, and how the dependencies between them will be managed. Luckily, we can use a set of tools collectively called Autotools to automatically generate the Makefile of our project.

Note: A handy script used when developing cross-platform applications is a "configure" script. This script, when run, reads rules from a "Makefile.in" and then creates from it a platform-specific "Makefile" after identifying the characteristics of the platform it is run from. We will get to see configure scripts next.

Automation with Autotools

We now want to automate the creation of two files: a "configure" file which will test the platform it is run from, and a "Makefile", which will ultimately build our project for a specific platform. To make the word "platform" less ambiguous, consider that you shipped your source code to someone whose default compiler is not g++. What you want is to generate a Makefile that can be used to compile on that platform, and this is the role of "configure".

We will use three applications: `aclocal`, `autoconf` and `automake`. `aclocal` must be run first to create some files needed by the Autotools. `autoconf` will be used to generate a "configure" file from a "configure.ac" file. `automake` will be used to create a "Makefile.in" from a "Makefile.am". Next, when you run `configure`, it will create a "Makefile" from the "Makefile.in" generated by `automake`. Confused? Here it is again.

```
WE HAVE TWO FILE: "configure.ac" AND "Makefile.am".
RUN ACLOCAL => CREATES "aclocal.m4" FILE (AND OTHERS)
RUN AUTOCONF => CREATES "configure" FILE FROM "configure.ac" FILE.
RUN AUTOMAKE => CREATES "Makefile.in" FILE FROM "Makefile.am".
NOW WE HAVE TWO ADDITIONAL FILES: "configure" AND "Makefile.in".
RUN "configure" => CREATES "Makefile" FROM "Makefile.in".
RUN MAKE => BUILDS THE PROJECT.
THAT'S IT!
```

Here are the source of the said files to build the above project:

```
#
# configure.ac
#

AC_INIT(main.cpp)
AM_INIT_AUTOMAKE(hello_world,1)
AC_PROG_CXX
AC_PROG_INSTALL
AC_OUTPUT(Makefile)
```

```
#
# Makefile.am
#

AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = program
program_SOURCES = main.cpp person.cpp
```

In "configure.ac", we specify a set of macros which will then be expanded by autoconf into the "configure" file. AC_INIT takes as argument the name of a file in the source tree, and when the "configure" script is run, it will check first for the existence of this file. AC_INIT_AUTOMAKE takes the program name and a version number as arguments, and is expanded into several standard checks. AC_PROG_CXX indicates that the source is in C++ (AC_PROG_CC indicates it is in C). AC_PROG_INSTALL adds an "install" target, which when run, will install the program to the default installation directory of the system. Finally, the AC_OUTPUT macro is used to generate a "Makefile".

Then in "Makefile.am", we specify a set of options that will be used by automake to create "Makefile.in". The first variable specifies that we are building a non-GNU project. If we were building a GNU project (the default), then we should create the files NEWS, README, AUTHORS and ChangeLog; the following will suffice:

```
touch NEWS README AUTHORS ChangeLog
```

touch will create four empty files with the above names. The next variable, bin_PROGRAMS, specifies that the name of the target binary is "program". program_SOURCES contains an *unordered* list of the project's source files (no need to add the header files). Note that the last variable's name is dependent on the value of bin_PROGRAMS. If for example, bin_PROGRAMS was set to "myprogram", then the last variable's name should have been myprogram_SOURCES instead of program_SOURCES.

To build your application, run the following commands while you are in the source tree:

```
aclocal
autoconf
automake --add-missing
./configure
make
```

And that is it. When you add a new source file to your project, you just add it in the Makefile.am, and then run the above set of commands. Other than that, while you're developing, all you need to do to build the project is to run the last command, make.

Note: Automake requires certain common files to exist in certain situations. The add-missing switch instructs automake to generate these files if they are not there.

To install the application, switch to the root user using the command "su" (and typing the root password when asked for it), then:

```
make install
```

This copies the program to the default programs directory in your system (for example, /usr/local/bin).

It should now be obvious how Autotools provide a very luxurious mean to automating the build process of your applications, by distancing you from having to define the dependencies between your source files.

Static Libraries

Static Libraries (or Archives) in GNU/Linux are the equivalent of Windows' .lib files. They are resolved at compile-time by the linker, and have the .a extension. To create a static library, we use the `ar` program to add one or more object files into one archive file. The following creates a Static Library for the Person class we created above:

```
ar rc libperson.a person.o
```

Note that you must create the object file of the Person class first. The parameter "rc" to the `ar` command is to add the file "person.o" to the archive (the `r` switch) and to create an archive named "libperson.a" (the `c` switch). Now that we created the archive, we can link it with main.o as follows:

```
g++ -o program main.o -L. -lperson
```

Now let us go through the above parameters one by one. The first switch specifies the name of the target binary, just as before. The `-L` switch, followed by a dot, specified an additional path to look for libraries is: the current directory. The parameter `-lperson` specifies a library name to link against; the linker will substitute the "-l" with "lib" and add a ".a" prefix, thus producing a name "libperson.a".

