1. The objectives of the subject

This first year subject is about Programming where programming is seen as a total, creative process beginning with a problem statement and passing through several stages until finally there exists a fully documented and evaluated solution to the problem. (Not a solution to some other problem!).

The objectives of the course are that students will be able to:

1. structure solutions to problems for execution by computer
2. use a microcomputer efficiently and effectively in developing total solutions
3. develop and express their solutions using good programming style
4. express their solutions in well-structured programs written in C++

To achieve these objectives you will need to learn a number of things, some of which you may already have some idea about and others which will be new to you.

You will need to:

Learn the jargon and how to use it correctly. Since the beginnings of computers those working in the area have developed new words, many of them acronyms, or have taken existing English words and phrases and adapted them to have related but subtly different meanings from their everyday use.

COBOL, byte, BASIC, strings, packages, records, WYSIWYG and even *ROTFL* are just some examples.

Extend your problem solving skills answers won't always leap out at you you may have to use skills that you didn't know you had.

Extend your writing skills so that you can communicate effectively and efficiently. Too much of our daily communication is imprecise (though probably meaningful to those for whom it is intended *Yawanna go out ? Yeah? G'day, Seeyuh there then).*

Learn some of the computer programming language *C++*.

Learn the elements of good programming practice and good programming style.

1.1 Why do we need a computer language?

At the start of the development of computers (only about 80 years ago) humans had to communicate with the machines initially by hard wiring and the setting on hundreds of switches. This progressed to the use of numeric internal codes which the computer used (and still uses) to control the electronic circuits. Then low level languages were developed which provided textual equivalents to the numeric codes.

Humans decided that strings of zeros and ones were difficult for them to understand and so developed, over time, higher level languages. These were languages which were closer to everyday English and which computer programs called compilers could translate into the necessary machine code.

Another reason for a computer language is that English (of all the natural languages) is, as any of you who have tried to learn it as a second language will know, very imprecise. Every time you think you have mastered a rule or concept, you come across an exception.

Consider pronunciation and spelling:
cough (*koff),* bough (bow), *enough* (enuff), and *although* (altho) or even simpler *to, too* and *two.*
It is possible to argue that according to the rules of English GHOTI should spell *fish* but lets leave that as an exercise in futility.

English also relies very heavily on context and practice to determine what any set of words really means. Let's look at an example.

Here are three statements, each with a context-dependent meaning of **it**.

*Go to the table where the telephone is ringing and pick **it** up.*
*Go to the mountain with the hut on top and sleep in **it**.*
*Go to the mountain with the hut on top and climb **it**.*

This sort of problem has plagued natural language translators for years. There are apocryphal stories about translation by computer where common words and phrases are translated into a foreign language and then retranslated to English. Supposedly the phrase: *out of sight, out of mind* became *invisible, insane.*

So there is a real need to have a much more limited language governed by strict rules of **semantics** and **syntax** to unambiguously describe to the compiler just what the human programmer requires.

1.2 Why C++ ?

Back in the late sixties, Ken Thompson at AT&Ts Bell Labs was developing a new operating system for computers called UNIX. He first implemented it on a PDP-7 in assembly language (the low-level language with text-based command codes), but then wanted to transfer it to a more powerful machine, a PDP-11. Unfortunately, PDP-7 assembly language was not the same as that for the new machine. If every time UNIX was to be implemented on a new machine the thousands of lines of code were to require rewriting in a different assembly language UNIX was going to be held back. So Thompson began searching for a high level language in which to rewrite UNIX.

In 1970, because no language took his fancy, Thompson designed a new language called B, based on an existing language called BCPL. By 1972, it was apparent that B was inadequate for implementing UNIX. A Bell Labs colleague, Dennis Ritchie, designed a successor to B,

which he called C. 90% of UNIX was now written in C.

This new language became very popular in colleges and universities all over the world and eventually spread even into the business area. Now that inexpensive C compilers are available on microcomputers, C has become the language of choice for most PC application implementation.

Although C is very powerful, it has some problems. It requires a level of discipline beyond the typical fledgling programmer. It is nearly 50 years old and things change. Until early 1990s, most university computer science courses introduced students to programming via so-called teaching languages such as Pascal and Modula-2, moving on to C later. This makes sound sense as all these languages have the classic programming language Algol as a common antecedent.

A recent approach to program design called object-oriented programming has become popular over the last few years. Another researcher at Bells Labs, Bjarne Stroustrup, has added these new concepts to C, along with eliminating many of the difficulties beginning programmers have with C. This language was later to be called **C++**. (You will see later where the ++ comes from.)

Although intended to be an object-oriented language, C++ can be used as a superset of C and that's how this subject will introduce programming. In this way it is anticipated that the one major language will be used throughout the undergraduate degree.

2. Some fundamental concepts of computing

The electronic computer is a relatively new device having its beginnings in the 1930's. Charles Babbage in the 19th Century developed his difference engine and his analytical engine to calculate range tables for the British Admiralty to control the aiming of naval guns. It was he who developed the five essential elements which an automatic calculator needed. These are INPUT, OUTPUT, STORAGE, PROCESS and CONTROL

It is also interesting to note that Ada Lovelace (Lord Byron's daughter) was responsible for programming Babbage's machines, developing many of the concepts of programming now in use. The false idea that computing is a man's world is certainly not based on historical fact.

