Let's start using the C++ components learnt so far to put together a program.

The simplest program consists of a main function (we'll see why it's a function later).

Most C++ IDE provide a skeleton of this program as follows:

```
#include <iostream.h>


int main(int argc, char* argv[])
{
```


```
    return 0;
}
```

For the moment the skeleton will remain a 'magic container'.

We've already met some C++ statements.

In fact, the most popular C++ statement in programs is the **expression statement** formed by taking an expression and terminating the statement with a semi-colon.

Here are some examples.

```
degF = 23.5;

Initial = 'P';

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

To convert 100ºF to Celsius, we could use the following two statements

```
degF = 100.;
degC = 5.*(degF-32.)/9.;
```

In fact this is the simplest statement:

```
;
```

called the null (or empty) statement.

Another statement already covered is the **declaration statement**.

We declare to C++ the names of variables and constants we want to use in the program and indicate the type of these entities.

C++ then knows how to provide memory locations for these identifiers.

If we include in our program

```
float degC, degF;
```

C++ knows we are using the names `degC` and `degF` to store variable values of type `float`.

Otherwise, C++ would not know what to do with these symbols when encountered.

Statements can be collected together into **blocks** called **compound statements** which can be treated as if they were just one statement.

A compound statement is a set of statements preceded by

```
{
```

and followed by

```
}
```

The two braces are usually aligned with each other – this is style.

The statements within the block are usually indented by one level (one tab) – again style.

Our magic container exhibits one such block which forms the **body** of the main program.

```
#include <iostream.h>

int main()
{
    float F, C;

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

It's great being able to calculate solutions to a formula, but we need to get the answer out of the computer.

We need output.

C++ does not actually provide output (or input) as part of the language.

(C doesn't either.)

C provides a library called **standard I/O** or `stdio` for this purpose.

C++ can also use `stdio`, but the most common form of C++ I/O is **stream I/O**.

Here we introduce just enough of stream I/O to get by, and leave the rest till later.

Stream I/O allows access to the keyboard and the screen via objects called **streams**.

There are three standard streams (pre-defined and automatically available)

`cin` for input

`cout` for output

`cerr` another output stream

Let's look at output first.

A stream is a sequence of characters.

`cout` guides characters that we provide it to the screen.

We have to provide the characters.

We provide the characters (in an indirect way) to `cout` in the form of the types we've already encountered.

For example, if we want to put the string

```
Here is the answer
```

on the screen, we give it to `cout` as

```
cout << "Here is the answer";
```

`<<`

is the operator used to assign information to `cout`.

We can send more than one entity to `cout` by multiple assignments

```
cout << degF << " in degrees Fahrenheit is "
   << degC << " in Celsius";
```

Now a reminder re multi-line statements.

```
cout << degF << " in degrees Fahrenheit is "
     << degC << " in Celsius";
```

contains whitespace (tabs, spaces and newlines) which are ignored by the compiler – except the ones within the strings.

So it's the ; that determines the end of the statement – not the newline.

The operator << converts (through code) any non-character type, such as `float` variables `degF` and `degC` to a sequence of characters describing the value stored at the memory location referred to by the variable name.

So if `degF` contained the value 100.0, we would expect the library to generate the characters `100.0` in sequence and pass these to `cout`.

```
cout << degF << " in degrees Fahrenheit is "
     << degC << " in Celsius";
```

Note the use of spaces within the constant strings.

Without them, the values would run together.

The stream library does not incorporate spaces into the conversion of numerical values to sequences of characters.

For example,

```
float x = 1.3;
int y = 2;

cout << x << y;
```

would result in output something like

```
1.32
```

Note also that multiple cout statements can be used to generate single lines of output. For example

```
cout << "The answer is ";
x = 3+7;
cout << x;
cout << endl;
```

produces

```
The answer is 10
```

Sometimes we want to separate the output by more than spaces, especially putting output on separate lines.

We use the newline character `\n` inside a string constant, or the keyword `endl`.

```
cout << "The first line.\n" << "The second line."
<< endl << endl << "The fourth line." << endl;
```

Two newlines generate a blank line.

