C for Science

Lecture 1 of 5

Sam Bott
http://www2.imperial.ac.uk/~shb104/c
s.bott@imperial.ac.uk

16th January 2013

Imperial College London

Introduction to the Course

This course has been adapted from its previous incarnation, run by Dr Steve Capper.
Originally, the course was based on the M3SC course run by Dan Moore.

Aims of the Course

- To introduce C programming from scratch.
- To provide insight into scientific computing.
- To write fast, efficient and maybe even multi-threaded code!

Five lectures

- Each afternoon will consist of a ≈ 1 hour lecture
- A ≈ 1 hour practical session.

Introduction to the Course

This course has been adapted from its previous incarnation, run by Dr Steve Capper.
Originally, the course was based on the M3SC course run by Dan Moore.

Aims of the Course

- To introduce C programming from scratch.
- To provide insight into scientific computing.
- To write fast, efficient and maybe even multi-threaded code!

Five lectures

- Each afternoon will consist of a ≈ 1 hour lecture
- A ≈ 1 hour practical session.

Course Content

What we'll cover

- Different number types in C (Integers and Floating Point).
- Operators, operands and their precedence.
- Conversions and casts.
- Mathematical and Logical expressions.
- Statements: if, else, ?, while, do, for, switch. . .
- Functions.
- Pointers, arrays and matrices.
- Characters, strings and interacting with the console.
- Reading and writing files.
- Optimisation and Debugging.
- Scientific C-Libraries and their uses: NAG, GSL, etc.
- An Introduction to Parallel Computing.
- C++ and other languages.

A Rough History of C

Invented ≈ 1970

By Dennis Ritchie working in Bell Labs USA; to facilitate development of a portable UNIX.

C has been standardised

- 1989 ANSI standard ratified *ANS X3.159-1989*.
- 1990 ISO standard *ISO/IEC 9899:1990*. Aka *C90*.
- 2000 ISO standard *ISO/IEC 9899:1999*. Aka *C99*.

C++ has evolved from C

Bjarne Stroustrup developed C++ (C with class). Unlike C, C++ is still under very active development (C++11 being the most recent standard at the time of writing).

A Rough History of C

Invented ≈ 1970

By Dennis Ritchie working in Bell Labs USA; to facilitate development of a portable UNIX.

C has been standardised

- 1989 ANSI standard ratified *ANS X3.159-1989*.
- 1990 ISO standard *ISO/IEC 9899:1990*. Aka *C90*.
- 2000 ISO standard *ISO/IEC 9899:1999*. Aka *C99*.

C++ has evolved from C

Bjarne Stroustrup developed C++ (C with class). Unlike C, C++ is still under very active development (C++11 being the most recent standard at the time of writing).

A Rough History of C

Invented ≈ 1970

By Dennis Ritchie working in Bell Labs USA; to facilitate development of a portable UNIX.

C has been standardised

- 1989 ANSI standard ratified *ANS X3.159-1989*.
- 1990 ISO standard *ISO/IEC 9899:1990*. Aka *C90*.
- 2000 ISO standard *ISO/IEC 9899:1999*. Aka *C99*.

C++ has evolved from C

Bjarne Stroustrup developed C++ (C with class). Unlike C, C++ is still under very active development (C++11 being the most recent standard at the time of writing).

What are C and C++?

- C is a cross-platform, compiled, general-purpose language.
- C++ can loosely be thought of as C's object-oriented big brother.

The vast majority of the programs running on your computer (including the operating system kernel), are written in either C or C++. In the case of Windows, another big contender for a lot of the more recent applications is C# (but it isn't appropriate at kernel-level or for fast number-crunching).

Why Use C? (Over Maple, Matlab. . . Excel(?!). . .)

Speed

C programs are compiled to machine code, the resulting routines *can* run several orders of magnitude quicker than their equivalents in interpreted environments.

Flexibility

The C language is intrinsically low level, one can manipulate complex data structures with surprisingly little code.

Portability

A well written C program can target many different environments (Windows PCs, Linux workstations, Apple Macs, DEC Alphas, Embedded devices. . .).

Why Use C? (Over Maple, Matlab. . . Excel(?!). . .)

Speed

C programs are compiled to machine code, the resulting routines *can* run several orders of magnitude quicker than their equivalents in interpreted environments.