Babbage's logical elements can be applied to all problem-solving situations, including humans. Consider this:

I will say three numbers.
Add the first two together and subtract the third.

For example

5, 6, 7
Answer is 4

What was the INPUT?
There was two parts – the instructions and the data.
How did you receive the INPUT?
Through the senses.

Where did you STORE the instructions?
In your brain. You may have temporarily stored them on paper but you had to store them in your brain before you could obey them.

Where did you STORE the data?
Again, in your brain or temporarily on paper.

What did the CONTROLLING?
You had to control the loading of instructions to your brain, the loading of the data as you went along, the processes you used and the storing of the partial and final results. This control was exercised by the brain.

What did the PROCESSING?
The adding and subtracting was done by the brain.

And, the element most often forgotten, the OUTPUT. How did you communicate the result?
You wrote it down or you spoke it out. Unless you provide some output nobody knows whether you have done anything or not.

All of these five logical elements are present in all computers:

INPUT provided by devices like keyboard, mouse, touch panel, scanner and trackball.

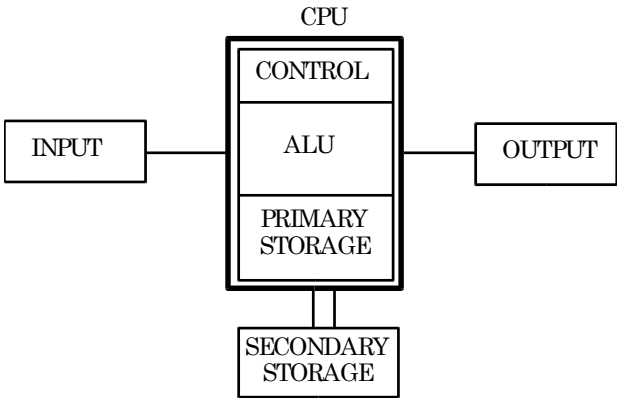
OUTPUT provided by screen, printer, plotter.

PROCESS provided by the Arithmetic Logical Unit in the Central Processor Unit

CONTROL provided by the CPU

STORAGE comes in two types just like you used in the example above. TEMPORARY (Primary) storage in the computers MEMORY (like your brain) and PERMANENT (Secondary) storage on various media such as magnetic disks, tapes, drums, optical disks and many other media.

The programmers view of the hardware can be represented by a block diagram.
The PRIMARY STORAGE can be thought of as a sequence of locations, each of which has a unique address but may have variable contents. When something is written to memory it overwrites any existing content of the location a destructive write but when something is read from memory only a copy of the contents is taken the original content remains unchanged a (non-destructive) read.



As we have already discussed, a high-level programming language allows us to express commands or instructions unambiguously.

TOTAL = PRINCIPAL + INTEREST

This is a valid C++ instruction which contains:

three labels TOTAL, PRINCIPAL, INTEREST

two operators = and +

When translated into machine language the command becomes:

Label a location in memory TOTAL, another PRINCIPAL and another INTEREST.
Take a copy of the contents of the location labelled PRINCIPAL and ADD its value to the contents of the location called PRINCIPAL.
ASSIGN (store) the result in the location called TOTAL.

What happens if I say to you 7, 8, 9? Someone will say "6" because you have remembered the instructions I gave earlier and you have applied them to another set of data. You have been programmed! You could easily remember the instructions and kept them separate from the data.

At machine level, the set of instructions to be obeyed (the PROGRAM) and the information on which the actions are to be carried out (the DATA) look exactly the same. That is to say they are simply strings of 0's and 1's.

For convenience, since machine level operates on patterns of binary digits, the term **bit** was coined standing for a binary digit.

To obtain larger groupings for useful operations the term **byte** was coined to represent a collection of bits sufficient for storing the range of different characters which humans needed to represent inside the computer. On most computers, including the Macintosh, a byte is 8 bits.

Computer memory is often considered to be made up of a number of **words** each of which can be handled by the computer as a single item. Typically words are 8, 16, 32 and 64 bits long. There are other possible word sizes used.

2.1 Representation of numbers

When dealing with numbers (remembering that was what computers were first created to handle) we need to understand a few mathematical concepts. Mathematicians differentiate between real numbers and integers.

5 is an integer
5.0 is a real

There are differences between these two sets of numbers in mathematical terms and there are physical differences between the way they are stored and represented within a computer.

Here is a 16-bit representation of the integer five: 0000000000000101
Each binary digit, read from right to left, represents a power of 2.
Thus 5 is made up of $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + \ldots$
or $1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 0 \cdot 8 + \ldots$

To save time and space lets just look at 4-bit representation of numbers

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110

7 = 0111

2.1.1 Representation of negative integers

How can we represent negative integers ?
If we replace all of the 0's by 1's and all of the 1's by 0's in the representation of the positive number, we get a possible representation of the corresponding negative number.

–0 = 1111
–1 = 1110
–2 = 1101
–3 = 1100
–4 = 1011
–5 = 1010
–6 = 1001
–7 = 1000

This is called **ones complement** because, in the negative version, we write down the opposite character (or the complement) to that used in the positive version. This means, among other things, that there are two ways to represent zero, namely +0 = 0000 and –0 = 1111.

