UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

**CS61B** **P. N. Hilfinger**
**Fall 1999**

# Basic Compilation: javac, gcc, g++, and gmake

## 1 Compiling Java Programs

[The discussion in this section applies to Java 1.2 tools from Sun Microsystems. Tools from other manufacturers and earlier tools from Sun differ in various details.]

Programming languages do not exist in a vacuum; any actual programming done in any language one does within a *programming environment* that comprises the various programs, libraries, editors, debuggers, and other tools needed to actually convert program text into action.

The Scheme environment that you used in CS61A was particularly simple. It provided a component called the *reader,* which read in Scheme-program text from files or command lines and converted it into internal Scheme data structures. Then a component called the *interpreter* operated on these translated programs or statements, performing the actions they denoted. You probably weren't much aware of the reader; it doesn't amount to much because of Scheme's very simple syntax.

Java's more complex syntax and its static type structure (as discussed in lecture) require that you be a bit more aware of the reader—or *compiler,* as it is called in the context of Java and most other "production" programming languages. The Java compiler supplied by Sun Microsystems is a program called `javac` on our systems. You first prepare programs in files (called *source files*) using any appropriate text editor (Emacs, for example), giving them names that end in '`.java`'. Next you compile them with the java compiler to create new, translated files, called *class files*, one for each class, with names ending in '`.class`'. Once programs are translated into class files, there is a variety of tools for actually executing them, including Sun's java interpreter (called '`java`' on our systems), and interpreters built into products such as Netscape or Internet Explorer. The same class file format works (or is *supposed* to) on all of these.

In the simplest case, if the class containing your main program or applet is called $C$, then you should store it in a file called $C$`.java`, and you can compile it with the command

```
javac C.java
```

This will produce `.class` files for $C$ and for any other classes that had to be compiled because they were mentioned (directly or indirectly) in class $C$. For homework problems, this is often all you need to know, and you can stop reading. However, things rapidly get complicated when a program consists of multiple classes, especially when they occur in multiple packages. In this document, we'll try to deal with the more straightforward of these complications.

## 1.1 Where 'java' and 'javac' find classes

Every Java class resides in a *package* (a collection of classes and subpackages). For example, the standard class `String` is actually `java.io.String`: the class named `String` that resides in the subpackage named `io` that resides in the outer-level package named `java`. You use a `package` declaration at the beginning of a `.java` source file to indicate what package it is supposed to be in. In the absence of such a declaration, the classes produced from the source file go into the *anonymous package,* which you can think of as holding all the outer-level packages (such as `java`).

**The Java interpreter.** When the `java` program (the interpreter) runs the main procedure in a class, and that main procedure uses some other classes, let's say `A` and `p.B`, the interpreter looks for files `A.class` and `B.class` in places that are dictated by things called *class paths.* Essentially, a class path is a list of directories and *archives* (see §1.4 below for information on archives). If the interpreter's class path contains, let's say, the directories $D_1$ and $D_2$, then upon encountering a mention of class `A`, `java` will look for a file named $D_1$`/A.class` or $D_2$`/A.class`. Upon encountering a mention of `p.B`, it will look for $D_1$`/p/B.class` or $D_2$`/p/B.class`.

The class path is cobbled together from several sources. All Sun's java tools automatically supply a *bootstrap class path,* containing the standard libraries and such stuff. If you take no other steps, the only other item on the class path will be the directory '.' (the current directory). Otherwise, if the environment variable `CLASSPATH` is set, it gets added to the bootstrap class path. Our standard class setup has '.' and the directory containing the `ucb` package (with our own special classes, lovingly concocted just for you). If you print its value with, for example,

```
echo $CLASSPATH
```

you'll see something like

```
.:/home/ff/cs61b/lib/java/classes
```

(the colon is used in place of comma (for some reason) to separate directory names). It is also possible to set the class path (overriding the CLASSPATH environment variable) for a single program execution with

```
java -classpath PATH ...
```

but I really don't recommend this.