Flexibility

The C language is intrinsically low level, one can manipulate complex data structures with surprisingly little code.

Portability

A well written C program can target many different environments (Windows PCs, Linux workstations, Apple Macs, DEC Alphas, Embedded devices. . .).

Why Use C? (Over Maple, Matlab. . . Excel(?!). . .)

Speed

C programs are compiled to machine code, the resulting routines *can* run several orders of magnitude quicker than their equivalents in interpreted environments.

Flexibility

The C language is intrinsically low level, one can manipulate complex data structures with surprisingly little code.

Portability

A well written C program can target many different environments (Windows PCs, Linux workstations, Apple Macs, DEC Alphas, Embedded devices. . .).

Getting Started

You will need:

- A C compiler (many different ones to choose from, some are free).
- Some documentation (such as the lecture notes/exercises from this course, a good book, online guides).
- Lots, and lots of time.

Commercial C Compilers

- `Intel` - for Windows or Linux. Compiles highly optimised code for Intel (and AMD) processors. Free for personal use and academic use by students. Full-academic and commercial licenses obtainable from: http://www.polyhedron.com
- `Microsoft Visual Studio 2010 Professional` - Microsoft's flagship compiler. Ninety day free trial available at: http://www.microsoft.com/visualstudio/en-us/try

Free C Compilers

Linux

- `gcc` - The GNU Compiler Collection, C compiler. http://gcc.gnu.org.

Windows

- `Visual C++ 2010 Express` - Microsoft's free compiler, http://www.microsoft.com/express/vc/
- `MinGW` - Minimalist GNU for Windows, http://www.mingw.org/.

Free C Compilers

Linux

- `gcc` - The GNU Compiler Collection, C compiler. http://gcc.gnu.org.

Windows

- `Visual C++ 2010 Express` - Microsoft's free compiler, http://www.microsoft.com/express/vc/
- `MinGW` - Minimalist GNU for Windows, http://www.mingw.org/.

Integrated Development Environments

gcc (for Linux or MinGW) is a command line driven compiler; an Integrated Development Environment (IDE) is a graphical application that provides tools to assist with editing, compiling and debugging of code. I'd recommend Visual Studio Professional as an IDE, this can be obtained through DreamSpark if you're eligible. For a non-windows or free alternative, I'd recommend:

Windows `Visual C++ 2010 Express`, the free Microsoft IDE. `http://www.microsoft.com/express/vc/`

Windows/Linux/Mac `Code::Blocks`, a cross-platform, open source IDE. `http://www.codeblocks.org/`

Books for C

Kernighan and Ritchie (K&R2)

The C Programming Language, *S*econd Edition, Prentice Hall. This is a fantastically structured reference book, it is written by the authors of C and is **the** C reference!

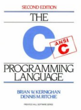
Numerical Recipes in C

By Press, Teukolsky, Vetterling & Flannery, *S*econd Edition, CUP. Full of high quality example scientific C code. A free online edition can be found at: `http://www.nr.com`
There is also a C++ edition in paper or online format.

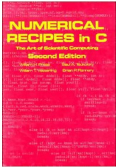
Books for C

Kernighan and Ritchie (K&R2)

The C Programming Language, Second Edition, Prentice Hall. This is a fantastically structured reference book, it is written by the authors of C and is **the** C reference!



Numerical Recipes in C



By Press, Teukolsky, Vetterling & Flannery, *S*econd Edition, CUP. Full of high quality example scientific C code. A free online edition can be found at:
`http://www.nr.com`
There is also a C++ edition in paper or online format.

Building a C Program

- To *build* an executable from source, we carry out the following three steps:

Edit Source

Use a text editor to create a `.c` file.

Compile

With a C compiler, this creates *object file(s)*.

Link

Combine the object files together into an *executable*.

- These steps are can be automated by *Integrated Development Environments* (IDEs).

Building a C Program

- To *build* an executable from source, we carry out the following three steps:

Edit Source

Use a text editor to create a `.c` file.

Compile

With a C compiler, this creates *object file(s)*.

Link

Combine the object files together into an *executable*.

- These steps are can be automated by *Integrated Development Environments* (IDEs).

Building a C Program

- To *build* an executable from source, we carry out the following three steps:

Edit Source

Use a text editor to create a `.c` file.

