Programming and Numerical Methods, class #2 (notes only)

Introduction

This class is going to cover some aspects of the compiled languages C and C++. I'm not going to attempt to teach *everything* about these languages in the classroom. I will attempt to teach you *enough to get things done*, unsystematically at first, then more systematically in the next 3 classes. If you want to know *more* about C/C++, you should consider getting a good book. What I am going to do first is go over basic things to know when writing your own programs for scientific or engineering purposes.

Various kinds of computer languages

Why not just program directly in machine code all the time? You'll have some idea after doing the exercises from the last class. Here are three good reasons: (1) If you followed the suggestion and relocated the multiply subroutine to a different address, you'll have noticed you had to recode the jump instructions. (2) In writing the program to test the multiplication subroutine, you'll have noticed you need to be careful how you use registers; in any non-trivial program, you run out of registers and use memory locations to store values. (3) In general, when working with machine code, you tend to spend a lot more time thinking about what the CPU is doing than thinking about what you're trying to accomplish.

- **Assembly language** lets you program using opcode mnemonics and symbolic names for memory addresses in your code and the memory you use for data. This takes care of (1) and makes (2) much easier – the assembler rewrites all the jump and memory-accessing instructions for you. But if you program in assembly language directly, you still spend the same amount of time thinking about what the CPU is doing.
- **Higher level languages** let you program using symbolic names for variables without worrying about where to put them in memory, and with more compact and direct (for a human) ways of directing the computations to be done, such as algebraic expressions. Imperative programming languages consist of instructions of what operations to do in what order, but in a more human-oriented fashion. There's a whole taxonomy of other types of high-level languages that we don't need to go into.

First C program: output, subroutines

Chances are you've seen this example:

```
#include <stdio.h>
int main() {
  printf(“Hello, World!\n”);
  return 0;
}
```

Or this one:

```
#include <iostream>
int main() {
  std::cout << “Hello, World!\n”;
}
```

Exercise 1: compiling “Hello, World!”

Get the C/C++ compiler for your computer installed (if it isn't already), make a text file with either of the above programs in it, and try compiling it into a working program. A few tips:

- The first “hello world” program is the same in C or C++, so you can name the file `hello.c` or `hello.cc` and compile it either way. The second only works in C++.
- Use GCC (gcc) for linux or Mac. The command to compile a C program is `gcc`, and the command to compile a C++ program is `g++`.
- Use Visual C for Windows. The command to compile either C or C++ is `cl`. It uses the file extension of the input to decide which it's compiling. (Actually gcc does this too.)

Some Terminology:

- Source code: the text file(s) with your program written in the high-level language of your choice. Use extension .c for C code, .cc for C++ code, .f or .for for Fortran, etc.
- Assembly code: assembly language text that can be processed by an assembler.
- Object code: binary data containing machine instructions and data plus symbol table information and relocation data for the linker (more about that later).
- Binary executable code: machine instructions that can be loaded and executed by the operating system. (Let's just assume you know what an operating system is, ok?)

Seeing what the compiler does

Your compiler may produce object code or executable code by default. When it does this, it's actually doing two or three steps: converting your source code into machine instructions (its main job), producing an object file, and calling the linker to make the executable. Compilers generally have an option to stop at the first stage and emit assembly language.

Exercise 2: seeing what the compiler does

Compile your program with the “emit assembly language” option and compare the assembly code to the source code.

- The option is “-S” for gcc and “/FA” for Visual C++.

What is a linker for?

There are some interesting things to notice about the assembly code produced for the “Hello World” program. For one, it refers to the address of a subroutine by the symbol `printf`, but never defines `printf`. For another, it defines `main` as the address at which the program starts, which is itself a subroutine, but it's not clear who calls `main`. The linker puts your object code together with object code for the `printf` function copied from a standard system library and a little bit of code in a standard place that just calls `main`. In systems that have “shared libraries” or “dynamically linkable libraries”, the linker might instead only add the information on where to find `printf`, rather than including the code for `printf` itself: in this case, the operating system writes the correct addresses into the call instructions when the program is loaded, or something equivalent.