Look at the trip meter or odometer on a car or bike (the instrument that tells you how far you have travelled). Suppose it only has 4 digits. When it gets to 0000, what was displayed one kilometre before? That is, what is one less than 0000? The instrument would have registered 9999. (Remember that the odometer is a base 10 device.)

+1 = 0001
0 = 0000
–1 = 9999

Under this scheme 9999 is the tens complement of 1 which is obtained by subtracting 1 from zero.

In a binary system, in which the base is two, we can also represent negative numbers using the **twos complement** of the positive value.

–1 = 1111
–2 = 1110
–3 = 1101
–4 = 1100
–5 = 1011
–6 = 1010
–7 = 1001

So, in summary

1's Complement
Replace each 0 in the positive number by a 1 and each 1 by a 0

+7	0111	–7	1000
+6	0110	–6	1001
+5	0101	–5	1010
+4	0100	–4	1011
+3	0011	–3	1100
+2	0010	–2	1101
+1	0001	–1	1110
+0	0000	–0	1111

2's Complement
Subtract the positive value from 0.

		–8	1000
+7	0111	–7	1001
+6	0110	–6	1010
+5	0101	–5	1011
+4	0100	–4	1100
+3	0011	–3	1101
+2	0010	–2	1110
+1	0001	–1	1111
+0	0000		

Some find it easier to calculate the ones complement and then add 1 to it rather than subtract the number from 0 to get the twos complement directly. Most computers store negative numbers in twos complement form.

If you look at the patterns that have been generated, you should notice that the positive numbers all have a leading bit which is 0 and all the negative numbers have a leading bit which is 1. This is the conventional way to represent positive and negative numbers. It also means that there is only one zero and it's "positive".

Let's now look at some bigger numbers.
Consider the binary number
0001010010101010 = 2 + 8 + 32 + 128 + 1024 + 4096 = +5290

Using ones complement, –5290 should be
1110101101010101

Using two's complement, we can just add 1 to the answer we just got, giving
1110101101010110

Whoops. We've just done some binary arithmetic without explaining how it's done. It's just the same as base ten except that
adding 1 to 0 gives 1
adding 1 to 1 gives 0 and 1 to carry.

There is yet a third way of assigning a bit pattern to the representation of negative integers.

Sign and Magnitude
Reserve the top (left-most) bit for a sign bit – 0 for positive and 1 for negative. For example, when using 16 bits, the top bit is the **sign** and the other 15 bits are the **modulus** or magnitude. For example
0001010010101010 is still +5290
but now
1001010010101010 is now –5290.

2.1.2 Use of octal and hexadecimal

Youve probably noticed the tediousness of writing binary representations just lots and lots of ones and zeros. It is traditional in computing to write internal computer representations in a higher base by combining together 3 or 4 bits at a time.

Octal (base eight) uses three bits and the digits 0 to 7. Thus, our earlier 16-bit example
0001010010101010 which is +5290 in decimal can be written as
0 001 010 010 101 010 (breaking into 3s from the right) and becomes
0 1 2 2 5 2
012252 in octal.

Hexadecimal (base 16) collects four bits at a time using the digits 0 to 9 and A,B,C,D,E and F (either upper or lower case). So 5290 now can be written as
0001 0100 1010 1010
or 1 4 A A in hex.
You will find that most texts and programming manuals, as well as dumps of internal representations provided for debugging purposes, use hexadecimal.

2.1.3 Representation of reals

The representation of real numbers is more complicated. Lets start by looking at the similarities between decimal reals and binary reals.
Consider
1 2 3 . 4 5 6
= $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$

For a binary real
1 0 1 . 0 1 0 1
= $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$
= $5\ 5/16$

To visualize how reals are represented inside a computer, imagine that computers represent reals in decimal form rather than binary. (IBM once made a machine which did this – the IBM1620).

Any decimal real can be expressed in a standard form which has a 0 before the decimal point, a decimal fraction, and a power of ten.
1 2 3 . 4 5 6 becomes $0 . 1 2 3 4 5 6 \cdot 10^3$
0 . 0 0 7 8 9 becomes $0 . 7 8 9 \cdot 10^{-2}$
The decimal fraction is called the **mantissa** and the power of ten is the **exponent.** Large numbers have positive exponents and small numbers have negative exponents. Since the number is **always** represented in a standard form we do not have to record where the decimal point is, that the whole number part is 0 or that the power of the multiplier is ten. We need only store the **fraction part** and the **exponent.**
Note that the fraction part always begins with a non-zero digit.

Suppose we allocate two decimal digits to represent the exponent part thus allowing one hundred values from 00 to 99. We have to be able to represent large and small numbers so we need to cater for positive and negative exponents.
Pick a number in our allowable range near the middle – 50 – and let it represent an exponent of 0.
53 would then stand for an exponent of +3
50 would stand for an exponent of 0
48 would stand for an exponent of –2.

This gives us a range of values of 10^{-50} to 10^{49}. (Where is 0?)
1 2 3 . 4 5 6
$0 . 1 2 3 4 5 6 \cdot 10^3$
stores as
+53123456

0 . 0 0 7 8 9
$0 . 7 8 9 \cdot 10^{-2}$
stores as
+48789000

Binary reals are stored in a similar way. There is an **exponent** field, a **fraction** field and a leading **sign bit**. Now the implied exponent base is of course **2**. But this time well make sure the bit before the binary point is a 1, and then do not store it. (We get more digits than if we always need to store the leading 1.)

