

# Compiling, Makefiles, and Debugging

## 1 Compiling Programs

The standard Unix/Linux compilers for C/C++ are distributed as part of the GNU project (see [www.gnu.org](http://www.gnu.org)). The C++ compiler is called `g++`, and the C compiler is called `gcc`. In addition, most systems have a (hopefully recent) version of Sun's Java Platform installed (see [java.sun.com](http://java.sun.com)). The Java compiler is called `javac`, and the Java interpreter is called `java`. If you don't know all the details about constructing and executing a C++, C, or Java program, don't worry — you can learn much more about `g++`, `gcc`, and `javac` using the man pages. For now we'll just look at the basics.

### 1.1 Compiling C/C++ Programs

To compile a C++ program named `sort.cc` from the shell, at the shell prompt in the terminal window type

```
g++ -o sort sort.cc
```

(Note that `.cc` is the most common filename extension used for C++ programs. You may see `.CC`, `.cpp`, or `.CPP` as well.) Similarly, to compile a C program named `sort.c`, type

```
gcc -o sort sort.c
```

The `-o` flag gives the name of the executable to create; if the option is omitted, the executable will automatically be named `a.out`.

In general, you'll want to compile your programs with more options specified. The following is a better way to compile the program.

```
g++ -g -Wall -o sort sort.cc
```

The `-g` flag produces debugging information in the executable for use with a debugger such as DDD (see Section 3 of this document). The `-Wall` option instructs the compiler to display “all warnings”. Take these warnings seriously, for they may point to a looming problem.

To compile multiple source files into a single executable, simply include them all in the same compile command. For example, to compile `sort.cc` and `swap.cc` into a single executable called `sort`, issue the command

```
g++ -g -Wall -o sort sort.cc swap.cc
```

Be careful when using the `-o` flag in a multiple source file compilation, however — you can easily and inadvertently destroy a source file. Consider the following.

```
g++ -o sort.cc swap.cc
```

In this example, the name of the executable has been mistakenly omitted. As a result, `g++` will assume that the executable should be named `sort.cc` and therefore will overwrite (destroy) the contents of `sort.cc`!

Once you have a working executable, go to a terminal window to execute it. Make sure you are in the correct directory and then type the name of the executable program.

## 1.2 Compiling Java Programs

To compile a Java program named `Sort.java`, type

```
javac Sort.java
```

Note that, unlike `g++` and `gcc`, `javac` does not create an executable. Instead, a file containing byte code is created with a `.class` filename extension. For example, successful completion of the `javac` command above would create a separate file named `Sort.class`.

If your Java source program depends on other local Java source programs, you only need to compile the main source program. You do not need to issue separate compile commands for each source program — `javac` will recognize everything that needs compiling. For example, if you have programs `Sort.java` and `Swap.java`, you only need to compile `Sort.java`. The `javac` compiler will (assuming your code is bug-free) compile and create `Sort.class` and `Swap.class` automatically.

To invoke the interpreter for the Java Virtual Machine (JVM), i.e., to execute the program, go to a terminal window and use the `java` command as in the following example.

```
java Sort
```

Note that you do not include any filename extension, nor do you include any other programs on which the program depends.

## 1.3 Compiling from within Emacs

If your editor of choice is Emacs, you can also compile programs by selecting the **Compile...** option from the **Tools** menu. (Alternatively, you can type `M-x compile`.) A compile command will appear in a minibuffer at the bottom of the window — you may need to edit the command to reflect the type of program you are compiling. To change the default command, look under the **Help** menu for customizing options.

For example, to compile the file `sort.cc`, you can type

```
g++ -g -Wall -o sort sort.cc
```

in the minibuffer and hit enter. Notice that the window splits and any output from the compiler is displayed in the bottom half. If there are errors in your code, you can click on the error message

with the middle mouse button to go directly to that line in the source file. If you want to hide the messages that appear in the lower half of the window, click in the top half and type `C-x 1`. (Alternatively, you can hide the messages by selecting the `Unsplit` (or `One Window`) option from the `Files` menu.)

If you are a `vi` or `vim` user (if not, why not?), you enjoy using the shell anyway, so just compile from there as well!