Debugging a program that was compiled with a static library is no different from debugging it if it was linked against the object file directly. Of course, the object file used to create the static library must have been compiled with `-g` switch to add the debugging symbols.

Shared Libraries

Shared Libraries, (or Shared Objected) in GNU/Linux are the equivalent of Windows' .dll files. They are dynamically linked at run time but statically aware, and have the .so extension. Although the libraries must be available during building, they are not included into the built executable. Here is how we build a Shared Library out of the Person class:

```
g++ -fPIC -c person.cpp
g++ -shared -o libperson.so person.o
g++ -o program main.o -L. -lperson
```

The `-fPIC` option is to tell the compiler to create Position Independent Code (create libraries using relative addresses rather than absolute addresses because these libraries can be loaded multiple times). The `-shared` option is to specify that an architecture-dependent shared library is being created.

To use a shared library, it must be placed in a directory specified in the `LD_LIBRARY_PATH`

environment variable, or in your system's default library path (for example, `/usr/local/lib`).

Run the program as usual:

```
./program
```

Most probably, and since the "current directory" is not in your PATH environment variable, you will get an error trying to run the program like this:

```
./program: error while loading shared libraries: libperson.so: cannot open shared object file: No such file or directory
```

Let us see why, type:

```
ldd program
```

Ldd is a program that prints the shared library dependencies of some executable. A sample output would be:

```
linux-gate.so.1 => (0xffffe000)
libperson.so => not found
libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0xb7e8b000)
libm.so.6 => /lib/tls/i686/cmov/libm.so.6 (0xb7e69000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0xb7e5f000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7d30000)
/lib/ld-linux.so.2 (0xb7f74000)
```

Only our "libperson.so" could not be found. We can copy it to the default library path, or more easily:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

This adds the current directory to the LD_LIBRARY_PATH variable for this console session. Now you can run your program.

Debugging a program using Shared Libraries is similar to using Static Libraries. When running the program in gdb, all the Shared Libraries it depends upon are loaded. For some versions of gdb, there could be an issue in setting breakpoints inside Shared Libraries, if that was the case with you, you can preload the Shared Library before starting gdb by adding its path to the environment variable

LD_PRELOAD:

```
export LD_PRELOAD=$LD_PRELOAD:./libperson.so
```

Note that using the dot in setting the environment variables like we did with LD_LIBRARY_PATH and LD_PRELOAD means that you would have to reside in your project's directory, because the dot means the current directory. If your source tree is in `/home/devel/project/`, and your current directory in the console was `/tmp`, then if you executed `gdb /home/devel/project/program`, it will look for the shared object as `/tmp/libperson.so`. This inconvenience can be solved by adding the full path to the environment variable rather than the dot.

Closing Remarks

Even though using command-line tools may seem counter-productive, its advantages certainly pay off. The problem is, as of this writing, you wouldn't find an IDE that abstracts the build process as Visual Studio does. Some, like Eclipse (with its CDT - C/C++ Development Tooling - plugin), have a project mode called "Managed Makefiles". In this type of project, Eclipse creates the Makefiles for you and you literally do not do anything except writing your code.

However, being Managed, you will certainly come to the point where you need control on how your project is built, and then you would find you cannot do so. And if you ship your code to someone, it is quite difficult to ask him to install Eclipse in order that he can compile your code. Even more, Eclipse is not a fully integrated development environment in the way Visual Studio is - you cannot use it in building GUI applications.

Another advantage for getting used to developing within the console is when you need to debug an application on a remote computer. Most, if not all, Linux-based servers are not configured with a Desktop Environment, so you would not be able to administer them in a fashion similar to using Windows' Remote Desktop Connection. Secure Shell, or SSH, is one of the most popular means to remotely administer a Linux server. Using the `ssh` program, you open a secure connection to a server, you're given a console on that server, and then, you can use it just like you were running a console on your machine. You can run `gcc`, `g++`, `gdb`, or whatever console applications you want. To connect to a machine running an SSH server, type:

```
ssh root@hostname.com
```

or

```
ssh root@89.29.233.42
```

You will then be asked for a password, and then you will be given a shell on that machine.

References

- The GNU C Library - http://www.gnu.org/software/libc/manual/html_node/index.html
- Using the GNU Compiler Collection - <http://gcc.gnu.org/onlinedocs/>
- GDB User Manual - http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html