CS3204 Operating Systems - Spring 2001
Instructor: Dr. Craig A. Struble
Project 0: Program Compilation and Submission

Assigned: Wednesday, Jan. 17 **Due:** 11:59.59 p.m., Thursday, Jan. 25

1 Introduction

Linux is the operating system used in this course for assignments. The purpose of this optional assignment is to familiarize you with the development and submission process under Linux. The specification contains a small program that you must transcribe to a Linux machine, compile, execute, and submit. If you complete this project, you will be allowed to correct one invalid submission during the semester without penalty.

2 Specification

You are to enter, compile, execute, and submit the program contained below. Files created during this assignment should be stored in a directory named `project0`. You should use a standard user account under Linux to perform all of your work for this assignment. Using the *root* account is dangerous and may result in destroying your Linux system.

2.1 Program

The contents of three C++ source files are contained below. You are to use an editor such as `vi`, `emacs`, `xemacs`, or `pico` to create the two source files `main.cpp`, `sine.h`, and `sine.cpp` under Linux.

Compile these three files and link them together to create an executable named `angle`. You will need to link with the math library to create the executable. Execute `angle` several times, and create sample input and output files to include in your submission.

```
//////////////////////////////////////////////////////////////////////
// File: main.cpp
// Purpose: This program calculates the sine of an angle given in
//          degrees. This file contains the main driver for the program,
//          which asks the user for an angle and prints out the
//          sine of that angle.
// Programmer: Craig A. Struble
// Last Modified: Jan. 16, 2001
//////////////////////////////////////////////////////////////////////

#include <iostream>
#include "sine.h"     // declares degreeSine

using namespace std;

// The argument count, argc, and argument list, argv, are unused.
int main(int argc, char **argv) {
```

```
    double degrees;   // the angle in degrees
    double sine;      // the sine of the angle

    // Get the angle from the user
    cout << "Please enter the angle in degrees: " << flush;
    cin >> degrees;

    // Calculate the sine of the angle and print the result
    sine = degreeSine(degrees);
    cout << "The sine of the angle is " << sine << "." << endl;

    // Completed successfully
    return 0;
}
```

```
////////////////////////////////////////////////////////////////////////
// File: sine.h
// Purpose: Contains the function prototypes for public functions
//          defined in sine.c.
// Programmer: Craig A. Struble
// Last Modified: Jan. 16, 2001
////////////////////////////////////////////////////////////////////////
#ifndef _SINE_H_
#define _SINE_H_

// Calculate the sine of an angle given in degrees.
double degreeSine(double degrees);
#endif
```

```
////////////////////////////////////////////////////////////////////////
// File: sine.cpp
// Purpose: Contains functions for calculating the sine of an angle
//          given in degrees and for computing degrees to radians.
// Programmer: Craig A. Struble
// Last Modified: Jan. 16, 20001
////////////////////////////////////////////////////////////////////////
#include <cmath>          // for sin and M_PI
#include "sine.h"         // for degreeSine prototype

using namespace std;

////////////////////////////////////////////////////////////////////////
// Function: degreesToRadians
// Purpose:  A private function converting a value in degrees
```

```
//           to a value in radians. Uses the fact that
//           360 degrees equals 2 pi radians.
// Parameters: degrees - the value in degrees.
// Returns:    the value converted to radians.
////////////////////////////////////////////////////////////////////////
static double degreesToRadians(double degrees) {
    // A full circle is 360 degrees
    const double FULL_CIRCLE = 360.0;
    // A half circle is 180 degrees
    const double HALF_CIRCLE = 180.0;

    // convert the degrees to be in the range [0-360)
    while (degrees < 0.0 || degrees >= FULL_CIRCLE) {
        if (degrees >= FULL_CIRCLE) {
            degrees = degrees - FULL_CIRCLE;
        } else {
            degrees = degrees + FULL_CIRCLE;
        }
    }
    return (degrees / HALF_CIRCLE) * M_PI;
}

////////////////////////////////////////////////////////////////////////
// Function: degreeSine
// Purpose:  Returns the sine of an angle given in degrees.
// Parameters: degrees - the angle in degrees.
// Returns:    The sine of the angle
////////////////////////////////////////////////////////////////////////
double degreeSine(double degrees) {
    double radians;

    radians = degreesToRadians(degrees);  // convert to radians

    // the math sin function calculates sine for radians
    return sin(radians);
}
```

2.2 Compilation

The command to compile C++ programs in Linux is `g++`. (Note that `c++` will work as well, but it's the same program.) Unlike Microsoft Visual C++, compilation and linking of programs in Linux is normally done in several steps: compile each source file individually, link the resulting object files into an executable program. You can compile and link in one command, but this is usually not done, especially for larger programs.

2.2.1 Generating Object Files

To compile your C++ program and generate an object file, the `-c` option is used. For example,

```
g++ -c util.cpp
```

compiles the `util.cpp` file and generates the `util.o` object file. You perform this command for each `cpp` file comprising your program.

2.2.2 Linking

Once you have generated all of the object (`.o`) files, you link them together to form an executable. The `-o` option is used to name the resulting executable. The parameter immediately following the `-o` option will be the name of your program. For example, suppose `main.o` and `util.o` are the object files that make up your program, and you want to generate the executable `myprog`. The following command would be used:

```
g++ -o myprog main.o util.o
```

The executable should be created in the current directory.

