COSC 2P95 – Lab 2 – Section 01 – Compilation, loops, and… stuff

As with all other labs for this course, you'll still be using Linux for this lab. As such, reboot and log back in if necessary (refer to Lab 1 for details).

In this lab, you'll be writing your first C++ programs, and learning to use variables, operators, conditionals and loops. Some introductory steps will be provided, but by the end you'll be figuring out how to do solve some tasks by yourself.

Again, check the final page to see what you actually need to do to submit.

Compiling Your First Program

Before we can learn any specifics, we first need to know that we can compile a program.
To that end, let's try compiling our first 'hello world' example from class.

So first, create a folder to put your work for this lab, and then let's grab the example!

You have two options:

1. Download it from the webpage in a browser (http://www.cosc.brocku.ca/~efoxwell/2P95/code/02/slide08-10/exampleC.cpp), and save it in the folder you created.
2. Download it directly from your bash terminal:

```
wget http://www.cosc.brocku.ca/~efoxwell/2P95/code/02/slide08-10/exampleC.cpp
```

wget is a very simple tool for downloading pages/files, primarily from web addresses. If you're interested, the `man` page has additional usage tips.

So, let's compile it:

1. Navigate to the folder where the file is (if you used wget to retrieve it, you should already be in the correct directory)
2. Type: `g++ exampleC.cpp`

Let's verify that the executable is in the current folder (`ls`). It should display as `a.out`.
You can execute it by typing `./a.out` but that '.out' stuff isn't really ideal. So let's delete that (`rm a.out`), and then try again:

```
g++ -o ex exampleC.cpp
```

This time, we're telling it the filename for output. Running it should be easy to figure out… right?

Before we get any further, make sure to read the source file, to understand how it works.

Step 1 – The Basics

Let's start by just making sure we can use variables, values, assignment, and standard mathematical operators.
Start a new program to edit. You can use either gedit or nano, depending on your preference.
It's worth noting that, though you may need to select the language manually, gedit *can* provide *syntax highlighting*.

Try entering the following program:
```
#include <iostream>
int main() {
	int i,j;
	double c,d,e;
	d=3.7;
	i=e=j=d;
	c=(int)d;
	std::cout<<"i: "<<i<<"\tj: "<<j;
	std::cout<<"\tc: "<<c<<"\td: "<<d<<"\te: "<<e<<std::endl;
}
```

Compile it. Run it. Understand it.
What's going on with `e`? Why doesn't `c` round up?

Quick version: `i=e=j=d` is a multiple assignment. Like any other expression, you need to work inwards, and move outwards. In this case, that means you need to figure out `j=d` before you can consider the `e=` portion. Since `j` is an integer type, the value of 3.7 needs to be *coerced* into an integer before the assignment can take place.
An assignment operation has the assigned value as its return type, which means it returns 3, rather than 3.7.

So, why doesn't `c` round up? Same reason `j` didn't. A cast to an integer takes as much information as it can, and discards (or *truncates*) the rest.

Next up is some basic math.
Let's try adding a single line to the end of the main function in our last example:
```
std::cout<<i*d<<std::endl;
```

Before you compile and test it, what would you *expect* to happen? Keep in mind that multiplication requires matching *types*. So, will they both be integers, or both be doubles?
To risk giving away the answer, considering either outcome would be rational, it's probably best to not throw away additional precision, right?

Before you continue, try using the /, +, -, and % operators. That last one (modulus) can be especially interesting.

Step 2 – Literals

Since the next step will deal with bitwise operations, let's revisit literals.
We've been using Base 10 ('normal' numbers), but we don't have to. Consider the following example of binary, octal, hexadecimal, and decimal (base-10).

```
#include <iostream>
int main() {
	int i=0b00011000;
	int j=030;
	int k=0x18;
	int l=24;

	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
}
```

Unfortunately, you'll be expected to learn how these number systems work; especially binary and hex.
Fun fact: every nybble (character) in hex corresponds to exactly 4 bits in binary. Just make sure that you start from the least-significant (right) end leftwards (e.g. 0b10000 is 0x10, not 0x80).

Step 3 – Bitwise Operators

Refer to your preferred C++ reference (or at least the lecture slides) for a good sampling of available operators. For now, let's look at some of the bitwise operators. This will be useful not just for bit fields and masks, but also more generally for things like packing 24-bit colour, cleaning up byte streams, etc.

Let's start with an example of several of them:

```
#include <iostream>
int main() {
	int i=3>>1;
	int j=i<<1;
	int k=-1;
	int l=~k;
	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
	i=0xF; //corresponds to 00(lots of 0s)01111
	j=0b00001001; //It's actually more than 8 bits; it just adds zeroes
	k=i&j;
	l=i^j; //The last nybble is 0110, or 6
	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
	i=0;
	j=!i;
	k=~i;
	l=j==k;
	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
}
```

Note that the complement (~) operator isn't the same as a 'not' (`!`).
This is important, because both ~0 and !0 will yield values (-1 and 1, respectively) that will be logically treated as 'true', but not equal to each other (i.e. not the 'same true').
To demonstrate this, try adding:

```
	i=~0;
	j=!0;
	k=i&&j;
	l=i||j;
	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<"\ti==j: "<<(i==j)<<std::endl;
```

As a general rule, when you want to have variables storing *true* or *false*, you'll typically want to set them according to explicit logical tests (e.g. `i!=j`, rather than `i-j`; both of which would technically work).