Compile

With a C compiler, this creates *object file(s)*.

Link

Combine the object files together into an *executable*.

- These steps are can be automated by *Integrated Development Environments* (IDEs).

Building a C Program

- To *build* an executable from source, we carry out the following three steps:

Edit Source

Use a text editor to create a `.c` file.

Compile

With a C compiler, this creates *object file(s)*.

Link

Combine the object files together into an *executable*.

- These steps are can be automated by *Integrated Development Environments* (IDEs).

The Traditional Way to Start

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

The “Hello World” Program

A traditional first program started by Ritchie. This is one of the smallest possible C programs that demonstrates some functionality (printing to screen).

The Traditional Way to Start

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello World!\n");
6      return 0;
7  }
```

Line 1

A *pre-processor directive* (it begins with a #) advertising extra routines to the compiler.

The Traditional Way to Start

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

Line 2

An empty line, or equivalently, a line consisting solely of *whitespace*. This is ignored by the compiler but makes the source code more readable.

The Traditional Way to Start

```c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

Line 3

A *function declaration*, defining our `main` function. The `main` function is where our program starts and is known as an *entry point*. Our main function takes *no parameters* (`void`) and *returns* an integer (`int`).

The Traditional Way to Start

```c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

Line 4

Opening brace, all statements enclosed between the braces {, } belong to the `main` function.

The Traditional Way to Start

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello World!\n");
6      return 0;
7  }
```

Line 5

A *statement*; the `printf` (print formatted) function is called with the argument `"Hello World!\n"`. This prints:

```
Hello World!
```

to *standard output* (usually a text console).

The Traditional Way to Start

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

Line 6

A *return statement*, we exit `main` with a return code of 0. The system interprets 0 as “success”.

The Traditional Way to Start

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

Line 7

A *closing brace*, everything after this line does not belong to `main`.

Another C Program - What does this do?

```c
#include <stdio.h>

int main(void)
{
    int low=-40, high=140, step=5, f, c;
    c = low;
    while (c <= high)
    {
        f = 32+9*c/5;
        printf("%6d \t %6d\n", c, f);
        c = c + step;
    }
    return 0;
}
```

Another C Program - What does this do?

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int low=-40, high=140, step=5, f, c;
6     c = low;
```

Lines 1, 2, 3 & 4

Identical meaning as in the previous program.

Another C Program - What does this do?

```c
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int low=-40, high=140, step=5, f, c;
6     c = low;
```

Line 5

Local variable declarations; the integers `low`, `high`, `step`, `f` and `c` are declared. These are local to `main`. The variables `low`, `high` and `step` are *initialised* with the values; whilst `f` and `c` are *undefined*.

Another C Program - What does this do?

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int low=-40, high=140, step=5, f, c;
6     c = low;
```

Line 6

The local variable `c` is *assigned* the value of `low`.

Another C Program - What does this do?

```
 7     while (c <= high)
 8     {
 9         f = 32+9*c/5;
10         printf("%6d \t %6d\n", c, f);
11         c = c + step;
12     }
13     return 0;
14 }
```

Lines 7, 8 & 12

A *while* loop is defined. For as long as the variable `c` is less than or equal to `high`, the code between the braces on lines 8 and 12 is executed.

Another C Program - What does this do?

```
7       while (c <= high)
8       {
9           f = 32+9*c/5;
10          printf("%6d \t %6d\n", c, f);
11          c = c + step;
12      }
13      return 0;
14  }
```

Line 9

The local variable `f` is assigned a value from the integer arithmetic expression involving `c`.

Another C Program - What does this do?

```
 7      while (c <= high)
 8      {
 9          f = 32+9*c/5;
10          printf("%6d \t %6d\n", c, f);
11          c = c + step;
12      }
13      return 0;
14  }
```

Line 10

The variables `c` and `f` are printed to standard out, each six characters wide, separated by a tab and two spaces.

Another C Program - What does this do?

```
 7      while (c <= high)
 8      {
 9          f = 32+9*c/5;
10          printf("%6d \t %6d\n", c, f);
11          c = c + step;
12      }
13      return 0;
14  }
```

Line 11

The local variable `c` is incremented by `step`.