We said earlier that there were a small set of reserved words that C++ knew about and we couldn't use as identifiers.

But `cout`, `endl` and `cin` weren't on that list.

So how does C++ know about them, and can we use them as variable names?

We have to tell it, and sometimes.

Stream I/O is an example of a **library**, a collection of

- pre-written code
- pre-defined constants & variables

We can use these in our programs.

But C++ doesn't automatically know about these – we have to tell it.

The information is contained in a **header file**.

This is text file with the extension .h such as

```
iostream.h
```

for stream I/O.

We include this file in our program file by incorporating

```
#include <iostream.h>
```

at the head of our source file.

The C++ compiler does not actually see this statement.

An earlier pass through the file by a pre-processor converts this line to the actual contents of the header file.

We'll see other pre-processor commands later. NOT

For now consider that the < and > are just like " – indicating the file name.

When a header file is included, many constants and variables are declared.

We may not even be aware of the existence of many of these – until we try to declare an identifier which has already been declared.

These names are almost like reserved words – we can't use them for anything else.

Most C++ IDEs, by default, includes many header files useful for OS-specific programs

We will be writing vanilla C++ programs and will not need these headers.

What happens if we forget to include a header file?
C++ will report syntax errors for identifiers not declared, or even syntax not defined.

Now that C++ knows about stream I/O we can use it to have C++ report answers to calculations.

```
#include <iostream.h>

int main()
{
    float degF, degC;

    degF = 40.;
    degC = 5.*(degF-32.)/9.;
    cout << degF << " degrees F is "
        << degC << " degree C" << endl;
    return 0;
}
```

converts 40ºF to its Celsius equivalent

But what if we want to convert 50ºF ?

We could edit the program, recompile and then execute.

Or we could get the value to convert from the keyboard using stream input.

Just as `cout` converts data types from binary representations to character strings, `cin` reverses the process.

Stream input determines how to interpret the characters from the input stream by looking at the type we want to store that value in.

```
cin >> degF;
```

tells stream input that we want to convert a sequence of characters into a `float` value, the type of `degF`.

The >> can be interpreted as 'extract'.

Thus an input statement is just like an assignment – only the conversion from characters to numeric value is performed at execution time instead of by the compiler.

Here is the more complete solution.

```
#include <iostream.h>

int main()
{
    float degF, degC;

    cout << "Please enter a Fahrenheit temperature: ";
    cin >> degF;
    degC = 5.*(degF-32.)/9.;
    cout << degF << " degrees F is "
         << degC << " degrees C\n";
    return 0;
}
```

A prompt – informs the person running the program what to type in.
Note also that the value input is also part of the output – echo input.

What happens when we request values from `cin`?

Characters entered from the keyboard are interpreted:

- whitespace is ignored, but used to separate values
- for a numeric value, all leading whitespace is absorbed
- all legitimate characters of the numeric vlaue are processed until an illegal character is encountered

When the illegal character is encountered, the value converted so far is stored in the variable requested.

If we are reading an integer and the stream contains

```
-212 degrees F
```

then the variable would receive the value -212. The stream would be positioned at the space between the '2' and the 'd'. The same value would result from

```
-212degrees F  or -212. degrees F
```

If we asked for another integer from

```
-212 degrees F
```

we would not get anything from the stream since the 'd' is an invalid character for an integer, and we've got no value so far.

The value received from `cin` and stored in that integer would be undefined.

C++ would **not**, however, tell us.

```
#include <iostream.h>

int main()
{
   float degF, degC;

   cout << "Please enter a Fahrenheit temperature: ";
   cin >> degF;
   degC = 5.*(degF-32.)/9.;
   cout <<degF<<" degrees F is "<< degC<<" degrees C\n";
   cout << "Enter another Fahrenheit temperature:";
   cin >> degF;
   degC = 5.*(degF-32.)/9.;
   cout <<degF<<" degrees F is "<< degC<<" degrees C\n";
   return 0;
}
```

Now the program can convert two temperatures – both stored in `degF`

When execution reaches the first `cin` statement

```
cin >> degF;
```

stream I/O looks at the input stream for a sequence of characters to convert to a float to store in `degF`.

