Introduction to Programming and C++ Fundamentals

Scripting Languages

Last week I talked about the different kinds of programming languages, well there are other kinds of programming languages known as scripting languages.

Some of these don't fit into the strict definition of a programming language, languages such as HTML. They are not used to code standalone applications that perform specific tasks.

There are other scripting languages such as perl are used to code standalone applications that perform specific tasks.

Many scripting languages emerged as tools for executing one-off tasks, particularly in system administration. One way of looking at scripts is as "glue" that puts several components together; thus they are widely used for creating graphical user interfaces or executing a series of commands that might otherwise have to be entered interactively through keyboard

So what is the difference between a scripting language and a more traditional language?
The boundary between scripting languages and regular programming languages tends to be vague, and is blurring ever more with the emergence of new languages and integrations in this fast-changing area.
One of the main difference is that languages such as C++ tend to be compiled where as scripting languages tend to be interpreted. Not that this is not always the case, there are always exceptions.

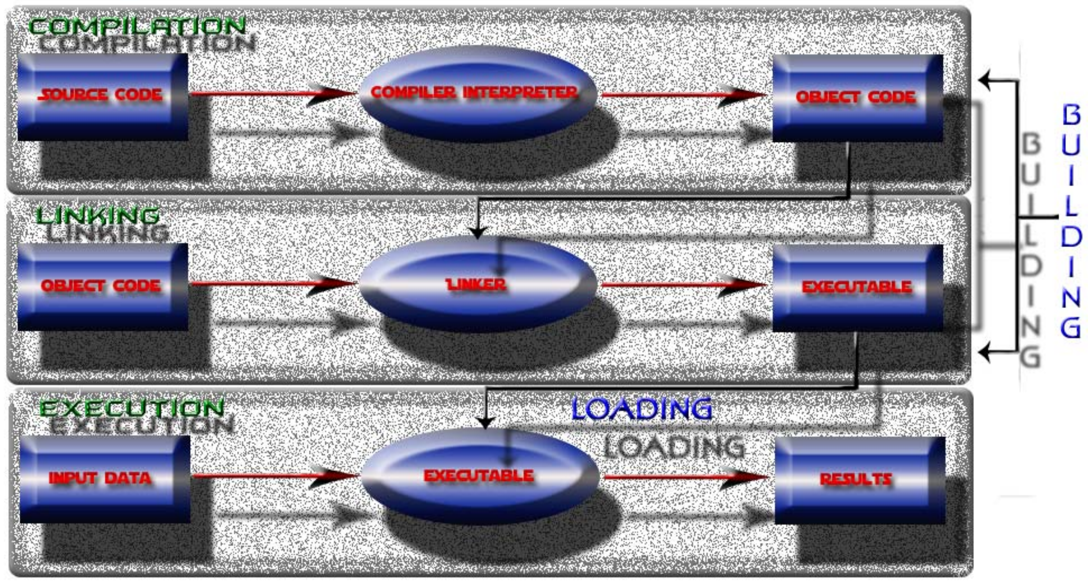
Interpretation vs. Compilation

Compilation is the process of taking our high level language code, converting into machine readable code and the producing a compiled executable.

Although the exact process by which an executable file is produced differs slightly depending on the compiler, most follow those indicated in the steps below:

Source Code: High level language instructions (C++) written by the programmer

Object Code: Low level machine instructions readable by the hardware



<u>Programming and Algorithms</u>

Last we I compared programming to giving directions in a very explicit way. This still holds true, how ever I want to expand on that.

“Computer programming (often simply programming) is the craft of implementing one or more interrelated abstract algorithms using a particular programming language to produce a concrete computer program”.

Sounds difficult, but it isn’t that difficult – essentially you can look at the algorithms as the directions.

You will often see the term algorithm used to refer to a program or part of a program. From this you can determine that algorithms are inherent parts of computer programming, but what is an algorithm?

The term algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. In computing it is often seen as a computable set of steps to achieve a desired result.

It was defined by Zwass as a “finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems”
It has also been defined as: A finite and precise description of a general method formulated within a fixed language.
The elementary constituents of an algorithm are executable processing steps.

Examples of algorithms

- How to add two decimal numbers p and q
- How to test if a number is prime
- How to compute the number e = 2, 1782...
- How to sort a new card into card index
- How to sort a card index

If you are thinking already that algorithms are pretty complex, well you’re in for a surprise as we have barely even scratched the surface – although we don’t need to learn much more. There are different classifications or algorithm and there is practically a science dedicated to algorithm analysis, so lets be glad we don’t have to do that.