## 2 Makefiles

When creating large software projects, it is good practice to divide the code into multiple source files. One of the main benefits is that errors are generally easier to find and eliminate. When you recompile the program, you only need to recompile the source file that you changed and those that depend upon it. Obviously, the task of determining which source files to recompile can quickly become overwhelming as the size of the project grows.

Unix/Linux has a powerful utility called `make` that allows you to efficiently compile and recompile a large, complex project. The `make` utility reads a special file that you construct known as a makefile. The makefile describes the files involved in the project and the dependencies among them. Each group of lines in a makefile has the following form.

```
targets: dependencies
<Tab> commands
```

In this context, `targets` is a list of target files separated by spaces, `dependencies` is a list of files on which the targets depend, and `commands` is a list of commands used to (re)create the targets. In other words, if any of the files in a dependency list is modified, `make` will recompile and/or relink the target. (NOTE: the `<Tab>` preceding `commands` must be a tab character — **not** spaces — or the `make` command will fail.) For an executable target, the dependency files are the object files that must be linked together to form the executable. For one of the object file targets, the dependency files are the source file(s) from which the object file will be compiled, including any header files (with `.h` extension) included in the source file(s).

Consider the following simple makefile.

```
sort: sort.o swap.o
    g++ -g -Wall sort.o swap.o -o sort

sort.o: sort.cc sort.h
    g++ -g -Wall -c sort.cc

swap.o: swap.cc swap.h
    g++ -g -Wall -c list.cc
```

This makefile describes the file dependencies for our simple `sort` compilation. The first two lines inform `make` that the executable `sort` depends on the object files `sort.o` and `swap.o`. If either of

the object files changes, the associated command `g++ -g -Wall sort.o swap.o -o sort` will be used to (re)create (i.e., link) `sort` from the object files. The next four lines tell `make` how to create the object files. For example, the object file `sort.o` should be (re)created (i.e., compiled) using the command `g++ -g -Wall -c sort.cc` if either of the dependency files `sort.cc` or `sort.h` is modified. (The `-c` option to `g++` tells the compiler to compile the source into an object file, but to not perform any linking — an executable will not be created.) So you can see that if, for example, `swap.cc` is modified, both `swap.o` and `swap` will be recreated, but `sort.o` will not.

To use such a makefile, it should be placed in the same directory as the source files and named `Makefile` (or `makefile`). Then simply execute

```
make sort
```

to compile and link all necessary files.

Now consider the following, more complicated example of a makefile.

```
# Makefile for compiling a network sender project

CC = g++
LD = g++

INCDIR = -I../
CFLAGS = -g -Wall -c $(INCDIR)
LDFLAGS = -g -Wall

all: sender

sender: sender.o packet.o ack.o
    $(LD) $(LDFLAGS) $^ -o $@

sender.o: sender.cc packet.cc ack.cc packet.h ack.h
    $(CC) $(CFLAGS) $<

packet.o: packet.cc ack.cc packet.h ack.h
    $(CC) $(CFLAGS) $<

ack.o: ack.cc ack.h
    $(CC) $(CFLAGS) $<

clean:
    /bin/rm -f sender *.o core*
```

In this makefile, you can see the use of *macros* like `CC` (the name of the compiler) and `LD` (the name of the linker). Macros simplify a makefile and make modification easier. For example, if you want to change the compiler of choice, you can easily change the `CC` macro (in only one spot!) instead of having to change each of the command lists in the makefile. Furthermore, in this example the

macros `INCDIR`, `CFLAGS`, and `LDFLAGS` each provide additional options. `INCDIR` gives an additional directory (the parent directory) in which the compiler will look for header files; `CFLAGS` gives the familiar compiler options; and `LDFLAGS` gives linker options.

There are several special variables you can use to make your makefile easier to modify. First, the variable `$@` in a command refers to the corresponding target. For example, in the command creating the `sender` target in the makefile above, `$@` refers to `sender`. Second, the variable `$^` refers to all files in the corresponding dependency list. For example, in the same first command, `$^` refers to the entire list of dependency files `sender.o`, `packet.o`, and `ack.o`. Finally, the variable `$<` refers to only the first file in the dependency list. For example, in the command creating the `sender.o` target, `$<` refers only to `sender.cc`. Appreciate that use of such variables reduces the amount of work required should future changes to your makefile be necessary.