When the input is requested, nothing happens until we enter a return.

```
-212.<ret>
```

```
-212.<ret>
```

is converted to the value -212. and stores the value in `degF`.

When the next `cin` statement is encountered, we could then respond to the prompt with another value and a return.

BUT we could have provided more characters to the stream at the first prompt.

```
-212.    43.<ret>
```

The same initial conversion would be the same, and the stream would be set at the space after the first '.' with the stream still containing more characters.

These other character would be interpreted for the second `cin`
– without waiting for us to type anything in.

Note that integer values (without the .) will be interpreted by stream input as a real value – but `cin` will terminate at a . when integers are being input.

In a complete program, we should ensure that accidental input errors can be reported and corrected, but this is not a simple task, and we will leave that to later.

For now, we will assume that input is correct – after all we're doing it!

We can read in more than one value in a `cin` statement.

```
cout << "Enter your height and age: ";
cin >> Height >> Age;
```

would get two values from the input stream.

(We'll see how this syntax works later.)

What would happen here?

`A, B, and C` are integer variables.

`X, Y, and Z` are reals.

```
cin >> A >> B >> C
       >> X >> Y >> Z;
```

with input

```
1  2  3  4  5.5  6.6
```

```
A ← 1     B ← 2     C ← 3
X ← 4.    Y ← 5.5   Z ← 6.6
```

How about

```
cin >> A >> B >> C;
cin >> X >> Y >> Z;
```

with input

```
1   2   3   4   5.5   6.6
```

Exactly the same.

And here?

```
cin >> A >> B >> C;
cin >> X >> Y >> Z;
```

with input

```
1   2
3   4
5.5   6.6
```

Again the same. Newlines don't matter.

What would happen here?

```
cin >> A >> B >> C
   >> X >> Y >> Z;
```

with input

```
1   2.2
3   4.4
5   6.6
```

A ← 1 B ← 2

then problems. Who knows what C, X, Y, and Z become.

What about character input?

Using the input stream cin, we cannot enter whitespace into a character variable – whitespace is ignored.

So

```
cout << "Please enter your initial: ";
cin >> Initial;
```

would read

¬¬¬¬F

as 'F'.

But what if we ask for two characters?

```
cout << "Enter two letters: ";
cin >> a_letter >> another_letter;
```

Although numerical values must be separated by whitespace, we don't have to separate characters.

Thus

```
ab<ret>
```

and

```
 a   b<ret>
```

result in the same two values being stored.

Binary values can actually be very useful in computer programs. We can choose whether to execute an instruction or not – depending on the truth or falsehood of a logical expression.

C++ represents these two logical values

true by integer value 1

false by integer value 0.

To form logical expressions we use nine operators.

Relational Operators

- binary operators with two operands of the same type (including a propagation)
- a comparison of values

<	is less than
<=	is less than or equal to
==	is equal to
!=	is not equal to
>=	is greater than or equal to
>	is greater than

Any of the basic data types can be used.

Note that we use a different operator for equality than the one for assignment. There is a trap in this.

```
a = b
```

means the variable `a` is assigned the value `b` has currently. But the expression also has a value – that of the result (`a`). After this expression `a` and `b` have the same value (are equal).

Suppose `b` has the value 0, and `a` has the value 1.

Then

```
a = b
```

results in `a` also being 0, and the expression is 0 (meaning `false`).

But

```
a == b
```

would then have the value `true` (or 1).

The C++ compiler provides a warning if an assignment is used where a comparison would normally be expected.

How do we handle multiple comparisons?

$0 \leq x \leq 1$

This does not translate to

```
0 <= x <= 1
```

The latter expression would be calculated as

```
(0 <= x) <= 1
```

For example, if x were 2, the first subexpression would be evaluated as `true` or 1.

Then it would evaluate

```
1 <= 1
```

again true.
But 0 Š x Š 1 is false for x being 2.

So we have to rewrite the composite comparison

$0 \leq x \leq 1$

as

$(0 \leq x)$ and $(x \leq 1)$.