Another C Program - What does this do?

```
 7      while (c <= high)
 8      {
 9          f = 32+9*c/5;
10          printf("%6d \t %6d\n", c, f);
11          c = c + step;
12      }
13      return 0;
14  }
```

Lines 13 & 14

Have an identical meaning as in the last program.

Commenting C Programs

A comment is text in the source file that gets ignored by the compiler. This is useful for providing human-readable notes about what the code is doing, or removing lines of code temporarily. There are two ways of commenting files in C:

Traditional Way

Anything between `/*` and `*/` is a comment, i.e.

```
/* This function is used to compute the
roots of a quadratic equation */
```

C++ Style

These are single line only, anything after `//` is a comment, i.e.

```
int c = 3; // set c to 3
```

Technically, C++ style comments aren't in the C standard. (But they are ubiquitous to C code anyway).

Commenting C Programs

A comment is text in the source file that gets ignored by the compiler. This is useful for providing human-readable notes about what the code is doing, or removing lines of code temporarily. There are two ways of commenting files in C:

Traditional Way

Anything between `/*` and `*/` is a comment, i.e.

```
/* This function is used to compute the
roots of a quadratic equation */
```

C++ Style

These are single line only, anything after `//` is a comment, i.e.

```
int c = 3; // set c to 3
```

Technically, C++ style comments aren't in the C standard. (But they are ubiquitous to C code anyway).

Variable Names

From K&R

"... Is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different."

What to Avoid. . .

- Punctuation or any other symbols are not allowed in variable names.
- The modern C standard discourages the use of an underscore as the first character of a variable name.

Variable Names

From K&R

"... Is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different."

What to Avoid. . .

- Punctuation or any other symbols are not allowed in variable names.
- The modern C standard discourages the use of an underscore as the first character of a variable name.

Preprocessor Directives

- One example is:

```
#include <stdio.h>
```

Which tells the compiler to search `<stdio.h>` for functions.

- Another example is:

```
#define MAXSIZE 1024
```

This replaces all occurrences of `MAXSIZE` with `1024`.

  - Define statements can be named in a similar way to variables, but
  - It is convention to use upper case for `#define` statements.

- Or even simpler:

```
#define NDEBUG
```

Meaning `NDEBUG` is defined. This will be expanded later on.

Preprocessor Directives

- One example is:

  `#include <stdio.h>`

  Which tells the compiler to search `<stdio.h>` for functions.
- Another example is:

  `#define MAXSIZE 1024`

  This replaces all occurrences of `MAXSIZE` with `1024`.
  - Define statements can be named in a similar way to variables, but
  - It is convention to use upper case for `#define` statements.
- Or even simpler:

  `#define NDEBUG`

  Meaning `NDEBUG` is defined. This will be expanded later on.

Preprocessor Directives

- One example is:

  `#include <stdio.h>`

  Which tells the compiler to search `<stdio.h>` for functions.
- Another example is:

  `#define MAXSIZE 1024`

  This replaces all occurrences of `MAXSIZE` with `1024`.
  - Define statements can be named in a similar way to variables, but
  - It is convention to use upper case for `#define` statements.
- Or even simpler:

  `#define NDEBUG`

  Meaning `NDEBUG` is defined. This will be expanded later on.

Numbers in C

There are two types of number in C:

Integers

Integers are a type of number that can only hold one of a finite range of whole number values.

Floating Point Numbers

Floating point numbers are more flexible and provide an approximation to real numbers. They store a representation of a number in a form with a similar concept to scientific notation.

Numbers in C

There are two types of number in C:

Integers

Integers are a type of number that can only hold one of a finite range of whole number values.

Floating Point Numbers

Floating point numbers are more flexible and provide an approximation to real numbers. They store a representation of a number in a form with a similar concept to scientific notation.

Numbers in C

There are two types of number in C:

Integers

Integers are a type of number that can only hold one of a finite range of whole number values.

Floating Point Numbers

Floating point numbers are more flexible and provide an approximation to real numbers. They store a representation of a number in a form with a similar concept to scientific notation.

Integers

Integer types in C can be thought of as rings of different sizes (i.e. hours on a clock face). They hold one of the range of integers in the ring and once the highest value in the ring is reached the next incremental value will be the lowest value in that ring.

- As integers lose all fractional data, multiplication is not always the inverse of division.
- Products higher than the size of the ring will wrap to the smallest value in the ring; division is not necessarily the inverse of multiplication.