5.0 = 1 0 1 . 0 0 0 · 2^0
= 1 0 . 1 0 0 0 · 2^1
= 1 . 0 1 0 0 0 · 2^2

This is stored as

0 10000001 01000000000000000000

Sign	exponent	fraction
1 bit	8 bits	23 bits

or

01000001001000000000000000000000

If the exponent occupying 8 bits were treated purely as the magnitude of the number and hence always positive, then values from 00000000 (0) to 11111111 (255) are possible. Choosing a mid-value of 127 in a similar way that 50 is half way between 00 and 99 in the decimal case, we get positive and negative exponents.

So 255 represented as 11111111 stands for an exponent of +128
127 represented by 01111111 stands for an exponent of 0
0 represented by 00000000 stands for an exponent of –127

This means, for example, that an exponent of +3 is represented by 127 + 3 or 10000010 (130) and an exponent of –3 is represented by 127 – 3 or 01111100 (124)

The range of values representable is then 2^{-127} to 2^{128} which is about 10^{-38} to 10^{38}.

So the internal representation of sign (s), exponent (e) and fraction (f)

s	e	f

represents the number

$(-1)^s \cdot 2^{e-127} \cdot 1.f$

Practically all of the time, programmers write reals in decimal form using E notation. For example, the number 123.4 can be written as $1.234 \cdot 10^2$ or 1.234e2. Similarly, 0.0001234 can be written 1.234e–4 (or 123.4e–6 or 0.1234e–3,. . .)

One of the problems of thinking in decimal and storing in binary is the fact that many numbers we consider simple decimals cannot be exactly represented in binary.

For example, 0.3 is a simple decimal number. What is it in binary?

0.3 = 0.25 + 0.05 .01?????
= 0.25 + 0.03125 + 0.01875 .01001??
= 0.25 + 0.03125 + 0.015625 + 0.003125 .010011?
and so on

In fact, unless a decimal can be multiplied by a power of 2 and yield an integer, it is not exactly representable as a binary real much in the same way as repeating decimals are in base 10.

To further understand the way numbers are stored in a computer, we need to recognise other limitations.

2.1.4 Sizes of stored numbers

We mentioned earlier that computers store information in bits, collected together into bytes and finally into words. So how many bytes or words do integers and reals occupy? Well, it varies form computer to computer and language to language. But there exist standards.

16-bit integers can store positive numbers from 0000000000000000 (0) to 1111111111111111 ($2^{16} – 1$ or 65535). But as soon as we want to store negative ones, the biggest positive becomes $2^{15} – 1$ or 32767. The negative number of greatest magnitude depends on whether we are using ones or twos complement. The limit is –32767 and –32768 respectively. (The simplest way of remembering which is which is that ones complement has two zeros and hence one less negative number.)

32-bit integers can store up to $2^{31} – 1$ or 2147483647. Let's stick to twos complement and say the negative limit is –2147483648.

32-bit reals use an 8-bit exponent and a 24-bit mantissa (including sign bit) and hence could have a multiplier from 2^{-127} to 2^{128}. However, the standard (IEEE 754) reserves the maximum and minimum for other purposes. So the actual range is 2^{-126} to 2^{127} still about 10^{-38} to 10^{38}. But what does the mantissa limit mean? If all the mantissa were zeros then the number represented would be

$1.00000000000000000000000 \cdot 2^{\text{exponent}}$

If all the bits were 1 then we would have

$1.11111111111111111111111 \cdot 2^{\text{exponent}}$

In decimal notation, this range of binary reals corresponds to a number with 5 to 7 significant decimal digits.

Note that it is impossible to represent zero as a binary number in the form 1.something times a power of 2. This is where the standard uses the exponent –127 or 00000000. In fact, zero is represented as all zeros exactly the same as a 32-bit integer zero!
The other exponent (all ones) is used to represent special values such as +INF (+∞) and –INF (–∞) and NAN (not a number).
Often, more precision is needed. That is 7 significant digits are not accurate enough. So we use more bits of mantissa.

64-bit reals use a sign bit, 11 bits of exponent and 52 bits of mantissa magnitude giving about 16 significant decimal digits and a range of 10^{-308} to 10^{308}.

Other representations include 48-bit, 80-bit, 96-bit and even 128-bit reals, each providing certain precision (significant digits) and range (exponent values).

2.1.5 Character representations

Obviously, we want to communicate with a computer using more than just numbers. We need to be able to talk symbols including alphabetic characters in upper and lower case, punctuation marks and other symbols.
The ASCII (American Standard Code for Information Interchange) standard uses binary bit patterns stored in a byte to represent 128 different characters.

Alphabetic characters
Both upper and lower case are provided, using the bit patterns corresponding to the octal values 101 (65) to 132 (90) for A to Z and 141 (97) to 172 (122). So, for example, the bit pattern 01000110 which is 106 in octal, 70 decimal represents the letter F. Space is code 40 octal.

Numbers
As characters, the digits 0 to 9 are represented by byte values 60 to 71 octal, 48 to 57 decimal.

Assorted symbols

$	44 (octal)
+	53
	55
*	52
=	75
/	57
?	77
.	56

and so on.