We need operators like **and**.

These are **logical operators**.

Logical Operators

- one unary operator ! meaning **not**.

  Used to negate a logical expression.

  For example

  ```
  age > 60
  ```

  is true if age has a value greater than 60 and false if it is less than or equal to 60.

  ```
  !(age > 60)
  ```

  is true if the expression in the brackets is false, and false if the inner expression is true.

The other two logical operators are binary.

`&&`	logical AND
`||`	logical OR

So

0 Š x Š 1

is written as

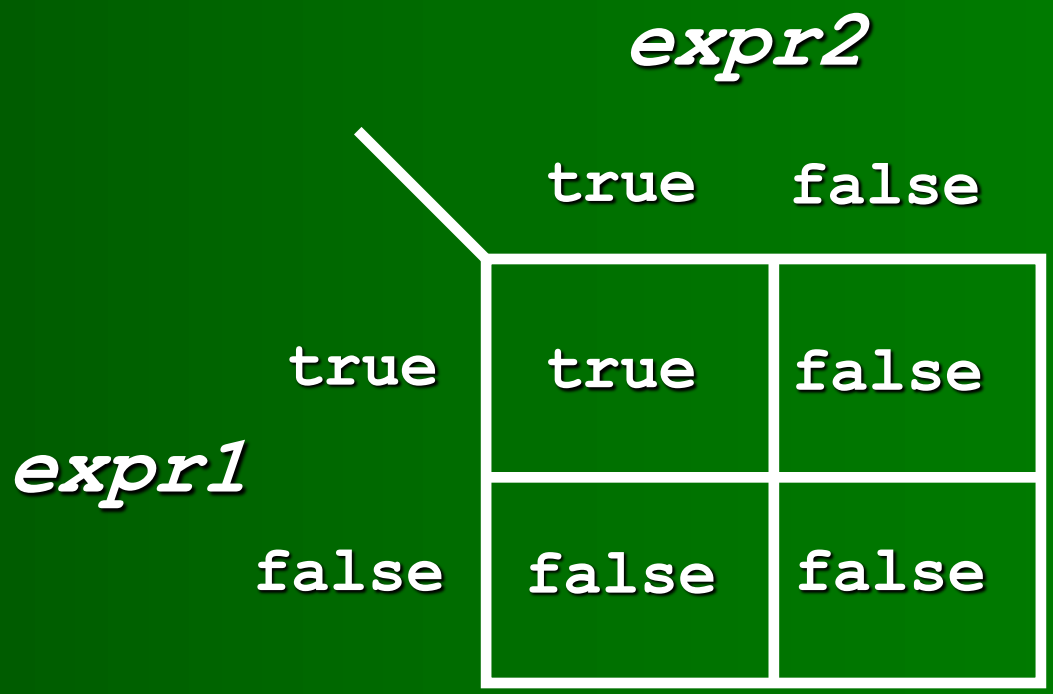
```
(0 <= x) && (x <= 1)
```

What does it mean?

The value of

expr1 `&&` *expr2*

can be summarised in a truth table.



Let's create a few logical variables to illustrate.

Variable 1: Your age exceeds 19.

```
age > 19
```

Variable 2: You are male.

```
sex == 'm'
```

Variable 3: You drive a car to university.

```
car_driver = true
```

```
!(age>19)
```
means
```
age <= 19
```

```
!(sex=='m')
```
means
```
sex != 'm'
```
or
```
sex == 'f'
```

```
!car_driver
```
means not a car driver

```
(age > 19) && (sex == 'm')
```