A computer program can often be viewed as an elaborate algorithm, although computer programs themselves may be made up of many algorithms. For our course when using mathematics and computer science, an algorithm will usually mean a small procedure that solves a recurrent problem.

To make a computer do anything, you have to write a computer program. To write a computer program, you have to tell the computer, step by step, exactly what you want it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal. In this way we can view the algorithms as directions.

Let's say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:

The taxi algorithm:
Go to the taxi stand.
Get in a taxi.
Give the driver my address.

The call-me algorithm:
When your plane arrives, call my cell phone.
Meet me outside baggage claim.

The rent-a-car algorithm:
Take the shuttle to the rental car place.

Rent a car.
Follow the directions to get to my house.

The bus algorithm:
Outside baggage claim, catch bus number 70.
Transfer to bus 14 on Main Street.
Get off on Elm street.
Walk two blocks north to my house.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

Designing Algorithms
Programming will often involve problem solving by means of implementing and sometimes designing an algorithm.
When designing an algorithm, even a basic one you should make sure it possesses the following qualities.

Properties
Algorithms should have the following properties:

- Finiteness – Algorithm must complete after a finite number of instructions have been executed
- Absence of Ambiguity – Each step must be clearly defined having only one interpretation
- Definition of Sequence – Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted
- Feasibility – All instructions must be able to be performed. Illegal operations (e.g. division by 0) are not allowed.
- Input – 0 or more data values
- Output – 1 or more results

The Need for Design

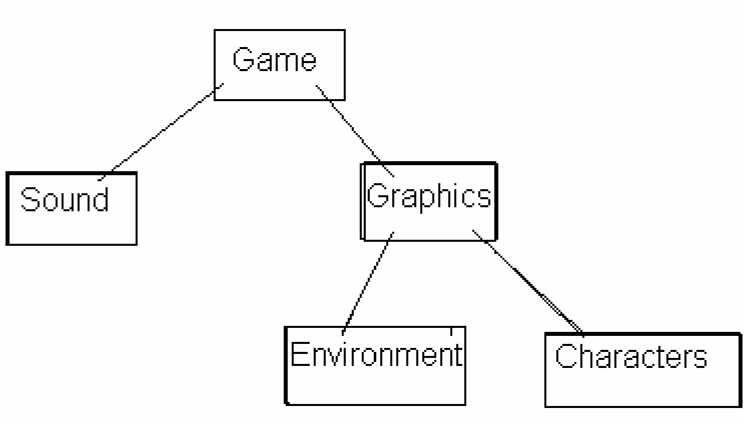
This leads us on nicely to the need for design when programming.
When a builder begins to build a house he doesn't just pick up a hammer and go. The house must be properly designed first.
This is the same with programming, it is best to design a solution before you begin. This sounds really good; in reality we won't be doing much design in programming 1. This is because our programs are going to be relatively short, it how ever is important to learn about these issues in programming.

Program design is also big business with people getting paid lots of money to design solutions.

There are 3 steps you should perform when you have to write a program:
1) Define the output
2) Develop the logic to get that output
3) Write the program

Define the Output
In the context of programming 1 this will be easy, I will often present you with a problem presented in the form of the necessary output or the output will be relatively simple. In terms of large scale programming such as that necessary for a game it isn't that simple, a computer game has a variety of outputs in the form of user stimulus.

The most often used and cited method of program design is top down – essentially top down design reverses the tree so we start with one trunk and get more braches as we go down.



A very crude example – but its only meant to give an idea.

Develop the Logic
In this stage you decide how you are going to solve the various stages, what additional programs, how it's going to be coded, decide what decisions the program needs to take and maybe even some algorithm design. In big companies they often use pseudo-code to develop the logic.

Write the program
Finally you generate the code. Which is the bit this course is mainly concerned with.

Pseudo-code

Some of you may have heard of pseudo-code and some of you may hear about it and wonder what it is. We aren't going to be doing much in pseudo-code, we are going to be principally focusing on actual coding, however it is good to know exactly what pseudo-code is.

Essentially pseudo-code is a generic way of describing an algorithm without use of any specific programming language-related notations. It is, as the name suggests, pseudo code. That is, it is not really programming code, but it models and may even look like programming code.

The algorithm should give you an idea of how you actually generate your own solutions to problems. This one aims to work out the square root of a number given by the user, the input.