What's a library?

Just a collection of object code all packaged together into one file.

Algebraic expressions, operators, and operator precedence

In the expression `1+2*3-4`, the symbols + and - and * are *binary operators* for addition, subtraction, and multiplication. (Binary means they use two values, one on the left and one on the right.) In the expression `(-5)`, the minus sign is the *unary* operator for negation.

C and C++ are very definite in how they apply operators. For example, `1+2*3-4` means (1+(2*3))-4 and not (1+2)*(3-4), and `1+2-3+4` means ((1+2)-3)+4 and not 1+(2-(3+4)). Each operator has a precedence level that says how to group operations when deciding between two operators of different precedence, and an associativity (left-right or right-left) that says how to group operators when deciding between two of the same precedence. This is almost always, *but not quite always*, the same as you'd expect from what you learned in algebra.

Level	Operator	Description	Associativity
3	`+ -`	unary sign operators	Right-to-left
6	`* / %`	multiplicative	Left-to-right
7	`+ -`	additive	Left-to-right
8	`<< >>`	shift	Left-to-right
9	`< > <= >=`	relational	Left-to-right
10	`== !=`	equality	Left-to-right
11	`&`	bitwise AND	Left-to-right
12	`^`	bitwise XOR	Left-to-right
13	`|`	bitwise OR	Left-to-right
14	`&&`	logical AND	Left-to-right
15	`||`	logical OR	Left-to-right
16	`?:`	conditional	Right-to-left
17	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment	Right-to-left
18	`,`	comma	Left-to-right

I've omitted a few non-mathematical operators for simplicity. Here are a few more points to keep in mind:

- The comparison operators (==, !=, <, >, <=, >=) compare two values and evaluate to “true” or “false”. In C, “true” is just a 1, and “false” is just a 0. C++ also treats “true” and “false” as if they were the integers 1 or 0 if you use them as if they were numbers.
- Don't confuse the comparison operator == with the assignment operator =. The latter is used to put a value into a variable. (More on this later.)
- The logical “and” and “or” operators are && and ||, and they evaluate to “true” or “false” just like the comparison operators.
- The <<, >>, &, ^, and | operators operate on the bits of integer values. (However, << and >> are also used for something else in C++, more on this later.)

Exercise 3: operator precedence

Consider the following C/C++ expressions and figure out what they really do. What are their equivalent mathematical expressions?

```
exp(-E/k*T)
 1 < 2 < 3
-1 < 0 < 0.5
```

Variables

Variables in imperative languages are just labels for *mutable* storage locations. In other words, setting a variable `x` equal to one simply means storing a 1 in the corresponding memory location. Later, your program can store any other value into `x`. This is very useful when using a variable as an accumulator, like when calculating a sum, or for index variables. But otherwise, it's best practice to let variables mean only one thing as much as possible.

Good example	**Bad example**

`sum=0.0;`
`for (i=0; i<n; i++)`
`  sum= sum + x[i];`
`mean= sum/n;`
	`x=1.0;  // position of ball`
	`printf(“Ball is at %f\n”,x);`
	`x=2.0;  // position of bat`
	`printf(“Bat is at %f\n”,x);`

The assignment operator “=” is used to put a value into a variable.

Basic Types

Character types	`char`
	`signed char`
	`unsigned char`
	`wchar_t`

Boolean type	`bool`

Integer type	`short`
	`int`
	`long`
	`unsigned short`
	`unsigned int`
	`unsigned long`

Floating point types	`float`
	`double`
	`long double`

Void type	`void`

Important points:

- The number of bits in a short, int, or long is not standardized! It might be that an int and a long are both 32bits, or that an int is 16 bits and a long is 32 bits, or that an int is 32 bits and a long is

64 bits. Similar comments apply to float, double, and long double.