**The Java compiler.** The compiler looks in the same places for `.class` files, but its life is more complicated, because it also has to find source files. By default, when it needs to find the definition of a class $A$, it looks for file $A$`.java` in the same directories it looks for $A$`.class`. This is the easiest case to deal with. If it does not find $A$`.class`, or if it does find $A$`.class` but notices that it is older (less recently modified) than the corresponding source file $A$`.java`, it will automatically (re)compile $A$`.java`. To use this default behavior, simply make sure that the current directory ('.') is in your class path (as it is in our default setup) and put the source for a class `A` (in the anonymous package) in `A.java` in the current directory, or for a class `p.B` in `p/B.java`, etc., using the commands

```
javac A.java
javac p/A.java
```

respectively, to compile them.

It is also possible to put source files, input class files, and output class files (i.e., those created by the compiler) in three different directories, if you really want to (I don't think we'll need this). See the `-sourcepath` and `-d` options in the on-line documentation for `javac`, if you are curious.

## 1.2 Multiple classes in one source file

In general, you should try to put a class named $A$ in a file named $A$`.java` (in the appropriate directory). For one thing, this makes it possible for the compiler to find the class's definition. On the other hand, although public classes must go into files named in this way, other classes don't really need to. If you have a non-public class that really is used *only* by class $A$, then you can put it, too, into $A$`.java`. The compiler will still generate a separate `.class` file for it.

## 1.3 Compiling multiple files

Java source files depend on each other; that is, the text of one file will refer to definitions in other files. As I said earlier, if you put these source files in the right places, the compiler often will automatically compile all that are needed even if it is only actually asked to compile one "root" class (the one containing the main program or main applet). However, it is possible for the compiler to get confused when (a) some `.java` files have *already* been compiled into `.class` files, and then (b) subsequently changed. *Sometimes* the compiler will recompile all the necessary files (that is, the ones whose source files have changed or that use classes whose source files have changed), but it is a bit dangerous to rely on this for the Sun compiler. You can ask `javac` to compile a whole bunch of files at once, simply by listing all of them:

```
javac A.java p/B.java C.java
```

and so on. Since this is tedious to write, it is best to rely on a make file to do it for you, as described below in §3.

### 1.4 Archive files

For the purposes of this class, it will be sufficient to have separate `.class` files in appropriate directories, as I have been describing. However in real life, when one's application consists of large numbers of `.class` files scattered throughout a bunch of directories, it becomes awkward to ship it elsewhere (say to someone attempting to run your Web applet remotely). Therefore, it is also possible to bundle together a bunch of `.class` files into a single file called a *Java archive* (or *jar file*). You can put the name of a jar file as one member of a class path (instead of a directory), and all its member classes will be available just as if they were unpacked into the previously described directory structure described in previous sections.

The utility program '`jar`', provided by Sun, can create or examine jar files. Typical usage: to form a jar file `stuff.jar` out of all the classes in package `myPackage`, plus the files `A.class and B.class`, use the command

```
jar cvf stuff.jar A.class B.class myPackage
```

This assumes that myPackage is a subdirectory containing just `.class` files in package `myPackage`. To use this bundle of classes, you might set your class path like this:

`setenv CLASSPATH .:stuff.jar:`*other directories and archives*

## 2 Compiling C and C++

`Gcc` is a publicly available optimizing compiler (translator) for C, C++, Ada 95, and Objective C that currently runs under various implementations of Unix (plus VMS as well as OS/2 and perhaps other PC systems) on a variety of processors too numerous to mention. You can find full documentation on-line under Emacs (use `C-h i` and select the "GCC" menu option). You don't need to know much about it for our purposes. This document is a brief summary.

### 2.1 Running the compiler

You can use `gcc` both to compile programs into object modules and to link these object modules together into a single program. It looks at the names of the files you give it to determine what language they are in and what to do with them. Files of the form *name*`.cc` (or *name*`.C`) are assumed to be C++ files and files matching *name*`.o` are assumed to be object (i.e., machine-language) files. For compiling C++ programs, you should use `g++`, which is basically an alias for `gcc` that automatically includes certain libraries that are used in C++, but not C.

To translate a C++ source file, $F$`.cc`, into a corresponding object file, $F$`.o`, use the command

`g++ -c` *compile-options* $F$`.cc`

To link one or more object files—$F_1$`.o`, $F_2$`.o`, …—produced from C++ files into a single executable file called $F$, use the command.

`g++ -o` $F$ *link-options* $F_1$`.o` $F_2$`.o ...` *libraries*

(The *options* and *libraries* clauses are described below.)