By default, `make` will update the target object that appears first in the file, i.e., if you issue the command

```
make
```

with no arguments. It is common practice to include as the first target `all`, followed by the target(s) you want to create if no target is specified on the command line.

In addition, the last target in the makefile (`clean`) demonstrates that compilation and/or linking need not be the result of using the `make` utility. In this example, the `clean` target specifies to `make` that the executable, all object files, and any core files should be removed (the `-f` flag tells `rm` to ignore nonexistent files and to never prompt). You can issue the command

```
make clean
```

anytime you want to clean up expendable files.

More information about the `make` utility is available via the man pages.

### 3 Debugging Programs

When debugging small programs, it is often the case that judicious use of print statements are sufficient to help you determine where any errors reside in your code. For complex programs or projects, however, print statements quickly become unmanageable. For such cases, an interactive debugging tool is the weapon of choice.

The standard Unix/Linux debugging tool for C/C++ programs is called GDB, the GNU DeBugger. GDB is a command line debugger — you can step through your program as it executes, add breakpoints, display variable values, etc. by typing appropriate commands at a prompt. The Java Platform provides its own command line debugger for Java programs, known as JDB (Java DeBugger). Either of these debuggers can be run from the command line by issuing either the command `gdb` or `jdb` respectively.

Generally speaking, it is preferable and more convenient to use a debugger with a GUI (Graphical User Interface). Perhaps the most popular GUI for Unix/Linux debugging is called DDD (Data

Display Debugger). DDD is a sophisticated interface that runs on top of your favorite debugger — e.g., GDB or JDB — and incorporates an interactive graphical data display by which data structures can be represented visually.

You can start DDD by typing `ddd` followed by the name of your executable program. By default, DDD will use GDB; to use JDB, you must type `ddd --jdb` followed by the name of your Java program. Your program need have been compiled with the `-g` option to include debugging information suitable for DDD (or, more correctly, GDB or JDB) to display. To simply run your program within DDD, press the **Run** button in the small floating tool window that appears. (If you need to supply command line arguments to your program, select the **Run** option from the **Program** menu.) The output of your program will be displayed in the window below your source code. An example screenshot of the DDD GUI is shown in Figure 1.

Usually when you are debugging a program, you want the program to temporarily stop execution at some point so that you can examine the value of one or more variables, or so that you can check some other part of your program. To stop your program, you must set a *breakpoint*. To do so, find the line of your source code where you want to pause and click to the left of the line with the left mouse button. Then press the **Break** button (with a stop sign) below the menu bar. A stop sign icon should appear to the left of the line. Now when you run your program, execution will stop every time this line is reached. You can set as many breakpoints as you need. To clear a breakpoint that has already been set, click to the left of the desired line, and then click the **Clear** button (with a dimmed stop sign) below the menu bar.

If you want to have your program stop every time the value of a variable of interest changes, you can set a *watchpoint*. To set a watchpoint, left-click on a variable name in the source code window and press the **Watch** button below the menu bar.

Once your program has stopped, there are two options for stepping through your program one line at a time — **Step** and **Next**. If you choose **Step**, you will proceed to the next line of source code in your program, even if that means entering a different function or method. If you choose **Next**, you will advance to the next line of source code, but function (method) calls will be treated as a single instruction. If you want execution to continue normally without stepping one line at a time, click the **Cont** button on the floating tool window. Execution will proceed until the next breakpoint is encountered or until the program terminates, whichever occurs first.

To view the value of a variable as your program runs, left-click on the variable name in the source code window and then press the **Display** button below the menu bar. A box with that variable's name and value will appear in a new window above your source code.

These are just the basics of DDD. The display function of DDD, although described *very* briefly here, is quite sophisticated. Under the **Help** menu, you can find context sensitive help and a detailed manual. Also notice that if you linger over a button with the mouse pointer, balloon help will appear to assist you. Even more information can be found at [www.gnu.org/software/ddd](http://www.gnu.org/software/ddd).