is true only if both

`age > 19` is true and

`sex=='m'` is true

```
!(sex=='m') && !(car driver)
```

is true only if both are true, that is

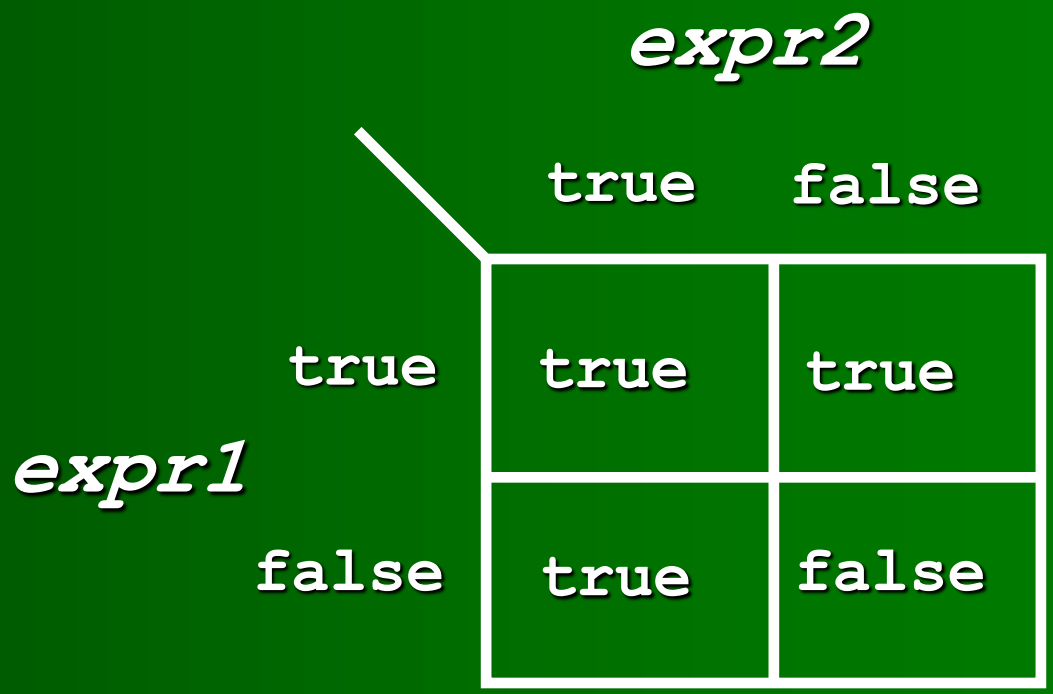
`sex=='m'` is false and

`car_driver` is false

The value of

expr1 || *expr2*

can be summarised in a truth table.



```
(age > 19) || (sex == 'm')
```

is true if either

`age > 19` is true and/or

`sex=='m'` is true

```
!(sex=='m') || !(car driver)
```

is true if either are true, that is

`sex=='m'` is false and/or

`car_driver` is false

This last expression

```
!(sex=='m') || !(car driver)
```

is equivalent to

```
!((sex=='m') && car_driver)
```

In general

```
!(expr1 && expr2)
```

is equivalent to

```
(!expr1) || (!expr2)
```

Similarly

```
!(expr1 || expr2)
```

is equivalent to

```
(!expr1) && (!expr2)
```

Note the use of parentheses to ensure order of evaluation.

Mixing arithmetic and logical expressions needs caution – make sure the order of evaluation is what we want – not what we expect.

```
b * b - 4. * a * c > 0.
```

would reasonably be calculated in the order

```
b * b - 4. * a * c > 0.
```

and is actually done that way (probably).

Here's the precedence of operators:

highest

```
!    +    -    (unary)
*    /    %
+    -
<    <=   >    >=
==   !=
&&
||
=
```

lowest

A Warning

Be cautious about testing for equality of reals.

You may know that

(1./3.)*3. should be 1

But

```
(1./3.)*3. == 1.
```

will most likely be `false`.

Collecting statements and expressions

A **block** is a sequence of statements (of any type, including blocks) collected together by an opening { and a closing }.

e.g.

```
    {
            a = 1;
            c = b;
    }
```

is one such block. Note the indenting.

A block can be used in a C++ program anywhere a single statement could be used.

Such compound statements do not, however, need a ; at the end.

The comma operator is a binary operator for linking assignment expressions (or any other expression) as in

```
a = 1, c = 'b'
```

producing one expression.

Such an expression

```
a = 1, c = 'b'
```

has the value of the rightmost expression.

The comma operator is used infrequently as it is not the most readable of structures.

It appear in certain other constructs for conciseness.