In the computing business technology evolves rapidly and it can seem that almost every three months (six months if you are lucky) that the programming languages that are available, including everything from Assembly through to C++, Visual Basic through to the scripting languages like Perl and PHP, all move along at such a break neck pace that keeping yourself up to date can be a real chore.

Every time you turn around you have to work with a new set of specifications, different syntax, sometimes different languages (or new versions so far removed from their predecessors as to seem almost entirely new). In this race of progress the success of a programmer can often be measured by the quality of their code (later on we will look at good coding standards as well, good coding practices greatly improve the quality of your code). Chunks of reusable code, blocks of logical processing - anything that cuts down development time is a valuable asset. With this in mind, bring on a new (and very old) player: pseudo-code.

- Used a great deal in industry and business.
- Used to document business processes, instructions to staff, user requirements for systems, etc.
- Use pseudo code to help you understand a problem, map out as solution, and design your algorithm.
- When you have done this, you should be able to write your C++ program to achieve the required results.

So what is pseudo code, well basically Pseudo-code is the "plain English" explanation of a piece of code or algorithm which does something in a tidy way (say a self-contained function).

Consider the following example, the first bit is written in C++ the next bit pseudo-code.

```
// check if age is equal to or greater than old
if(Age => Old)
{
     cout << “You’re Old \n”;
}
else
{
     cout << “You’re Young \n”;
}
```

```
if Age is greater than Old

output "You’re Old!"

else

output " You’re Young "

end if
```

Pseudocode Issues
The one problem with pseudocode is that its style varies from author to author, luckily there is usually an accompanying introduction explaining the syntax used.

Programming Syntax, Semantics and Definitions

Ok, know that we have looked at the background lets look at some definitions that apply directly to programming.

Assignment = in computer programming, an assignment is the defining of a variable; this will be covered in more detail later on.

Statement = in computer science, a statement is an instruction, string, or other arrangement of symbols; often grouped into a statement block. In programming languages, a statement is a set of declarations, or a step within a sequence of actions. Statements contain declarations, definitions, expressions, and procedural statements

Statement Block = in computer programming, a statement block is a section of code which is grouped together, much like a paragraph; such blocks consist of one, or more, statements. In C, C++, and some other languages, statement blocks are enclosed by braces {}.

Syntax and Semantics

Syntax = (grammar) rules that specify how valid instructions (constructs) are written

Refers to the spelling and grammar of a programming language. Computers are inflexible machines that understand what you type only if you type it in the exact form that the computer expects. The expected form is called the syntax.

Each programming language defines its own syntactical rules that control which words the computer understands, which combinations of words are meaningful, and what punctuation is necessary.

Semantics = rules that specify the meaning of syntactically valid instructions

In general, semantics is the study of meaning, in some sense of that term. Semantics is often opposed to syntax, in which case the former pertains to what something means while the latter pertains to the formal structure/patterns in which something is expressed

For instance in the following statement (written in pseudo-code):

```
    a-value = expression;
```

Where a-value is something like a variable, whose value can be changed, and expression is a valid C++ statement is an example of the syntax of an assignment statement.

The semantic rule for an assignment statement specifies that the value on the right hand side is stored in the `a-value` on the left hand side.

C++ Words

There are three types of words: Language-defined (called Reserved words), System-defined (called library words in your text), and user-defined words or identifiers.

- Reserved words are predefined in the language (C++) and cannot be used for anything other than the purpose for which they are reserved.
- Examples: int, while, if, for, template
- Reserved words and library words together are known as keywords.

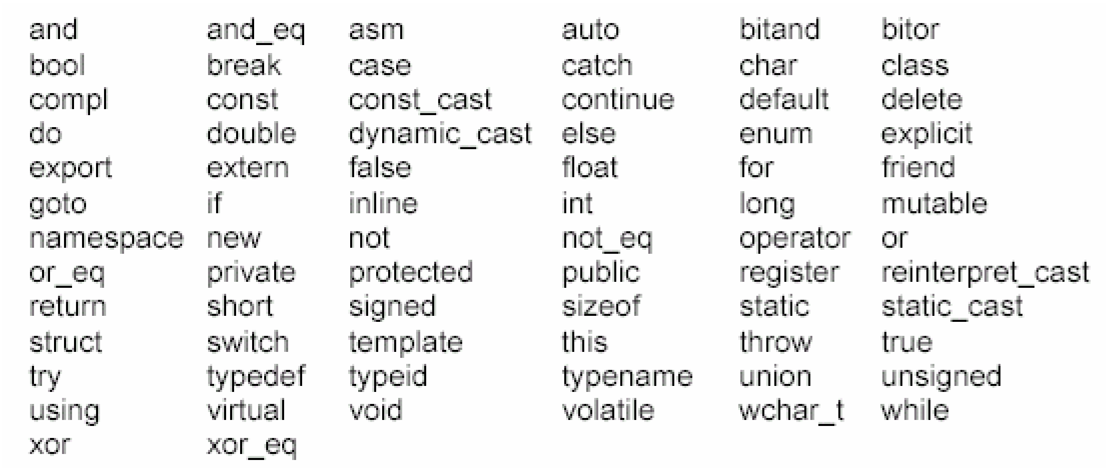
Reserved Words
Every language has words that you can only use in the way the designers of the language say they can be used. These are reserved words.