You can bunch these two steps—compilation and linking—into one with the following command.

```
g++ -o F compile-and-link-options F1.cc ... other-libraries
```

After linking has produced an executable file called $F$, it becomes, in effect, a new Unix command, which you can run with

```
./F  arguments
```

where *arguments* denotes any command-line arguments to the program.

## 2.2 Libraries

A *library* is a collection of object files that has been grouped together into a single file and indexed. When the linking command encounters a library in its list of object files to link, it looks to see if preceding object files contained calls to functions not yet defined that are defined in one of the library's object files. When it finds such a function, it then links in the appropriate object file from the library. One library gets added to the list of libraries automatically, and provides a number of standard functions common to C++ and C.

Libraries are usually designated with an argument of the form `-l`*library-name*. In particular, `-lm` denotes a library containing various mathematical routines (sine, cosine, arctan, square root, etc.) They must be listed *after* the object or source files that contain calls to their functions.

## 2.3 Options

The following compile- and link-options will be of particular interest to us.

**-c** (Compilation option)
Compile only. Produces `.o` files from source files without doing any linking.

**-D***name*=*value* (Compilation option)
In the program being compiled, define *name* as if there were a line

```
#define name  value
```

at the beginning of the program. The '`=`*value*' part may be left off, in which case *value* defaults to 1.

**-o** *file-name* (Link option, usually)
Use *file-name* as the name of the file produced by `g++` (usually, this is an executable file).

**-l***library-name* (Link option)
Link in the specified library. See above. (Link option).

**-g** (Compilation and link option)
Put debugging information for `gdb` into the object or executable file. Should be specified for *both* compilation and linking.

**-MM** (Compilation option)
Print the header files (other than standard headers) used by each source file in a format acceptable to `make`. Don't produce a `.o` file or an executable.

**-pg** (Compilation and link option)
Put profiling instructions for generating profiling information for `gprof` into the object or executable file. Should be specified for *both* compilation or linking. *Profiling* is the process of measuring how long various portions of your program take to execute. When you specify `-pg`, the resulting executable program, when run, will produce a file of statistics. A program called `gprof` will then produce a listing from that file telling how much time was spent executing each function.

**-Wall** (Compilation option)
Produce warning messages about a number of things that are legal but dubious. I strongly suggest that you *always* specify this and that you treat every warning as an error to be fixed.

## 3 The make utility

Even relatively small software systems can require rather involved, or at least tedious, sequences of instructions to translate them from source to executable forms. Furthermore, since translation takes time (more than it should) and systems generally come in separately-translatable parts, it is desirable to save time by updating only those portions whose source has changed since the last compilation. However, keeping track of and using such information is itself a tedious and error-prone task, if done by hand.

The UNIX `make` utility is a conceptually-simple and general solution to these problems. It accepts as input a description of the interdependencies of a set of source files and the commands necessary to compile them, known as a *makefile*; it examines the ages of the appropriate files; and it executes whatever commands are necessary, according to the description. For further convenience, it will supply certain standard actions and dependencies by default, making it unnecessary to state them explicitly.

There are numerous dialects of `make`, both among UNIX installations and (under other names) in programming environments for personal computers. In this course, I will use a version known as `gmake`[1]. Though conceptually simple, the `make` utility has accreted features with age and use, and is rather imposing in the glory of its full definition. This document describes only the simple use of `gmake`.

[1]For "GNU `make`," GNU being an acronym for "GNU's Not Unix." `gmake` is "copylefted" (it has a license that *requires* free use of any product containing it). It is also more powerful than the standard `make` utility.