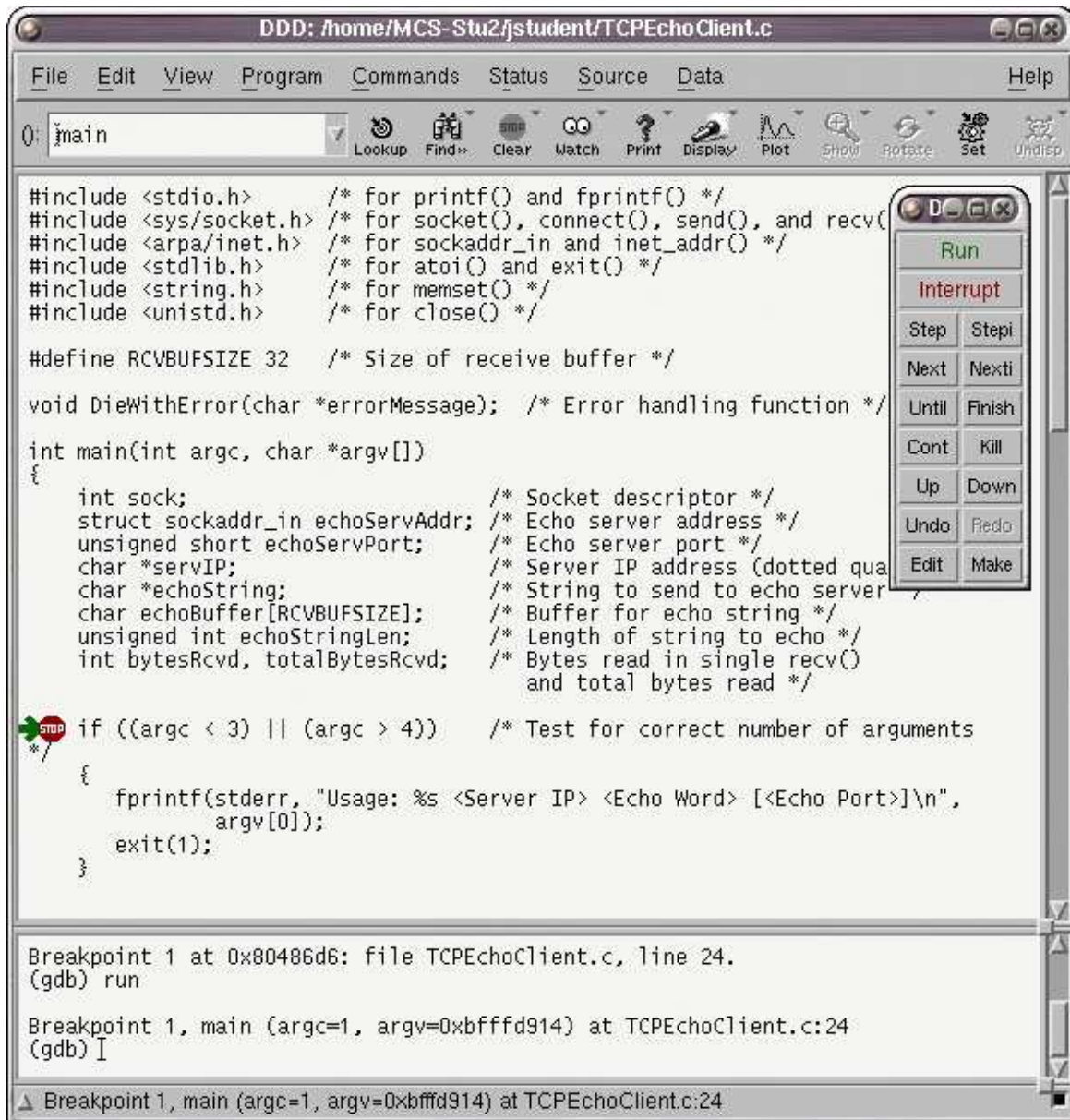


Figure 1: The DDD Gui Interface