Occasionally you will need to link with an additional library, such as the standard math library. To do this, you use the `-l` option followed immediately by the name of the library. The standard math library name is `m`, so the command to link previous program with the math library is

```
g++ -o myprog main.o util.o -lm
```

Note that the actual file to be linked is found in one of a standard set library directories (e.g., `/lib`, `/usr/lib`, `/usr/local/lib`, `/usr/X11R6/lib`), and incorporates the library name into a much longer filename for the form: `lib`*name*`.a` or `lib`*name*`.so`. For example, the contents of the math library above are actually in the file `/usr/lib/libm.so` (which happens to be a link to the file `/lib/libm.so.6` in RedHat 6.2).

2.2.3 Debugging

Throughout the semester, you will likely need to debug your programs. Normally, the compiler does not generate debugging information as part of your program. To turn this on, use the `-g` option with `g++`. To compile `util.cpp` with debugging information, the following command is used:

```
g++ -g -c util.cpp
```

The `util.o` file will contain debugging information so that debuggers like `gdb` and `ddd` will work properly.

You do not need the `-g` option when linking, although including it will not cause problems. For more information about debuggers under Linux, visit the course website.

2.3 Archiving

The Curator is used by this class to collect your project submissions. To make a submission, you need to place all of your source files and supporting files (i.e., `README` file, `build` script or `Makefile`, and sample input/output files) in a single archive. The most commonly used command for generating archives under Linux/Unix is the `tar` command. In case you are wondering, `tar` stands for *Tape ARchive*, because it was originally used to store files onto tapes.

The `tar` command is powerful and has several options, but I will focus only on the important ones.

The `-c` option is used to create a tar archive. You list the files to archive at the end of the command line. If the file named is actually a director, `tar` will recursively store all of the files contained in that directory. Suppose that `project` is a directory containing project files. To archive all of the files in the `project` directory and store them in an archive named `project.tar`, use the command

```
tar -c -f project.tar project
```

All files are stored in a directory named `project` in the archive.

The `-f` option names a file to create, extract, or list. The parameter following the `-f` is the name of the file. Without this option, `tar` defaults to the device `/dev/rmt0`. If you forget the `-f` option, tar will either print an error or appear to hang. The following command will extract files from the archive named `project.tar`:

```
tar -x -f project.tar
```

The `-v` option denotes verbose processing. When used with other options for creating and extracting archives, the files added or extracted are printed out to the screen. The following command will extract files from the archive named `project.tar` and print each file name extracted from the archive:

```
tar -x -v -f project.tar
```

The `-t` option lists the files stored in an archive file. To list the files stored in an archive named `project.tar`, use the command

```
tar -t -f project.tar
```

You can also include the `-v` option for detailed information about each file.

The `-z` option is used to compress and uncompress data using `gzip` when creating or extracting/listing an archive respectively. To create a compressed archive named `project.tar.gz` containing the files in the `project` directory and listing those files as they are processed, use the command

```
tar -z -c -v -f project.tar.gz project
```

2.4 Build Scripts and Makefiles

You are expected to include either a `build` shell script or `Makefile` to automate the process of compiling and linking your projects. The `build` script file is simply an text file containing the compilation steps used to compile your program. Use the `chmod` command to make sure that script is executable in order to run it from the command line.

```
chmod +x build
```

Makefiles are more sophisticated and their creation is beyond the scope of this document. However, makefiles should have been covered in a previous course with Unix content. Makefiles have a benefit over build scripts in that they do not recompile source files that have not changed. This saves time, especially with large projects. You makefile must be created in such a way that the single command

```
make
```

will create all executables needed for an assignment. You are encouraged to practice using makefiles but not required to use them.

3 Submission

We use the Curator, `http://ei.cs.vt.edu/~eags/Curator.html` to collect program submissions. The URL for submission is `http://spasm.cs.vt.edu:8080/curator/`. Only the servlet interface to the Curator is supported. No grading is done by the Curator.

You are to submit a single tarred (`man tar`) and gzipped (`man gzip`) archive containing

- A text file named `README` describing the program, describing the contents of the archive, providing building instructions (including the platform you used for development), a user's guide (including how to start the program), and examples of usage with your test files;
- The source code for your program;
- The test files you used for your program;
- A script named `build` or a suitable `Makefile` for building your program.

Your files must be contained in a directory named `project0` of the archive. Be sure to include only the files listed above. Do not include extra files from an integrated development environment such as `configure` scripts, automake related files, etc. This is primarily an issue if you are using *KDevelop*.

Be sure to include your name in all files submitted. **DO NOT** include executables or object files of any type in the archive. **Be careful to FTP in binary mode if you are transferring your file to a Windows machine before submitting to the Curator.**

4 Programming Environment

As stated in the syllabus, you may use either FreeBSD or Linux and `gcc/g++` to implement this project. **All data structures used in your program must be student implemented. Using the standard template library (STL) or other third party libraries for data structure implementations is strictly prohibited.** Using C++ input and output streams and C++ strings is OK.