## 3.1 Basic Operation and Syntax

The following is a sample makefile[2] for compiling a simple editor program, `edit`, from eight `.cc` files and three header (`.h`) files.

```
# Makefile for simple editor

edit : edit.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o
        g++ -g -o edit edit.o kbd.o commands.o display.o \
                   insert.o search.o files.o utils.o -lg++

edit.o : edit.cc defs.h
        g++ -g -c -Wall edit.cc
kbd.o : kbd.cc defs.h command.h
        g++ -g -c -Wall kbd.cc
commands.o : command.cc defs.h command.h
        g++ -g -c -Wall commands.cc
display.o : display.cc defs.h buffer.h
        g++ -g -c -Wall display.cc
insert.o : insert.cc defs.h buffer.h
        g++ -g -c -Wall insert.cc
search.o : search.cc defs.h buffer.h
        g++ -g -c -Wall search.cc
files.o : files.cc defs.h buffer.h command.h
        g++ -g -c -Wall files.cc
utils.o : utils.cc defs.h
        g++ -g -c -Wall utils.cc
```

This file consists of a sequence of nine *rules*. Each rule consists of a line containing two lists of names separated by a colon, followed by one or more lines beginning with tab characters. Any line may be continued, as illustrated, by ending it with a backslash-newline combination, which essentially acts like a space, combining the line with its successor. The '#' character indicates the start of a comment that goes to the end of the line.

The names preceding the colons are known as *targets*; they are most often the names of files that are to be produced. The names following the colons are known as *dependencies* of the targets. They usually denote other files (generally, other targets) that must be present and up-to-date before the target can be processed. The lines starting with tabs that follow the first line of a rule are called *actions*. They are shell commands (that is, commands that you could type in response to the Unix prompt) that get executed in order to create or update the target of the rule (we'll use the generic term *update* for both).

Each rule says, in effect, that to update the targets, each of the dependencies must first be updated (recursively). Next, if a target does not exist (that is, if no file by that name exists)

[2]Adapted from "GNU Make: A Program for Directing Recompilation" by Richard Stallman and Roland McGrath, 1988.

or if it does exist but is older than one of its dependencies (so that one of its dependencies was changed after it was last updated), the actions of the rule are executed to create or update that target. The program will complain if any dependency does not exist and there is no rule for creating it. To start the process off, the user who executes the `gmake` utility specifies one or more targets to be updated. The first target of the first rule in the file is the default.

In the example above, `edit` is the default target. The first step in updating it is to update all the object (`.o`) files listed as dependencies. To update `edit.o`, in turn, requires first that `edit.cc` and `defs.h` be updated. Presumably, `edit.cc` is the source file that produces `edit.o` and `defs.h` is a header file that `edit.cc` includes. There are no rules targeting these files; therefore, they merely need to exist to be up-to-date. Now `edit.o` is up-to-date if it is younger than either `edit.cc` or `defs.h` (if it were older, it would mean that one of those files had been changed since the last compilation that produced `edit.o`). If `edit.o` is older than its dependencies, `gmake` executes the action "`g++ -g -c -Wall edit.cc`", producing a new `edit.o`. Once `edit.o` and all the other `.o` files are updated, they are combined by the action "`g++ -g -o edit` ···" to produce the program `edit`, if either `edit` does not already exist or if any of the `.o` files are younger than the existing `edit` file.

To invoke `gmake` for this example, one issues the command

`gmake -f` *makefile-name target-names*

where the *target-names* are the targets that you wish to update and the *makefile-name* given in the `-f` switch is the name of the makefile. By default, the target is that of the first rule in the file. You may (and usually do) leave off `-f` *makefile-name*, in which case it defaults to either `makefile` or `Makefile`, whichever exists. It is typical to arrange that each directory contains the source code for a single principal program. By adopting the convention that the rule with that program as its target goes first, and that the makefile for the directory is named `makefile`, you can arrange that, by convention, issuing the command `gmake` with no arguments in any directory will update the principal program of that directory.

It is possible to have more than one rule with the same target, as long as no more than one rule for each target has an action. Thus, I can also write the latter part of the example above as follows.

```
edit.o : edit.cc
        g++ -g -c -Wall edit.cc
kbd.o : kbd.cc
        g++ -g -c -Wall kbd.cc
commands.o : command.cc
        g++ -g -c -Wall commands.cc
display.o : display.cc
        g++ -g -c -Wall display.cc
insert.o : insert.cc
        g++ -g -c -Wall insert.cc
search.o : search.cc
        g++ -g -c -Wall search.cc
files.o : files.cc
```

```
        g++ -g -c -Wall files.cc
utils.o : utils.cc
        g++ -g -c -Wall utils.cc

edit.o kbd.o commands.o display.o \
    insert.o search.o files.o utils.o: defs.h
kbd.o commands.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

The order in which these rules are written is irrelevant. Which order or grouping you choose is largely a matter of taste.

The example of this section illustrates the concepts underlying `gmake`. The rest of `gmake`'s features exist mostly to enhance the convenience of using it.

## 3.2 Variables

The dependencies of the target `edit` in §3.1 are also the arguments to the command that links them. One can avoid this redundancy by defining a variable that contains the names of all object files.