In fact, the characters we normally need are all represented by the 96 byte values from 40 octal to 176 octal all with uppermost bit 0.
Those characters below 40 octal (and 177 octal) are called control characters and include return, tab, backspace, line feed, and many other special characters.
Those above 177 octal (with uppermost bit 1), not part of the ASCII standard set, are called the extended character set and are used by many computers to display foreign characters such as é, ô, ü, and Greek characters such as π and μ.

2.1.6 Summary

Thus, given the contents of 32 bits of computer memory, it can be interpreted as a binary bit pattern, an integer, a real or 4 characters (or ?).
For example,

01000110011100100110010101100100

can be written as

10634462544 in octal
46726564 in hex

and can be interpreted as

1181902180	an integer
1.551335e4	a real
or "Fred"	a string of four characters.

It all depends on the program.

2.2 Basic Computer Concepts

As mentioned earlier, early computers had to be instructed to perform tasks using **machine instructions**. These were bit patterns which the machine could interpret as an order to do some simple task such as loading a value into its accumulator, adding a number to the accumulator or storing the value from the accumulator into a memory location. Let's look at a very simple model of a computer.

The Central Processing Unit (CPU) contains an Arithmetic Logic Unit (ALU) and four fast access memory locations called registers. These are:

Accumulator	holds the result of operations
Program Counter	holds the location of the next instruction
Instruction Register	holds the next instruction to be decoded and executed
Data Register	holds the location of the operand

The computer has a two-part cycle called the **fetch/execute** cycle. During the fetch part of the cycle, the next instruction is fetched from memory, loaded into the Instruction Register and decoded. The OpCode (the code for the operation to be executed) stays in the Instruction Register while the address of the operand is placed in the Data Register.

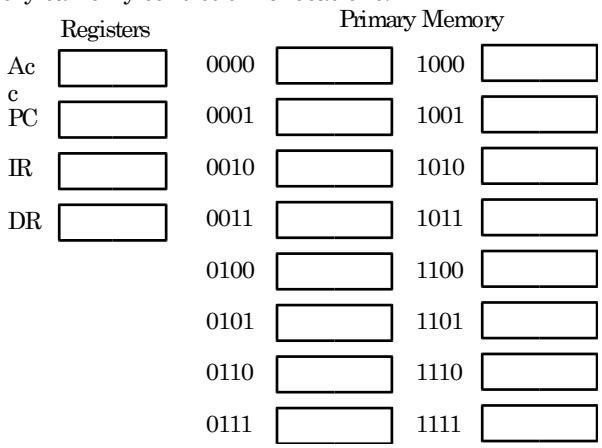
During the execute part of the cycle, the instruction is sent to the ALU where it is carried out and when this is finished the Program Counter is incremented to the location of the next instruction.

This simple machine has only a very simple machine instruction set allowing trivial operations to be carried out. Each instruction is made up of 6 bits: a 2 bit OpCode and a 4 bit address where the operand can be found.

The four possible instructions are

`00`	`xxxx` STOP	(the address is ignored)
`01`	`aaaa` LOAD	put the data at the location given into the accumulator
`10`	`bbbb` ADD	add the data at the location to the value in the accumulator
`11`	`cccc` STORE	store the value in the accumulator in the location given

Note that with only 4 bits of address, memory can only consist of 16 locations.

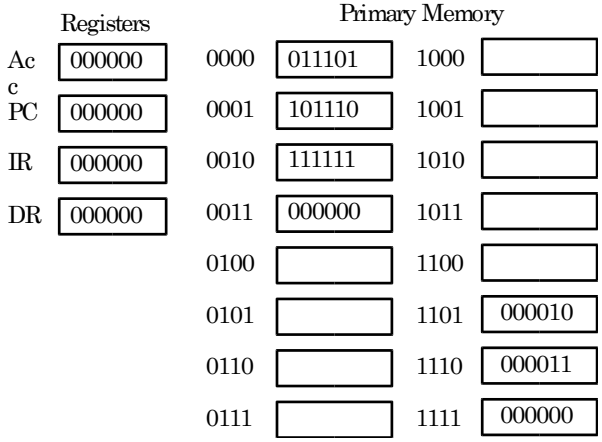


Each memory location has physical **address** which is fixed, but its **contents** can be changed.

When a value is read from a location, no change occurs to the contents of that location a copy of the contents is taken. This is a (non-destructive) **read**.

When a value is written to a location, the new value replaces whatever was there before a (destructive) **write**.

Let's load memory with some binary patterns. Instructions and data are usually loaded into different sections of memory. We'll put program at the bottom and data at the top. Of course there is in fact no difference in instructions and data both are just bit patterns. It's what the computer does with the information that makes the difference.



All of the registers have been set to 000000 representing 0.
Now we can look at the fetch/execute cycle as the program is told to run or execute.

Cycle 1:
Fetch
Program Counter (PC) is pointing at location 0000 so the instruction there is copied into the Instruction Register (IR), **decoded** as OpCode 01 (LOAD) and the address of the operand 1101 is copied into the Data Register (DR).
Execute
The LOAD instruction is sent to the ALU and the **contents** of the memory location held in the DR is placed into the Accumulator (Acc)

which now shows 000010 (decimal 2).
The PC is now incremented to point at 000001, the location of the next instruction.