- Every variable has to have a specified type in C and C++, before it is used. This is done in the variable declaration, e.g.,
  ```
  double x;  // a declaration in C++
  ```
  This is different from languages with a “define by assign” approach in which the first assignment to a variable defines what it is.
- Once a variable is declared to have a type, it can't be redeclared to be a different type within the same block of code. This is different from “dynamically typed” languages such as Python and Perl.
- New types can be derived and defined using these types, including arrays, structures, and classes. More about structures and classes later.

Arrays

Arrays are declared in C and C++ by putting the size of the array in square brackets after the name of the variable, like this:

```
double x[25];  // array of 25 doubles
char name[32]; // array of 32 characters
```

The individual elements of an array are accessed by putting any mathematical expression that evaluates to an integer inside square brackets after the variable name, like in the “Good example” in the Variables section above.

Exercise 4: “Hello World” with input; array overruns

So far, we've seen at least something of how to produce output (Exercise 1, using the pre-defined variable `cout` and the `<<` operator.), how to do computations (algebraic expressions, operators), and how to store and retrieve values in memory (variables). That's 3 out of 4 of the things we said a computer had to have, the remaining one being a way of getting input. One way to do this in C++ is the pre-defined `cin` variable and the >> operator. Try this:

```
#include <iostream>
int main() {
  char quest[16];
  char name[32];
  std::cout << “What is your name?\n”;
  std::cin >> name;
  std::cout << “What is your quest?\n”;
  std::cin >> quest;
  std::cout << “Hello, ” << name << “!\n”;
  std::cout << “Good luck as you “ << quest <<”!\n”;
}
```

Try running the program entering your name as one word, less than 32 characters, and your “quest” as one word, less than 16 characters.

Then try entering two words for the first input.

Then try entering one word longer than 32 characters for the first input and one word longer than 16 for the second.

Notes:

- The >> operator for a character array only reads characters up to the first white space. In most cases, C and C++ I/O is whitespace delimited. There is a special function named `gets` which reads characters up to the first newline character, but it's better to use the `getline` function defined for C++ strings described below.
- Overruns of fixed length arrays are a frequently occurring problem. They're one of the most common causes of security vulnerabilities in software, and they're also a source of errors in programs written by scientists and engineers intended only for themselves.

Vectors and Strings

C++ provides a way to define variable length arrays, which are called `vectors` for some reason. The syntax looks like this:

```
vector<double> x;  // vector of doubles, zero length
x.resize(100);     // make the vector have 100 elements
```

There are other ways of adding elements to a vector that we will talk about later. The individual elements of a vector are accessed in exactly the same way as an array.

A special type is defined for variable-length arrays of characters that should be treated as text, also known as character strings. The syntax looks like this:

```
string s;
s= “Hello, World”;
```

Many operations are defined for text strings, such as “addition” (concatenation) of two strings, input (>>) and output (<<), and many functions such as search-and-replace, insertion, deletion, etc. One very nice function is getline, which can be used like this:

```
getline(cin, s);
```

A few more points:

- In order to use `vector`s, you have to have the line
  `#include <vector>`
  in your program before you define any function using a `vector`.
- Similarly, to use `string`s, you have to have the line
  `#include <string>`
  in your program.
- You actually have to put `std::` in front of the words vector and string everywhere, unless you add the line
  `using namespace std;`
  in your code first. This can be done either before or inside the body of your function.
- The above features are part of the Standard Template Library (STL). There is very nice documentation for the STL [STLProgGuide].

Exercise 5: Rewrite Example 4 using the C++ string class

First try just replacing the arrays of chars with strings.

Then try using the `getline()` function so that you can read multiple words in your “name” and “quest” answers.

Try the same tests on this program that you tried on Exercise 4.

Assignment: write a program to convert Fahrenheit to Celsius

Requirements: It should have input and output, it should use two floating point variables (one for the input and Farenheit and one for the converted value in Celsius), it should compile and run without errors, and it should do the calculation correctly.

References

[STLProgGuide] Hewlett-Packard Company and Silicon Graphics International, Standard Template Library Programmer's Guide, Release 3.3, June 8, 2000, 2000; http://www.sgi.com/tech/stl/ browsed May 17, 2010.