```
# Makefile for simple editor

OBJS = edit.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o

edit : $(OBJS)
        g++ -g -o edit $(OBJS)
```

The (continued) line beginning "`OBJS =`" defines the variable `OBJS`, which can later be referenced as "\$(OBJS)" or "\${OBJS}". These later references cause the definition of `OBJ` to be substituted verbatim before the rule is processed. It is somewhat unfortunate that both `gmake` and the shell use '\$' to prefix variable references; `gmake` defines '\$\$' to be simply '\$', thus allowing you to send '\$'s to the shell, where needed.

You will sometimes find that you need a value that is just like that of some variable, with a certain systematic substitution. For example, given a variable listing the names of all source files, you might want to get the names of all resulting `.o` files. I can rewrite the definition of OBJS above to get this.

```
SRCS = edit.cc kbd.cc commands.cc display.cc \
       insert.cc search.cc files.cc utils.cc
OBJS = $(SRCS:.cc=.o)
```

The substitution suffix '`:.cc=.o`' specifies the desired substitution. I now have variables for both the names of all sources and the names of all object files without having to repeat a lot of file names (and possibly make a mistake).

Variables may also be set in the command line that invokes `gmake`. For example, if the makefile contains

```
edit.o: edit.cc
        g++ $(DEBUG) -c -Wall edit.cc
```

Then a command such as

```
gmake DEBUG=-g ...
```

will cause the compilations to use the `-g` (add symbolic debugging information) switch, while leaving off the `DEBUG=-g` will not use the `-g` switch. Variable definitions in the command lines override those in the makefile, which allows the makefile to supply defaults.

Variables not set by either of these methods may be set as UNIX environment variables. Thus, the sequence of commands

```
setenv DEBUG -g
gmake ...
```

for this last example will also use the `-g` switch during compilations.

## 3.3 Implicit rules

In the example from §3.1, all of the compilations that produced `.o` files have the same form. It is tedious to have to duplicate them; it merely gives you the opportunity to type something wrong. Therefore, `gmake` can be told about—and for some standard cases, already knows about—the default files and actions needed to produce files having various extensions. For our purposes, the most important is that it knows how to produce a file *F*`.o` given a file of the form *F*`.cc`, and knows that the *F*`.o` file depends on the file *F*`.cc`. Specifically, `gmake` automatically introduces (in effect) the rule

```
F.o : F.cc
        $(CXX) -c -Wall $(CXXFLAGS) F.cc
```

when called upon to produce *F*`.o` when there is a C++ file *F*`.cc` present, but no explicitly specified actions exist for producing *F*`.o`. The use of the prefix "CXX" is a naming convention for variables that have to do with C++. It also creates the command

```
F : F.o
        $(CXX) $(LDFLAGS) F.o $(LOADLIBES) -o F
```

to tell how to create an executable file named *F* from *F*`.o`.

As a result, I may abbreviate the example as follows.

```
# Makefile for simple editor

SRCS = edit.cc kbd.cc commands.cc display.cc \
       insert.cc search.cc files.cc utils.cc

OBJS = $(SRCS:.cc=.o)
```

```
CC = gcc

CXX = g++

CXXFLAGS = -g

LOADLIBES = -lm

edit : $(OBJS)
edit.o : defs.h
kbd.o : defs.h command.h
commands.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
```

There are quite a few other such implicit rules built into `gmake`. The `-p` switch will cause `gmake` to list them somewhat cryptically, if you are at all curious. It is also possible to supply your own default rules and to suppress the standard rules; for details, see the full documentation, which is available on our systems through the `C-h i` command in Emacs.

## 3.4 Special actions

It is often useful to have targets for which there are never any corresponding files. If the actions for a target do not create a file by that name, it follows from the definition of how `gmake` works that the actions for that target will be executed each time `gmake` is applied to that target. A common use is to put a standard "clean-up" operation into each of your makefiles, specifying how to get rid of files that can be reconstructed, if necessary. For example, you will often see a rule like this in a makefile.

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

Every time you issue the shell command `gmake clean`, this action will execute, removing all `.o` files.

Another possible use is to provide a standard way to run a set of tests on your program—what are typically known as *regression tests*—to see that it is working and has not "regressed" as a result of some change you've made. For example, to cause the command