Cycle 2:
Fetch
PC pointing at 0001 so the instruction there is copied into IR, **decoded** as OpCode 10 (ADD) and the address of the operand 1110 is copied into the DR.
Execute
The ADD instruction is sent to the ALU where the **contents** of the memory location held in DR is added to the contents of Acc which would now hold the sum of 000010 and 000011 which is 000101 (5).
The PC is incremented to 000010.

Cycle 3:
Fetch
PC pointing at 0010 so the instruction there is copied into IR, **decoded** as OpCode 11 (STORE) and the address of the operand 1111 is copied into the DR.
Execute
The STORE instruction is sent to the ALU where the **contents** of Acc (000101) is written into the memory location held in DR so that it also holds 000101 (5). The contents of Acc are unchanged.
The PC is incremented to 000011.

Cycle 4:
Fetch
PC pointing at 0011 so the instruction there is copied into IR, **decoded** as OpCode 00 (STOP) and the address of the operand is ignored.
Execute
The STOP instruction is sent to the ALU where execution of the program is terminated. What happens to the contents of the registers at this time depends on implementation.

2.3 High Level Languages

Programming in machine language is difficult. High level languages were developed so that humans did not need to write complex binary patterns which the computer could understand. Instead, they could write instructions in a form which they could understand and a program could be written which could do the translation to machine code for them.

Like most high-level languages C++ has clearly defined rules of **syntax** (governing the allowable arrangement of symbols) and **semantics** (governing the meaning of elements). The point has already been made that programming languages are designed to be unambiguous (unlike English). For this to occur, the rules must be obeyed. Provided you abide by the rules then there will be no mistakes between what you **intend** and what the compiler produces.

The compiler processes the source code written in C++ character-by-character and matches what it finds against the syntactic rules of the language. If there are no errors, the compiler generates machine code to perform the tasks. Usually, some high-level operations are supported by existing compiled code called **libraries** which are **linked** to the user's program at this stage. Finally, the programmer can ask the computer to run the program.

3. Introduction to Programming

This course introduces the concepts of program design and implementation. The chosen high-level language is C++. There is no specified platform for running your programs, although the laboratory provided contains Dell computers using the Linux operating system. Students can carry out the programming assignments on any machine they wish. However, submitted assignments must not use **any** constructs peculiar to a particular platform.

Descriptions of compiling and running programs during this subject refer to the compiler g++ on Linux.

It is assumed that, by the time this point has been reached in the subject, the student is familiar with the basic operations of Linux regarding:

- using zip disks
- creating directories
- copying files
- printing

3.1 Our first C++ program

Let's look at our first example C++ program to convert Fahrenheit temperatures to Celsius temperatures. There exists a simple **equation** aiding in the conversion:

$$F–32 = 9\ C / 5$$

This says that if we subtract 32 from the temperature F in Fahrenheit, we get the same value as multiplying the equivalent Celsius temperature C by 9 and dividing by 5. Note the use of the symbol = to represent the words is equal to.

To convert the equation into a **formula** we must make the required value the sole entity on one side of the equation. Thus

$$C = 5\ (F–32) / 9$$

is a formula for determining C from F. This can be read as:

If we have the value F for temperature in degrees Fahrenheit, then subtracting 32, followed by multiplying by 5 and dividing by 9, will yield the temperature C in degrees Celsius.

Heres the C++ program to perform the task:

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
     float F, C;

     cout << "Enter Fahrenheit temperature to convert: ";
     cin >> F;
     C = 5.*(F-32.)/9.;
     cout << F << " in Fahrenheit is " << C << " in Celsius\n";
     return 0;
}
```

Probably the first thing you will notice is the formula written as

```
     C = 5.*(F-32.)/9.;
```

This is merely C++ representation of the formula with some syntax requirements such as the necessity of specifying an operator (*) for multiplication.

Here's that same program in a slightly different form:

```
#include <iostream>
using namespace std;int main(int argc, char* argv[]){float F,C;cout<<
"Enter Fahrenheit temperature to convert: ";cin>>F;C=5.*(F-32.)/9.;cout<<F<<
" in Fahrenheit is "<<C<<" in Celsius\n";return 0;}
```

Which program do you prefer? They are in fact the same program! All that has changed is the removal of 'unnecessary' **white space**. That is, any blanks, tabs and new lines. Any C++ compiler ignores unnecessary white space – but we can't.

Spaces, new lines and tabs make programs **readable**. Programs may be specifically for a compiler to convert to code that the computer can understand. But people must be able to read and understand a program so that we are in control of the programming process.

What if there is an error in a program like the one above? A compiler will tell us where such a bug may be. But we can avoid the need for

such debugging by following certain rules when entering a program into the computer. These rules – not a requirement of the language syntax – are usually collected into what is called **programming style**.

Style can be very personal. That is, each programmer can develop their own individual way of providing readability. As a beginning programmer, you can learn style by initially following someone else's style. Once you have mastered the syntax of a language, you may then develop your own style.

As you read different texts you will find each author may have slight variations in style. In this course, we will propose a certain style. You should follow these guidelines.

3.2 Programming Style

The first rule to follow is:

"Short programs with little white space are unreadable"

Each line of program should be readable. That is, another programmer should be able to determine quickly just what is meant by each line. If a line is too complicated and cannot reasonably be simplified, add a **comment** to explain in English what is going on. Try this:

