**Coding Do's and Don’ts**

Most of the time our compilers are very helpful in teaching us how to write code, because of the error messages they give when we code something wrong. But sometimes they can teach us wrong because of how they forgive certain coding mistakes. It's not because there is something wrong with the compiler -- it's because there are variations in the implementation of the C++ standard among compilers. It's one reason why the American military came up with the ADA language -- a C++ type language, but with very specific standards so that it works the same on all compilers.

This course teaches how to write code that works on any system and compiler. Your compiler may forgive (and even encourage the use of) coding techniques that work only with that compiler. So be very careful to follow what's presented in the rest of this reading. It will be up to you to follow these coding rules, often without the help of your compiler that you've come to trust.

For-Loops

For-loop syntax allows this: for (int i = 0; i < n; i++). There are two semi-colons separating three parts of the statement.

**The first part** may declare and initialize a variable, like int i = 0. It may also initialize a variable that's already declared, like the i = 0 in this: for (i = 0; i < n; i++). You can even declare and initialize multiple variables, as long as they are of the same type, like this: for (int i = 0, j = 0; i < 100; i++, j++), or initialize already-declared variables of any type, like this: for (p = head, count = 0; count < n; count++, p = p->next).

But you absolutely cannot just put a variable name there without an equals to assign a value, like this: ~~for (~~**~~i~~**~~; i < n; i++)~~. Code::Blocks is one of the compilers that forgives this mistake.

**The middle part** never separates multiple logical tests with commas like the first and last parts do. So never do this:

~~for (int i = 0, j = 0;~~**~~i < 100, j < 100~~**~~; i++, j++)~~

Do this instead:

  for (int i = 0, j = 0; **i < 100 && j < 100**; i++, j++)  
                     ...or...  
  for (int i = 0, j = 0; **i < 100 || j < 100**; i++, j++)

Value-Returning Functions

Value returning functions must always end with a return -- and not written as if-return or else-if-return, but just a simple return statement, returning a value matching the function's return type, like this:

bool fun(...)  
{  
  ...  
  return true;  
}

Never end with logic like the following, because the 2nd else-if would never be false, so why even have it?!

bool fun(...)  
{  
  ...  
  if (x == 0) return true;  
  ~~else if (x != 0)~~return false;  
}

You could do a simple return false; above, or even else return false;. Compilers like to be told what to return in any possible logical way through them. We know that there's no possible way the above would ever reach the bottom of the function, because one of the two if's will be true. But compilers don't look inside the parentheses! They just see "if this, do this, else if that, do that". So their problem is -- what to do if it's neither this nor that. Some keep that to themselves, others complain.

String Library Includes

When the C++ data type "string" appears in any CPP (or H) file, you must #include <string>. But some compilers, most notably Apple XCode, forgives this oversight, if you've included iostream. That's because their implementation of the iostream library automatically includes the string library!

So especially if you're coding on a Mac, watch out for this one! Every time you find yourself typing "string", **stop**, scroll to the top of you code where the includes are, and include the string library!

Uninitialized Variables

The value assigned to int i; is what we call "unpredictable". Just like a box of chocolates, you never know what you're going to get. But some compilers, most notably Apple XCode, initializes uninitialized variable to their data type's default value -- zero for an int. But take your code to another system and compiler and it's probably not going to be zero anymore!

Pointer Values

You can set a pointer value to zero (0) or NULL. They are interchangeable. But never use NULL for anything other than pointers. It's a "macro" defined in every C or C++ library include, so it only works at all if there's at least one #include in your file.

You can use nullptr with C++11 extensions, too.

Extraneous Code

Including a library you don't use, or declaring a variable you don't use, adds inefficiency to the compiling and running of your code. But even more important than that in a course like this, it suggests that you don't exactly understand what you're doing! It can also misled those reading your code by making them think you're doing something that you're not -- kind of like introducing a character in a novel but never referring to them again.

If-statements that are always true, like the else-if clause in the "value-returning function" section above, is also extraneous -- no need to ask a question if the answer is known in advance.

Your work won't be returned to you for redo if it contains extraneous code. But it will be marked down.

Working In "Debug" Mode

If you're using an IDE like Visual Studio or Code::Blocks or XCode, you should see a menu-selectable option to work in debug mode or release mode. By default, these IDEs choose debug mode. But doing so can mask errors like overrunning array bounds. Only when you switch to release mode for delivering the final product will such programming error be revealed -- when they are very hard to diagnose.

Work in "release" mode, and only switch to "debug" mode temporarily when using step-by-step tracing.