```
make test
```

to feed a test file through our editor program and check that it produces the right result, use:

```
test: edit
```

```
        rm -f test-file1
        ./edit < test-commands1
        diff test-file1 expected-test-file1
```

where the file `test-commands1` presumably contains editor commands that are supposed to produce a file `test-file1`, and the file `expected-test-file1` contains what is supposed to be in `test-file1` after executing those commands. The first action line of the rule clears away any old copy of `test-file1`; the second runs the editor and feeds in `test-commands1` through the standard input, and the third compares the resulting file with its expected contents. If either the second or third action fails, `make` will report that it encountered an error.

Figure 1 illustrates a more general set-up. Here, the makefile defines the variable TEST-PROGRAM to be the name of any arbitrary testing command, and TESTS to be a list of argument sets to give the test program. The makefile also includes the template shown in the figure. Suppose that my makefile includes this template and also the definitions

```
TESTPROGRAM = ./test-edit

TESTS = "test-commands1 test-file1 expected-test-file1" \
        "test-commands2 test-file2 expected-test-file2"
```

Then `gmake test` will run

```
./test-edit test-commands1 test-file1 expected-test-file1
./test-edit test-commands2 test-file2 expected-test-file2
```

and will report which tests succeed and which fail. The script `test-edit` in this case could be

```
#!/bin/sh
# $1: command file. $2: output file.
# $3: standard for the output file.
rm -f $2
# The following command runs the editor and compares the output
# against the standard.  This script returns normally if the editor
# returns normally and diff finds no differences.
./edit < $1 && diff $2 $3
```

Of course, doing things this fancy requires that you learn a fair amount about the shell language (the Bourne shell, in this case).

The definition of the `test` target in Figure 1 illustrates the advanced use of shell commands in a makefile. Because the action is a single (compound) shell command—a loop—you must inform `gmake` not to break it into 7 separate commands; that's the purpose of the backslashes at the end of each line. Also, in an ordinary shell script, I'd write `${test}` rather than `$${test}`. However, `gmake` treats `$` as a special character; to avoid confusion, `gmake` treats `$$` as a single dollar sign that is supposed to be included in the command.

## 3.5 Details of actions

By default, each action line specified in a rule is executed by the Bourne shell (as opposed to the C shell, which, most unfortunately, is more commonly used here). For the simple makefiles we are likely to use, this will make little difference, but be prepared for surprises if you get ambitious.

The `gmake` program usually prints each action as it is executed, but there are times when this is not desirable. Therefore, a '@' character at the beginning of an action suppresses the default printing. Here is an example of a common use.

```
edit : $(OBJS)
        @echo Linking edit ...
        @g++ -g -o edit $(OBJS)
        @echo Done
```

The result of these actions is that when `gmake` executes this final editing step for the `edit` program, the only thing you'll see printed is a line reading "`Linking edit...`" and, at the end of the step, a line reading "`Done`".

When `gmake` encounters an action that returns a non-zero exit code, the UNIX convention for indicating an error, its standard response is to end processing and exit. The error codes of action lines that begin with a '-' sign (possibly preceded by a '@') are ignored. Also, the `-k` switch to `gmake` will cause it to abandon processing only of the current rule (and any that depend on its target) upon encountering an error, allowing processing of "sibling" rules to proceed.

## 3.6 Creating makefiles

A good way to create makefiles is to have a template that you include in your particular makefile. Something like the example in Figure 1, for example. You have one or more of these for various uses (C++ programs, Java programs, etc.). For any particular program, your makefile might then look like the following example:

```
PROGRAM = edit

CXX_SRCS = edit.cc kbd.cc commands.cc display.cc \
           insert.cc search.cc files.cc utils.cc

include $(HOME)/lib/c.Makefile.std
```

We will maintain a template like this in `$MASTERDIR/lib/c.Makefile.std`, which you include with

