CS 61B Lab 15
May 3-5, 2000

Goal: To teach you about makefiles and to practice your understanding of the memory model underlying C and C++.

Makefiles are normally processed by a tool called "make", but we will be using the GNU version, called "gmake". makefiles express the dependencies between different files that comprise a large software project. "make" or "gmake" is almost always used by C/C++ programmers in a Unix environment for compiling large multiple-file programs. (It is often used for Java projects as well, and is used by the autograder.)

Copy the Lab 15 directory by starting from your home directory and typing:

```
  mkdir lab15
  cd lab15
  cp $master/lab/lab15/* .
```

Compile the program by typing

```
  g++ -g -Wall -o trees *.cc
```

to create an executable called "trees".

The class TreeNode is declared in the header file "treenode.h" and defined in treenode.cc. TreeNode is similar to the ListNode class in Homework 10, and could be used as a node in a Tree class (though we will not use a Tree class).

Part I: Separate Compilation and Makefiles (1 point)

This program is small enough that it isn't terribly slow to recompile every file after making a change to one of them. In a large software project, however, you only want to have to recompile the files that have actually changed, or the files that depend on the changed files (e.g. a .c file that #includes a modified .h file).

There are two steps to creating the "trees" program. As Lab 14 showed, you may first compile treenode.cc and main.cc, yielding treenode.o and main.o, then link the .o files into an executable. These steps are clumsy, because you have to remember and retype the commands, and figure out what files must be recompiled when something changes. For example, if you modify treenode.h, you must recompile treenode.cc and main.cc, because they both include treenode.h.

The rules for when to recompile are

1) recompile a .cc file if you change it,
2) recompile a .cc if it includes any .h file that has changed (including any .h file included by any other .h file), and
3) relink the .o files if any of them changes (because a .cc file is recompiled).

makefiles record these rules and dependencies so you don't have to remember them. The make/gmake program uses file timestamps to check whether any file is older than one of the files it depends upon, and tries to regenerate the file (called the "target") if so. Makefiles also record the methods by which each file can be regenerated.

Makefile Syntax

A typical makefile is named "makefile" or "Makefile" (the former for this lab). The syntax for makefiles is as follows. To specify dependencies, list a target filename followed by a colon and the files depended upon. Here are some examples for the "trees" program.

```
target          depends on
-----------     ----------------------
trees:          treenode.o main.o
treenode.o:     treenode.cc treenode.h
main.o:         main.cc treenode.h
```

Standard libraries like iostream.h are not listed as dependencies because they don't ordinarily change. This example tells gmake what the dependencies are, but it doesn't tell gmake what it should do to update files.

To tell gmake how to update a target, add commands on the line after the target's dependencies, indented by one or more tabs characters--NOT SPACES! (Spaces will cause error messages, for reasons of historical stupidity. See the "Unix Haters' Handbook" for details.) A complete makefile might look like this.

```
# A makefile for tree nodes

trees: main.o treenode.o
        g++ -o trees main.o treenode.o
treenode.o: treenode.cc treenode.h
        g++ -c -Wall treenode.cc
main.o: [finish this part yourself]
```

This example is included as "makefile" in the hw10 directory. Complete the makefile by writing the dependencies and command for main.o. You may insert blank lines for readability. Comments in a makefile begin with "#" and can start anywhere on a line.

Running gmake

You can run gmake from the shell as "gmake <target>", where <target> is the name of the file you want to update (e.g. trees). If you run gmake with no targets on the command line as parameters, it tries to make the first thing in the file--in this case "trees". In addition to generating the named file, it will recursively generate any file the named file depends on, if it has been told how to make those files and they aren't already up to date. By "up to date" I mean that the file exists, and the files it depends on haven't been changed since the file was generated. (gmake deduces this based on the timestamps specifying when the files were written.)

A makefile can also have commands to help manage software. A common convention is to have command called "clean" that will remove .o and (sometimes) executable files (which can be quite large) and keep only the .cc and .h files. Add a command named "clean" to the botton of the makefile. "clean" depends on no file, so no dependencies are listed. The command for clean should be

```
        rm *.o *~ trees core
```

which removes .o files, emacs backup files ending with ~, the trees executable, and core. Be careful not to put a space after either of the "*"s or you will end up deleting everything the directory. (One of the reasons for defining a make command is to avoid these kind of typing errors.) Check that your makefile is working by typing "gmake clean" and checking (with ls) that the directory has been cleaned up. Then type "gmake" to regenerate the .o and trees files. You should find the latter command useful while working on Part II.

Part II: C++ Memory Management (1 point)

In Lectures 38 and 39 you learned about C/C++ memory management. Below is a picture of a possible memory configuration that uses TreeNode objects. Fill in the body of the part2() function so that this will be an accurate picture of the program stack and heap just before the part2() function returns. Your function should not contain any variables other than the ones shown here. The X's in the "tn1" box and elsewhere are NULL pointers. There is more than one correct solution.

```
  The stack      The heap (everything to the right of the stack)

  |        |
  | part2()|
  |    --- |
  | tn1|X| |
  |    --- |     -----------     -----------
  | tn2|.+-+---->| item: 2 |     | item: 3 |
  |    --- |     | left: .-|---->| left: X |
  | tn5|.+-+--   | right:.-|-    | right:X |
  |    --- |  \  ----------- \   -----------
  |--------|   \              \
  |        |    \              -->-----------
  | main() |     \                | item: 4 |
  ----------      \               | left: .-|-
                   \              | right:X | \
                    \             -----------  \
                     \                          -->-----------
                      \                            | item: 5 |
                       --------------------------->| left: X |
                                                   | right:X |
                                                   -----------
```

Part III: More on the C++ Memory Model (2 points)

Draw a picture (using paper and pen) of how memory looks just before the function part3() ends. Show both the program stack and heap as we have done above.

Check-off
----

1 point: Show your TA or lab assistant your completed makefile. Show that "gmake clean" and "gmake trees" work correctly.
1 point: Show your TA or lab assistant your code for the part2() function.
2 points: Show them your picture of memory for part3().

Notes

When "trees" is run, execution begins with the main() procedure in main.cc, which has "#include" statements at the top. The first of these #includes the iostream standard C++ library. The second #includes the header file for treenode.h. The first is in "<>" brackets to tell the compiler to search in the places where standard libraries are normally hidden. (These places vary from compiler to compiler.) The second is in quotes to tell the compiler to search in the current directory.

Near the beginning of treenode.h are two lines beginning with "#". These refer to variables that the C preprocessor (part of the compiler) interprets (they are not C++ code). The first of these,

```
#ifndef _treenode_h
```

says that if "_treenode_h" is *not* defined, then continue processing the file up to the "#endif" at the bottom of the file. If "_treenode_h" *is* defined, then ignore everything up to the #endif. The second line:

```
#define _treenode_h
```

defines the variable "_treenode_h" so that, if a file includes "treenode.h" more than once, the compiler won't process the contents twice and complain about redefined symbols. Why would any file include treenode.h more than once? It is not uncommon for a .c or .cc file to #include two different .h files that each #include some other common .h file.