1 File Manager

Go to the CSCI 203 home page and click on the “File Manager” tutorial and complete it.

2 Mixed Mode Arithmetic

If you have not already done so, create a sub-directory `Lab3` within your `Labs` directory. Move into the new directory and complete the following lab work there. Remember to insert your banner file at the beginning of each file and update it.

Type in the following program and save it as `temp.cc` in your `Lab3` directory. Compile and run the program. In **emacs** use the **Tab** key to indent.

```
#include <iostream>

// illustrates I/O of ints and doubles
// illustrates arithmetic operations

int main() {
    int    intFahr;
    double doubleFahr;

    // convert 'int' temperature
    cout << "Enter a Fahrenheit temperature ";
    cin  >> intFahr;

    cout << intFahr << " = "
         << (intFahr - 32) * 5/9
         << " Celsius" << endl;

    // convert 'double' temperature
    cout << "Enter another temperature ";
    cin  >> doubleFahr;

    cout << doubleFahr << " = "
         << (doubleFahr - 32.0) * 5/9
         << " Celsius" << endl;

    return 0;
}
```

Note: This program illustrates that it is possible to use an expression in an output statement — this is done in both `cout` statements.

Include in your `handin.txt` file answers to the following questions. Notice that the word “why” appears in each question. Be sure to answer appropriately.

1. Run the program, entering the value 40 for both temperatures. Why does the program return two different answers?

2. If the output expressions in the program are changed as follows

   ```
   cout << intFahr << " = "
        << (intFahr - 32) * (5/9)
        << " Celsius" << endl;
   ....
   cout << doubleFahr << " = "
        << (doubleFahr - 32.0) * (5/9)
        << " Celsius" << endl;
   ```

   the value printed by both statements will be 0. Why is this?

3. What is printed (and why) if parentheses are not used in either of the output expressions? In other words, the output statements are

   ```
   cout << intFahr << " = "
        << intFahr - 32 * 5/9
        << " Celsius" << endl;
   ....
   cout << doubleFahr << " = "
        << doubleFahr - 32.0 * 5/9
        << " Celsius" << endl;
   ```

4. If the output statement using `intFahr` is changed as follows

   ```
   cout << intFahr << " = "
        << (intFahr - 32.0) * 5/9
        << " Celsius" << endl;
   ```

   what will the output be if the value of 40 is entered? Why?

3 Dealing with Errors

Copy the file `errors.cc` from `~cs203/Labs/Lab3` into your `Lab3` directory. Spend some time and study the program. Compile this program as follows:

```
g++ errors.cc -o errors.exe
```

and run it.

3.1 Syntax Errors

Remove the *semicolon* from the end of line 9 (the one with `int a;`), save the file and recompile the program. Remember to use the up arrow to retrieve the last command in a Terminal window.

Study the error message. It is important that you learn how to deal with error messages. Notice that the first error message says:

```
errors.cc: In function ‘int main()’:
errors.cc:10: parse error before ‘double’
errors.cc:28: ‘b’ undeclared (first use this function)
errors.cc:28: (Each undeclared identifier is reported
errors.cc:28: only once for each function it appears in.)
```

This says the first *syntax error* was detected in the function `int main()`, at line 10, before the occurrence of `double`. The problem, of course, is that there is nothing on line 10 before the `double`. This can only mean that the error is at the end of line 9 — the semicolon is missing. This error message seems odd, but remember that the message is given when the compiler discovers an error. In this case the compiler doesn't know there is an error until it reads `double` — the compiler knows that `double` cannot terminate a statement. This is a very common occurrence when reading compiler errors. **Remember:** Compiler errors are often in the line *before* the indicated line.

Now look at the second error message. On line 28 **b** is undeclared. This must mean, because of the error on line 9, the compiler doesn't have a definition for the identifier **b**. So later, on line 28, the compiler reports that **b** (in line 28) has not been declared. This illustrates another common situation — an error on one line can make the compiler think there are errors on other lines, even though the other lines are correct — these are phantom errors and they will go away when the original error is corrected. (It's interesting to notice that **a** on line 22 hasn't caused an error. So even though the declaration of **a** was improperly terminated, the declaration of **a** was still recognized.)