```
include $(MASTERDIR)/lib/c.Makefile.std
```

(always assuming, that is, that you use the standard class setup files, which set the environment variables MASTER and MASTERDIR to the CS61B home directory.)

As a final convenience, the -MM option to gcc creates dependency lines for C and C++ automatically. The template shown in Figure 1 uses this to automatically generate a file of dependencies. The depend special target in in that file allows you to recreate the set of dependencies when needed by typing 'gmake depend'.

## 3.7 Makefiles with Java

To be honest, Java does not show the make utility at its best. The problem is that Java does not really allow the separation of header files from implementation files. For example, suppose file B.java contains uses of methods or classes from A.java. From make's perspective, we have to say that B.class depends on A.java. Thus, whenever a method in A.java is changed, as far as make knows, B.java must be recompiled—even if the signatures and names of the classes, methods, fields in A.java have not changed. There is often nothing for it at the moment but to write trivial sets of dependency rules in which every .class file depends on every .java file. Still, make is useful for making the compilation process easy: you can still arrange for a plain gmake command to compile everything that needs to be compiled. Thus, a Java program contained in files Main.java, Car.java, Truck.java, and Drive.java might use makefile rules like this:

```
JAVA_SRC =  Main.java Car.java Truck.java Drive.java

JFLAGS = -g

CLASSES = $(JAVA_SRC:.java=.class)

all: $(CLASSES)

$(CLASSES): $(JAVA_SRC)
        javac $(JFLAGS) $(JAVA_SRC)

clean:
        rm -rf $(CLASSES) *~
```

As it happens, the javac compiler attempts to do something like this, but more cleverly. If there are no .class files lying around, it will find all the source files that need to be compiled when it is asked to compile the main class or applet. However, javac is erratic at partial compilation—that is, at recompiling all and only the source code that needs to be recompiled when there are still .class files lying around from the last compilation. Although it *often* works, it sometimes will fail to recompile one or more classes[3], which can result in extremely frustrating bugs. Therefore, I suggest that you forgo the temptations that javac offers and instead set up a makefile such as the one above. In fact, we have a template for this purpose, called $MASTERDIR/lib/java.Makefile.std, so that your Makefile can look like this:

[3]What happens is that there are cases in which javac cannot tell from examing a .class file what other classes it depends on. Examples are available on request.

```
JAVA_SRC =  Main.java Car.java Truck.java Drive.java

include $(MASTERDIR)/lib/java.Makefile.std
```

```
# Standard definitions for make utility: C++ version.

# Assumes that this file is included from a Makefile that defines
# PROGRAM to be the name of the program to be created and CXX_SRCS
# to the list of C++ source files that go into it.
# The including Makefile may subsequently override CXXFLAGS (flags to
# the C++ compiler), LOADLIBES (-l options for the linker), LDFLAGS
# (flags to the linker), and CXX (the C++ compiler).

# Targets defined:
#    all:    Default entry.  Compiles the program
#    depend: Recomputes dependencies on .h files.
#    clean:  Remove back-up files and files that make can reconstruct.
#    test:   Run the testing command in variable TESTPROGRAM for each
#            of the arguments given in the variable TESTS.

LOADLIBES = -lm

LDFLAGS = -g

CXX = g++

CXXFLAGS = -g -Wall

OBJS = $(CXX_SRCS:.cc=.o)

# Default entry
all: $(PROGRAM)

$(PROGRAM) : $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) $(LOADLIBES) -o $(PROGRAM)

clean:
        /bin/rm -f $(OBJS) $(PROGRAM) *~

test:  $(PROGRAM)
        for test in $(TESTS); do \
            echo "Running $(TESTPROGRAM) $${test} ..." ; \
            if $(TESTPROGRAM) $${test}; then \
                echo "Test succeeds."; \
            else echo "Test failed."; \
            fi; \
        done

make.depend:
        rm -f make.depend
        $(CXX) -MM $(CXX_SRCS) > make.depend

depend:
        $(CXX) -MM $(CXX_SRCS) > make.depend

# If the make.depend file does not exist, gmake will use the rule
# for make.depend above to create it.
include make.depend
```

**Figure 1:** An example of a standard makefile definitions that can be included from a specific makefile to compile many simple collections of C++ programs.