Step 4 – Bitwise Operators: Revengeance

We've already referenced the concept of bit masks and bit fields. Let's just look into them a little bit more.
Suppose we set a number to 23. What's that in binary? (By the way, this is probably a good time to point out that you probably have a calculator program in Red Hat that can have its mode changed to *Programming*, which includes base conversions)
$23_{10}=00010111_2$ (i.e. 16+4+2+1)
Why is this useful? Because, in that single number (that's actually probably at least 32 bits long), you could argue that we're storing multiple boolean values: the zeroth, oneth, twoth, and fourth bit positions are true, while the remainders are all false.
But, how do we get that?

```
#include <iostream>
int main() {
	int i=23; //0b00010111
	int j=i&1;
	int k=i&0x02;
	int l=i&0b00010000;
	std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
}
```

Well. That's… bad. (Seriously, try running it. You get 1, 2, and 16 for some of the bit flags)

What's the problem? See Step 3: we want logical true/false.
We could just do this:
```
#include <iostream>
int main() {
        int i=23; //0b00010111
        int j=i>>0&1;
        int k=i>>1&1;
        int l=i>>4&1;
        std::cout<<"i: "<<i<<"\tj: "<<j<<"\tk: "<<k<<"\tl: "<<l<<std::endl;
}
```
But the need to use intermediate storage is unfortunate.
Perhaps we could put the bit shifts and masks directly into the `cout` line?

Try this:
```
#include <iostream>
int main() {
      int i=23; //0b00010111
      std::cout<<"i: "<<i<<"\ti[0]: "<<i>>0&1
            <<"\ti[1]: "<<i>>0&1<<"\ti[4]: "<<i>>4&1<<std::endl;
}
```
See how badly that fails? That is some high quality terrible code!
So, what's the problem? Well, the compiler's error messages tell the story: because of operator precedence, the *stream insertion* and *bitshift* operators are getting mixed-up.
However:
```
      std::cout<<"i: "<<i<<"\ti[0]: "<<(i>>0&1)
            <<"\ti[1]: "<<(i>>0&1)<<"\ti[4]: "<<(i>>4&1)<<std::endl;
```
works.

Note that, for most of the tests within Step 4 that deal with true/false values, we should probably be using the `bool` type, but that doesn't mean it would necessarily change the results.

Step 5 – Conditionals and Loops

Let's consider the following code:
```
#include <iostream>
int main() {
        int sum=0;
        int j=-2;
        for (int i=0;i<5;i++,j=i)
                do
                        sum++,++j;
                while (j<3);
        std::cout<<sum<<std::endl;
}
```
Don't run it right away! Instead, see if you can guess at the final value of `sum`.
It's okay if you can't (hopefully you're using paper, or at least a text editor, to work it out).
The point is largely to demonstrate why you need to write understandable loops, and ideally comment them.

Note that the post-test loop (do-while) will always run at least once. So even the iteration where i/j start at 4 will see an execution of sum++.

Of course, that isn't a common use for the comma operator. Having a single 'for' loop with two counters is more typical. e.g. for (int i=0,j=10;i<=j;i++,j/=2)

Naturally, loops are arguably just a more complicated relative of conditionals:

- Both rely on a boolean condition
- Both either result in a single statement or a block
- Both really just rely on jumps to another place in the program, based on that condition

Refer to the course slides (http://www.cosc.brocku.ca/~efoxwell/2P95/slides/Week02.pdf) for examples on 'if' and 'switch' statements.

To verify that you're following along, can you rewrite the following as a *switch*?

```
#include <iostream>
int main() {
	int val;
	std::cout<<"Enter a value between 1 and 10: ";
	std::cin>>val;
	if (val==1)
		std::cout<<"One! Aha ha!"<<std::endl;
	else if (val==2)
		std::cout<<"Two! Two! Aha ha!"<<std::endl;
	else if (val==3||val==4) {
		std::cout<<"Some! Some, aha ha!"<<std::endl;
		std::cout<<"Many! Aha ha!"<<std::endl;
	}
	else if (val==5)
		std::cout<<"Many! Aha ha!"<<std::endl;
	else
		std::cout<<"I don't think you understood the instructions."<<std::endl;
}
```

Step 6 – Ternary Conditional Operators

Most operators are *binary*, in that they accept two *operands*. Some are *unary*, and accept only one (e.g. ~).
However, there's a *ternary* operator, technically known as the ternary conditional operator, `?`.
How does it work? It's meant to be used as part of larger expressions. It can be used to return either of two choices, depending on the result of a boolean test. For example:

```
#include <iostream>
int main() {
	int j,k;
	std::cout<<"Enter j: ";std::cin>>j;
	std::cout<<"Enter k: ";std::cin>>k;
	std::cout<<"Personally, I'm a fan of "<<(j<k?"kayaking":j>k?"jaywalking":"sitting quietly")<<".\n";
}
```

This includes an example of *nested ternary conditional operators*. It's really not so bad to read, so long as you break it down into pieces. Ensure that you understand how this works before you continue (because you may need it for your exercise/assignment).

Submission

To demonstrate that you understand the content from this lab, do the following:
Write a program that contains an *unsigned char* variable, initialized to the hexadecimal value 9A.

Starting from the least-significant (rightmost/1's column), print all (8) bits from the variable, but with a caveat: print “one” and “zero” instead of *1*s and *0*s. (This will, of course, print the values in reverse sequence, and it doesn't matter if you include spaces in between the words or not)

Of course, your program must actually extract those binary values. That is, if the lab demonstrator chooses a different value and recompile, your output should change to reflect the new value.