CS 61B Lab 14
April 26-28, 2000

Goal: To introduce you to g++, the GNU C++ compiler, and gdb, the GNU debugger. gdb is notably better than the Java debugger, but still takes some effort to learn. However, C and C++ are much harder to debug than Java, and you are STRONGLY encouraged to learn gdb as well as you can before hitting the upper division courses. (If you have a PC with Microsoft's Visual C++ or Visual Java, these programs have visually-oriented debuggers that you may find preferable to Unix-based debuggers like gdb.)

Copy the Lab 14 directory by starting from your home directory and typing:

```
  mkdir lab14
  cd lab14
  cp $master/lab/lab14/* .
```

If the gcc and gdb reference cards are not attached to this lab, print the files gcc-refcard.ps and gdb-refcard.ps (using lp or ghostview, NOT enscript). We recommend you hang on to these through the rest of your CS career at Berkeley. They will be useful in many upper-division classes.

Part I: Compiling C++ (1 point)

There are some essential differences between how javac works and how g++ works.

1) javac produces class files with a .class extension. Class files contain Java bytecode and are executed using the interpreter "java". g++ produces _object_files_ with a ".o" extension. Object files are not interpreted; they are _linked_ together to create an executable, which contains raw machine instructions and can be run without an interpreter or any other accompanying software. Executables have no extension, and are run by typing their names (like a Unix command). g++ does both jobs--compiling .cc files into .o files, and linking .o files into executables. These two steps can be done separately (the norm for a program made of many .cc files) or in a single command.

2) When you type "javac Filename.java" the compiler compiles not only the Filename class, but any other files that Filename depends upon. C and C++ syntax do not easily permit this automatic inference of file dependencies, and C compilers don't try. It's your job to keep track of which files must be compiled to make a complete program (often done with the help of _makefiles_ and the make or gmake programs). For this lab, our program has only one file.

Try to compile main.cc. You can do this from a shell, or by using the compile command (ctrl-x ctrl-e or meta-x compile) in Emacs. (In the latter case, Emacs will remember your command the first time you type it, so you won't have to type it again.) The command for compiling main.cc is

```
  g++ -g -Wall -c main.cc
```

If there are no compiler errors, the object file main.o is created.

- The "-c" switch tells g++ to compile main.cc into object file main.o, rather than creating an executable.
- The "-g" switch tells g++ to include debugging information in main.o.
- The "-Wall" switch tells g++ to warn you about potential problems in your program, even if they don't prevent it from compiling. It will warn you when you do dangerous things like "if (a = b)" instead of "if (a == b)". (Java won't compile thie former, but g++ will, thereby creating a hard-to-find bug.)

Fix the errors discovered by the compiler. Most of the syntax in this program is similar enough to Java's that you should have no difficulty spotting the problems. Where the problem is with C++ style I/O, you can find the problem by looking at a couple of statements before and after the offending one.

Part II: Linking the Program (1 point)

"Linking" may seem to be a misnomer when there is only one .o file in a program, but we must still run a linker to turn the object file into an executable. Type the following command.

```
  g++ -g -o dumbmath main.o
```

- The "-o dumbmath" switch specifies the name of the executable file to write.

The two g++ commands above could have been combined into one:

```
  g++ -g -Wall -o dumbmath main.cc
```

but it's useful to learn how to run g++ in two steps. (Your TA or lab assistant will look for your main.o file, so use the two-step process for this lab.)

Part III: Debugging Run-Time Errors (2 points)

Run the executable file by typing "dumbmath". Observe that dumbmath is not correctly computing GCDs, nor is it computing good estimates of pi. Run dumbmath within the debugger by typing

```
  meta-x gdb
```

within emacs. On the emacs command line, fill in the executable name to yield

```
  gdb dumbmath
```

Check out the gdb reference card for useful commands. The most helpful for you now are the following.

In the gdb window:

```
  break proc         Set a breakpoint at the beginning of procedure "proc".
  run                Run the program from the beginning.
  step               Single-step through code; step into function calls.
  next               Single-step through code; run function calls in full.
  cont               Continue running the program from the current breakpoint.
  print expr         Print the value of the variable or expression "expr".
  bt                 Print the program stack.
```

All but "run" can be abbreviated to one letter (b, s, n, c, p). If your code goes into an infinite loop, and you want to make it break, ctrl-c ctrl-c might work.

In the program window (only after you've begun running but are stopped):

```
  ctrl-x <space>     Set a breakpoint at this line of code.
```

Experiment with gdb and isolate the errors. There is one error in each of the three functions called by main (but none in main itself). Each error requires very little typing to fix, but you may need to read the leading comments of the function and trace the execution by setting breakpoints, single-stepping, and examining variables.

When you think you have found each error, correct it, save the file, recompile, and execute to see if the problem is solved.

```
Check-off
---------
2 points:  Show your TA or lab assistant your main.o and dumbmath files.
           BRIEFLY list the changes you had to make to get main.cc to compile.
2 points:  Run your working dumbmath program from gdb within emacs.
           Deomonstrate how you found a bug (of your choice) by setting
           breakpoints and/or examining variables.
```