CSCI 2270 Lab 1: Compiling simple C++ code in the lab environment.
Files due by moodle 1/18/2014, by 11:50 pm.

Purposes:

1. Make you familiar with the environment we're using (Geany, Dropbox, and Moodle).
2. Review/introduce common variable types in C++.

Files to submit: `lab1_task1.cxx` and `lab1_task2.cxx`.

Begin by logging onto a lab machine and making a Dropbox account if you do not already have one.

I've illustrated this setup using Geany, which is one way you can write and edit C++ code in the lab. (If you prefer other editors, you can use them; Geany here is just for illustration.)

In C++ and Java, you must define the types of the variables you use. If you have seen Python or MATLAB before, you might not be used to defining variable types every time—on the other hand, you can still write code that violates the rules of types in each language. For instance, in Python, the + symbol adds two numbers to get their sum, or adds two text strings by sticking them together, but just try to add an integer to a string with +, and see how far you get. (It makes no sense. What's 1.25 + "banana"? Meaningless!)

You can assign

```
int a = 3;
```

and

```
double b = 8.4;
```

No surprise yet, these are expected values for these types. But assignments between related types are directional. You can also assign

```
double b = 8;
```

because assigning an `int` to a `double` is no problem; `b` just thinks of this as 8.0.

If you instead say

```
int a = 3.4;
```

you will find that `a` is now storing a 3, and has lost the part of the number after the decimal point.

The other bad news about numeric types is that they can't represent extremely large or small numbers. For instance, if we say

```
int a = 99999999999999999999999999999999999999999999999;
```

we will get a warning at compile-time about 'overflow', which is the error you get from assigning a value that is bigger than what the `int` data type can hold. If we go ahead and run the code without the warning, the statement

```
      cout << a << endl;
```

will show us (improbably) that `a = 2147483647`, which doesn't look anything like what we said it should be. (This is the upper limit of the values a signed integer can take on, in our C++ world.)

The other thing that can happen in code when you increment integers is that when the number becomes too large to fit in the allowed range, it 'rolls over' to 0. A similar problem appears when you have code like the following, given an 'unsigned' integer that can't become negative:

```
      unsigned int q = 0;
      cout << q-- << endl;
```

Task 1 in this lab will involve this rollover.

It's good practice to set all your variables—in case they don't get initialized to sensible values. For example, the loop below will be infinite if `z` is initialized to a number > 0, and will terminate if `z` is initialized to something <= 0, because `z` starts with some random garbage value we forgot to initialize. In older versions of C++, you can't trust your variables before they get a starting value.

```
      int z;
      while (z != 0)
            cout << "z is " << z++ << endl;
```

Task 0: If you have not made an account on Dropbox, please do so now, so that you can copy files to the cloud. Please also make an account on the Moodle site (http://moodle.cs.colorado.edu/**).** To enroll in the class, use the enrollment key (your TA will tell you this in class, and I will tell you this in lecture).

Task 1. Simple loop problem.

Download the file `lab1_task1.cxx` from the Moodle site and open it in Geany. (You can start Geany by clicking the button that looks like this

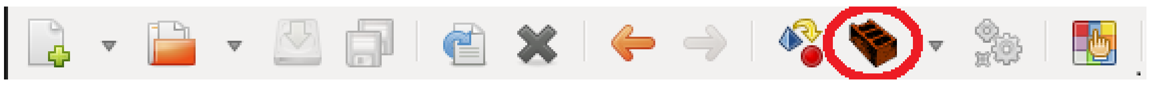
on the left-hand side of the screen. This gives you a (somewhat klutzy) editor.

The `lab1_task1.cxx` code has a loop that is trying to use an unsigned integer to print the numbers from 25 to 0. Unfortunately, this loop is buggy. Here is the buggy part:

```
      unsigned int m = 25;
      while (m >= 0)
```

```
            cout << m-- << endl;
```

First, ask Geany to 'build' the code.   Look at the taskbar at the top of Geany and click on the bookshelf icon, which is circled below, to build.

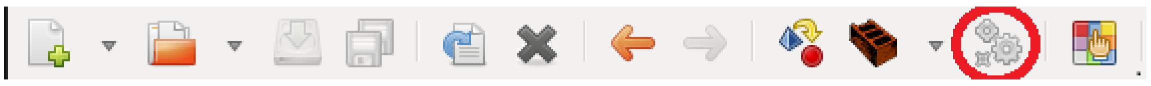


In the bottom window, you'll see that clicking that bookshelf has issued the command:

```
g++ -Wall -o "lab1_task1" "lab1_task1.cxx"
```

Check the window at the bottom for any errors in the compilation of the starting code. (If you start with mine, it should build (compile); alas, when it runs, it will be buggy.

Run the code by clicking on the 'gears' icon, which is circled below.



Running should confirm that this code is somehow not doing the right thing. (Hint: if you need to stop the code, hitting the keys Control and C at the same time may help you here.)

Next, rewrite this loop so that it still relies entirely on `unsigned ints`, but now works as it should. The information in this handout should be enough to fix this. (A subtraction may help.) When it works, show the code and the output to your TA. (Of course, you could get round this by letting the `int` counter be signed, but there are cases where `unsigned ints` make better sense to use, so it's good to know how to make them behave well.)

Task 2. Find floating-point single- and double-precision 'epsilon' values.

Just as `ints` are restricted to a certain range, so are floating-point variables like `floats` and `doubles`. When you work with these variables, a common problem is 'underflow', where the value of a floating-point number gets so close to zero that it begins to act like a zero. We're going to produce an estimate of this number in our C++ environment. A simple attack on the problem follows:

1. Create a floating point variable called `eps1` and initialize it to 1.
2. As long as `eps1` is noticeably greater than zero, halve `eps1`.
3. When `eps1` is *no longer greater than zero* in the computer's mind, stop halving it and instead print `2 * eps1`, the last value of `eps1` you saw that was greater than 0 for the computer. Make a note of this number. It's (approximately) the smallest number the computer thinks is greater than zero.

Make a new file in Geany called `lab1_task2.cxx`. You can start with my empty version from the Moodle. Code up a method for computing this epsilon for a floating-point variable; my version calls this `underflow_float( )`. The method should return the smallest value the computer considers to be above 0 (or, at least, that value within a factor of less than 2).

When you have `underflow_float( )` working, copy your method to another one, called `underflow_double( )`, that defines a double precision number `eps2` and performs the same calculation. Do you get the same answer? Show your TA that your code produces a reasonable answer for `floats` and `doubles`.

This exercise, by the way, is something you need to remember when comparing single or double precision floating point values. Small errors (at the level of this epsilon) can sometimes make it dangerous to compare floating-point numbers using the equals (`==`) operator, like this:

```
        if (a == b)       // dangerous
```

But if you know the (approximate) value of epsilon, you can instead compare `fabs(a – b)` to epsilon, where fabs is the C++ command for the absolute value

```
        if (fabs(a – b) < epsilon)        // safer
```

and if this condition is true, the numbers `a` and `b` can be considered equal.

Before you leave, upload your two files (the fixed `lab1_task1.cxx` and `lab1_task2.cxx` files) to the moodle. If you can, show your TA the 2 files and their output, so he or she can grade you more quickly. You'll be using this web page to submit all your work for the class, and we need to make sure that there are no bugs in the process this week. Also, please copy your files to your Dropbox folder for CSCI2270, to back them up safely.