```
#include <iostream>  // this defines all input output
using namespace std; // this makes sure we're using the right names

int main(int argc, char* argv[])              // the start of the program proper
{
     float F,   // a variable to hold the Fahrenheit temperature
           C;   // a variable to hold the Celsius temperature
/* the following line prints out a prompt */
     cout << "Please enter Fahrenheit temperature to convert: ";
     cin >> F;  // here we receive the requested value of F
     C = 5.*(F-32.)/9.;   // here's the calculation
/* now we output the F and C values */
     cout << F << " in Fahrenheit is " << C << " in Celsius\n";
     return 0;  // we're finished
}
```

This is clearer? Of course not. This is overkill. A comment is required when you want to explain something that is not obvious. Remember that a person who doesn't know C++ is not going to be helped by an unnecessary comment. We can't see the programming for the comments.

We've now discussed our first (and perhaps most important) C++ construct the comment.
Whenever a C++ compiler sees a comment, it ignores whole sequences of text.

There are two syntax forms for a C++ comment.

Form 1:

any sequence of characters between `/*` and `*/`

This includes comments extended over many lines. For example, it is common practice to place some form of comment near the beginning of each program which explains the purpose of the program (and the author). We could incorporate into our program the following:

```
/*
     This program requests a temperature in degrees Fahrenheit
     calculates the equivalent temperature in degrees Celsius
     and prints out the two temperatures.
     Written by Peter Castle.
*/
```

Form 2:

any sequence of characters between `//` and the end of a line

For example

```
     C = 5.*(F-32.)/9.;   // here's the calculation
```

The end of a line is determined by pressing the return key while typing in the program.

Again, do not overdo comments. Following other components of style will avoid the necessity for many explanations you may at this time think necessary.

Another part of good style is the use of blanks (or spaces). Separating certain symbols from each other by white space is **required** by the syntax of C++. Other times, it just improves the readability. For example, the space in

```
     float F,C;
```

between the `t` and the `F` is so that the compiler can recognise the word `float` from the symbol `F`. However, if we put a space after comma following the `F` such as

```
float F, C;
```

then we are simply using style. As the syntax of C++ is introduced, the required spaces and style spaces will be identified.

Should sequences of spaces be used, or if we want certain parts of consecutive lines to align themselves, we can use the **tab** key. This causes the characters typed after the tab to start at pre-defined **tab stops**. Where the tab stops are set are also part of the style. Linux defaults to 8-character positions.

Returning to our program:

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
      float F, C;

      cout << "Enter Fahrenheit temperature to convert: ";
      cin >> F;
      C = 5.*(F-32.)/9.;
      cout << F << " in Fahrenheit is " << C << " in Celsius\n";
      return 0;
}
```

Note that the lines between the pair of braces `{` and `}` are indented by one **tab**, and the braces are aligned. This is style.

Here is some more style.

The choice of names for entities which you select within your program mean little to the compiler. (The names we use for entities are called **identifiers**.) However, they have **a major part** to play in documenting your code so that you at a later date and anyone else who reads the program, can easily understand what the program does.

1. Choose names for all of those things which you need to identify which are meaningful to you and which give some insight into what they are meant to be.
2. Balance readability with the need to minimise typing. If in doubt use a longer rather than a shorter name. In general between 3 and 10 characters should be sufficient.

   In our example, the symbols `C` and `F` are perhaps too short. But are they preferable to calling them `Celsius` and `Fahrenheit`?

   (`Celsius = 5.*(Fahrenheit-32.)/9.;`)

   Perhaps `degC` and `degF` are suitable.
3. No symbol should be expected to do more than one job in any program. Whenever a symbol appears in a program, the **thing** it represents should **always** be the same. This does **not** mean that the **value** will never change.
4. Use comments freely to clarify the intention but balance this with the use of meaningful identifiers. It is stupid to have a statement like

   ```
   answer = number + INCREMENT;
   ```

   followed by a comment such as

   ```
   // add the increment to the number and store in answer
   ```
5. Use blank lines to clearly delineate sections of the program.
6. Use indentation to show the structure of the program and to place similar items in alignment.

3.3 Finding syntax errors

Whenever you write a program, there is no guarantee that it will work first time. No matter how careful you are when entering program code there will be mistakes. Some of these will be *blunders*, that is to say you knew what you wanted to type but pressed the wrong key. Others will be *errors* in that you pressed the key you wanted but it was an incorrect key. The compiler doesn't care **why** or **how** you made the mistake. It will simply report that the error exists.

Programming languages have very strict syntax rules and it is these rules that the compiler applies to your source code. A good compiler will provide you with something more than a simple error number which you have to look up in a reference manual.

When you attempt to compile a program in g++ nothing much happens, until you get the prompt back. This means that it has compiled OK.

This doesn't mean that there were **no** errors in the code – simply that none were detected.

If there are syntax errors in the source code g++ will describe the errors it found and on what lines of the program they occurred. Often later errors are due to the first one anyway. And frequently the error reported is not actually the error – just that the compiler encountered something wrong at that point (probably caused by something earlier).
Some common errors include

Mistyping an identifier either by misspelling it or by using different case letters from that in other occurrences of the same symbol.

Using incorrect punctuation such as ; instead of , or leaving out punctuation.

Forgetting vital spaces.

The diagram at right illustrates the steps involved in implementing a program.