In the context of the C++ language these are specified its official vocabulary.

The reserved words (and reserved symbols as well) are pre-empted by the C++ language definition and may only be used in the manner the language rules specify.

The number of reserved words, comparatively to the English language, is relatively small, < 300.

Here is an example of some of the reserved words:



and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while
xor	xor_eq				

If you wish to look at all the reserved words then there are books that list them all. There may be a website as well, although I haven't found it.
In Visual Studio, and most text editors, reserved words are coloured blue.

Library Words
Beginning C++ programmers are sometimes confused by the difference between the two terms reserved word and predefined identifier (library word), and certainly there is some potential for confusion.

One of the difficulties is that some keywords that one might "expect" to be reserved words just are not. The keyword `main` is a prime example, and others include things like the `endl` manipulator and other keywords from the vast collection of C++ libraries.

For example, you could declare a variable called `main` inside your `main` function, initialize it, and then print out its value (but you probably shouldn't, except as an experiment to verify that you can!).

Hopefully you remember from last week what our main looked like:

```
int main()
{
     std::cout << "Hello World! \n";

     return 0;
}
```

On the other hand, you could not do this with a variable named else. The difference is that else is a reserved word, while main is "only" a predefined identifier.
In Visual Studio using these predefined identifiers will sometimes generate errors or warnings.

Here is a very short list of some of the predefined identifiers:

cin	endl	std	iomanip	main	npos
cout	include	string	iostream	MAX_RAND	NULL

First Program

```
/* Hello World C++ Program */

#include <iostream>

int main()
{
  std::cout << " Hello World! \n";

  return 0;
}
```

Examining the Code

Lets examine the elements that make up the program:

`#include` – Preprocessor (or include) directive that tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the cout statements that appear in our code). The header file "iostream" contains basic information about this library.
The # (hash) symbol is the signal to the preprocessor to modify the code, the modified code includes the file specified after the `#include`, in this case `iostream`.

Because the program is short, it is easily packaged up into a single list of program statements and commands.

```
int main()
{
  std::cout << " Hello World! \n";
  return 0;
}
```

All C++ programs have this basic "top-level" structure. Notice that each statement in the body of the program ends with a semicolon. In a well-designed large program, many of these statements will include references or calls to sub-programs, listed after the main program or in a separate file. These sub-programs have roughly the same outline structure as the program here, but there is always exactly one such structure called main.

`int main()` – This tells the compiler that there is a function named main and that this function returns an int….notice the 'int' bit.
When the program is started `main()` is called automatically, and execution begins from the first statement in function main.
Every c++ program needs a `main()`.
The ({ brackets (parenthesis) signal the beginning of function parameters and code blocks, the) } brackets signal the end of function parameters and code blocks.

`std::cout << " Hello World! \n"; -` This line of the program may seem strange. If you have programmed in another language, you might expect that print would be the function used to display text. In C++, however, the `cout` function is used to display text. It uses the << symbols, known as insertion operators.
The quotes tell the compiler that you want to output the literal string (hello world) as-is. The `\n` means 'new-line'.
The semicolon is added onto the end of all function calls, declarations and statements in C++, we'll come to variable declaration later.

`return 0; -` means "return the value 0 to the computer's operating system to signal that the program has completed successfully". More generally, return statements signal that the particular sub-program has finished, and return a value, along with the flow of control, to the program level above.
When you return a values from a function you return what data type was declared at the beginning of the function declaration, in our case `int` (from the function `int main()`), to the where the function was called. In the case of `main` the return value is passed on to the operating system.
The last curly bracket closes off the function.