Therefore, a *good strategy* is to fix the first error or the first few errors and then recompile.

Go to line number 9 by entering `C-c g` and then the line number, i.e. 9, when prompted at bottom. Add the semicolon at the end of line 9.

3.2 Missing Include Errors

Comment out the line with `#include <cmath>` and recompile. "Comment out" usually means to add the `//` comment marker at the beginning of the specified line as follows.

```
// #include <cmath>
```

Study the error message which results.

```
errors.cc: In function ‘int main()’:
errors.cc:28: implicit declaration of function
              ‘int pow(...)’
```

This message indicates that the function `pow()` has been used, but there is no definition. Remove the comment marking from the `#include <cmath>` line before moving on.

3.3 Semantic Errors

Comment out line 28 and add a new assignment statement before it as follows.

```
b = sqrt(-2.0);
// b = pow(2.0, 4.0);
```

Recompile and run the program — we know something funny should happen since we shouldn't be able to take the square root of a negative number! The error which results is a *semantic error* — an error due to wrong meaning — is signaled by the `NaN` (not a number) reported as the value of **b**. This is one way a running program can report a faulty computation.

Another related problem involves division by zero. If you replace the call to `sqrt(-2.0)` by the expression '`a/0`', the compiler gives a general warning that division by zero will occur on line 28. If you run the resulting program, it will print the error message "Floating Exception" when line 28 is executed.

Do not change the program from its current configuration.

3.4 More Semantic Errors

Some semantic errors are caught by the compiler. Replace the `'a/0'` with `sqrt(s3)` and recompile the program. Remember that `s3` is declared as a `string` object. Study the error message. In this case, `g++` can detect the improper use or meaning — you can't take the square root of a string object. Change the argument of `sqrt` to 2.0 (*not* -2.0).

3.5 Intent Errors

Some programming errors are referred to as *intent* errors. A program may run and execute without errors, but it may not be correct. Modify the `cout` statement just before the `return` statement to the following.

```
cout << "b is " << a << endl;
```

Now compile and run the program. Although no errors are generated, the output of the program is not correct. Do you see why?

Note: It is important for you to learn how to read and extract useful information from the compiler's error messages. After a reasonable attempt at deciphering them yourself, you should ask someone else for help. It is equally important to verify that the output generated by your program is correct. Remember, a program can compile and execute without errors, yet still not be correct.

4 Animation in Samba

Copy the file `bullseye.cc` from `~cs203/Labs/Lab3` into your `Lab3` directory. Read and study the program.

Compile the program by typing the following:

```
g++ bullseye.cc -L/home/hydra/COURSES/cs203/lib -lSamba -o bullseye.exe
```

Run the program and feed the output to Samba by typing the following:

```
bullseye.exe | samba
```

Remember, you must move the main Samba window in order to get access to the control panel containing the `Start` button. Press "Start" to see picture.

Next, change the color of `Circle` object `c2` to "blue".

Now, modify the program to have each `Circle` object start in a different corner and *move* to the center. Use the `MoveToPosition()` member function to move them. Place them after the `Display()` member functions. For example, to move `Circle c1` from its original position to the position $(0.5, 0.5)$, type the following:

```
c1.MoveToPosition(0.5, 0.5);
```

The Samba screen has coordinates of 0.0 to 1.0 in the **x** direction and 0.0 to 1.0 in the **y** direction. Position (0, 0) is the lower left corner.

Before you press the **Start** button on the **Polka Control Panel**, adjust the speed to **Slow**. You may alter the speed of the animation as it runs by adjusting the slider.

After the circles stop, play with the **In** and **Out** buttons on the bottom.

Include in your `handin.txt` file, a listing of your working program.

5 What To Hand In

Hand in the answers to the questions from section 2 and the listing of your working program from section 4. *Don't* forget the banner at the beginning of the file!

Also: When you hand in a lab or project, please *staple* the sheets from your lab or project together. This will help the graders and instructors a great deal. *Please obtain and use a stapler!!*