So far, we've discussed the steps Think, Edit and Compile. Once you can compile your source code without error, the next stage proceeds. The compiled program can then be executed. Here is where other bugs can be encountered logic **errors**. That is, the program does what you said not what you meant. A program is only successful when it performs the task for which it was written. If it doesn't then the program contain errors of design. That is, the steps required to be performed are not performed by the instructions provided in the program. The process of running the program, finding logic errors and correcting them is called **debugging**.

4. Algorithm Design

Consider the following problem:
What is the numerically largest and numerically smallest telephone number in the alphabetical subscriber section of the Wollongong White Pages?

How do we start? How can we be sure we don't miss any number and that we find the smallest and the largest numbers? What is the set of rules we must follow? What is the **algorithm** we need to use? (What is an algorithm?)

Open the telephone book at the first page of the listing.

Take two sheets of paper and write LARGEST on one and SMALLEST on the other.

Look at the first number in the list.
It is the largest seen so far. Write it down on the paper headed LARGEST.
It is the smallest seen so far. Write it down on the paper headed SMALLEST.

LOOP
Read the next telephone number.

Compare it with the number on LARGEST.
If the number we are reading is larger than the largest seen so far then cross off the existing one and write the new number on LARGEST.

Compare it with the number on SMALLEST.
If the number we are reading is smaller than the smallest seen so far then cross off the existing one and write the new number on SMALLEST.

Continue the LOOP until we reach the end of the listings.

The two numbers written on the paper are the largest and smallest numbers in the book.

(AHA! So an algorithm is an English description of the steps **we** have to follow to perform the task.)

Question: What is always the number written on the paper headed LARGEST?
Answer: The largest number seen so far even when we haven't read any number from the telephone book.

Question: What is always the number written on the paper headed SMALLEST?
Answer: The smallest number seen so far even when we haven't read any number from the telephone book.

Question: Do we always have to do the second test to see if the number is smaller than the SMALLEST?
Answer: NO. Obviously, if the number is larger than the largest seen so far it can't be smaller than the smallest seen so far so we are wasting time checking the second case if the first case is true.

We can save ourselves some time by changing the LOOP as follows:

LOOP
Read the next telephone number.

Compare it with the number on LARGEST.
If the number we are reading is larger than the largest seen so far then cross off the existing one and write the new number on LARGEST.

OTHERWISE
Compare it with the number on SMALLEST
If the number we are reading is smaller than the smallest seen so far then cross off the existing one and write the new number on SMALLEST.

Continue the LOOP until we reach the end of the listings.

Question: How will we know when there are no more numbers in the listing?
Answer: As humans we can see when there are no more. When we have reached the section in the book marked by a page headed Using Your Telephone Service which tells us there are no more numbers.

If we wanted to translate this algorithm into C++ to do the task for us we would need to know when to stop.

One way is to count up the number of telephone numbers in the book and tell the program to stop when it has processed that many telephone numbers. Another possibility is to pick a value which cannot possibly be a valid telephone number (such as zero or a negative number) and use this as the end of data marker. We will deal with different ways of terminating loops later. For now we will use a zero or negative number as the end of data marker.

Question: Is this zero or negative number part of the data to be processed?
Answer: NO. We chose it because it can't be a valid data item and so it must not be processed. If it were processed it would automatically be the SMALLEST.

Let us now look at a C++ program to carry out this task for us using the algorithm developed above:

```
#include <iostream>
using namespace std;
/*
     Program designed to read a set of integers and to find
     the largest and smallest numbers in the set. The program
     is to stop when it read a non-positive number. There
     will always be at least one positive number in the data.
    Written by Peter Castle.
*/

int main(int argc, char* argv[])
{
     int number, maximum, minimum;

     cout << "Type in integers (<=0 to end)\n";
     cin >> number;
     maximum = number;
     minimum = number;
     while(number > 0)
     {
          if (number > maximum)
               maximum = number;
          else if (number < minimum)
               minimum = number;
          cin >> number;
     }
     cout << "Largest number found was " << maximum << endl;
     cout << "Smallest number found was " << minimum << endl;
     return 0;
}
```

The program loops around until a certain condition is met which will terminate the program. Before entering the loop, the first number is read in. We can do this because the specifications included the fact that there would always be at least one valid data item to be read. The basic structure of the program is:

READ the first number
WHILE the number is valid
    PROCESS the number
    READ the next number
WRITE the results

Part of the processing requires us to check to see whether or not the number is bigger than the BIGGEST NUMBER READ SO FAR and if it is to update maximum. If it is not, we need to check to see whether or not the current number is smaller than the SMALLEST NUMBER READ SO FAR and if it is to update minimum.

To enable us to do so we need to initialise both maximum and minimum to some values. The program sets them to the first number read in.

Some people would tend to pick an abnormally large value and set minimum to this and pick an abnormally small value and set maximum to that thus ensuring (?) that any numbers read in will alter these values. Since we set maximum abnormally low surely some value read in will be bigger than it? And similarly for minimum.

There are some problems with this latter approach. Firstly, how will we **know for sure** that the numbers we pick will work?

Secondly, this approach breaches the rule that a variable **always** reports the same thing. Why? Because, taking maximum as the example, it is supposed to contain the largest number in the data set, or the BIGGEST NUMBER READ SO FAR. If we arbitrarily allocate a value to it, there is a time when it does **not** contain the largest number in the set in fact it may contain a number that does not belong to the set at all.

It's about time to start learning some C++.