Integer Types

`short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`

Integer Types - For a 32 bit program

Type	Min	Max
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	4294967295
long	-2147483648	2147483647
unsigned long	0	4294967295
long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

For example, here are two bit patterns for `short`:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= -1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	= 7

(for more information see `<limits.h>`)

Integer Types

- Two main subtypes *signed* and *unsigned*. Signed types use a sign bit.
- For signed types we, usually, have:
  - minimum value: $-2^{\text{size}-1}$
  - maximum value: $2^{\text{size}-1} - 1$
- For unsigned types we have:
  - minimum value: 0
  - maximum value: $2^{\text{size}} - 1$
- `short` is often used to conserve memory.
- `int` represents the *native* CPU integer type so is used for speed. (If in doubt use `int`).
- `long` and `long long` are used to maintain accuracy.

Integer Arithmetic

Base Operators

The four usual operators are defined $+$, $-$, $*$ and $/$.

Ring arithmetic

Division is not always the reverse of multiplication:
`1/2=0, 0*2=0.`
Also, any result of a computation must lie within the ring, any number outside the range of the current data type will “wrap” around. (i.e. 11am + 3 hours gives 2pm).

Modulo Operator

The remainder operator `%` is unique to integer types, it acts as expected: `7%2 = 1`.

Floats

These are much more flexible numbers, but still *NOT* the same as $\mathbb{R}$.

- Because of the way the number is stored, even fairly simple-looking base-10 numbers $(0.1, 0.2, 0.3 \ldots)$ are only stored as an approximation.
- Because numbers are kept as approximations: associativity and commutativity don't always apply and multiplicative inverses don't always exist.
- The way in which the numbers are stored can result in a loss of the least significant parts of numbers. This causes problems when adding/subtracting big and small numbers together.
- Programming floats well for numerical problems, especially with large/small numbers, is an art form!

Float Types

`float`, `double`, `long double`

Floating Point Numbers (IEEE 754 Standard)

On my machine, a `float` (single precision) looks like:

b	e	e	e	e	e	e	e	e	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f

It consists of three parts, the *sign bit*(b), the *biased exponent*(e) and the *fraction*(f). We break down a number x:

$$x^{\text{float}} = (-1)^{b} \times 2^{e-127} \times (1 + f \times 2^{-23}), \quad \begin{array}{c} 0 < e < 255 \\ 0 \leq f \leq 2^{23} - 1 \end{array},$$

We have four special numbers, `-Inf` $(-\infty)$, `Inf` (∞), `NaN` (Not a Number) and zero.
For `double` (double precision) we have:

$$x^{\text{double}} = (-1)^{b} \times 2^{e-1023} \times (1 + f \times 2^{-52}), \quad \begin{array}{c} 0 < e < 2047 \\ 0 \leq f \leq 2^{52} - 1 \end{array}.$$

Floating Point

Base Operators

As with integers, we have $+$, $-$, $\star$ and $/$.

Floating point code

- It looks like integer code but with a decimal point suffix.
- Scientific notation is achieved with `e`:
  `double speedofLight = 2.997e8;` (2.997×10^8)

Float Arithmetic

- Division is not always the reverse of multiplication.
- Operators may not be commutative!

$$A + B + C \neq A + C + B \qquad \text{(sometimes)}$$

More Mathematical Functions in `<math.h>`

- Maths functions come with the *ANSI Standard C Library*, which contains many maths functions. To use them we need a:
  `#include <math.h>`
- Here some example functions:

`sin(x)`	`asin(x)`	`sinh(x)`	`exp(x)`
`cos(x)`	`acos(x)`	`cosh(x)`	`log(x)`
`tan(x)`	`atan(x)`	`tanh(x)`	`log10(x)`
`sqrt(x)`	`atan2(x,y)`	`pow(x,y)`	`fabs(x)`

  (all the trigonometric functions use radians!)

The `pow(x, y)` function (declared in `<math.h>`)

Exponentiation

There is no exponentiation operator (e.g. $\wedge$, `**`) in C. Instead we have the following:

$$x^y = \texttt{pow}(\texttt{x}, \texttt{y})$$

This assumes x and y are of type `double`.

Beware

The `pow` function is often implemented as:

```
exp(y*ln(x))
```

For whole integer powers (i.e. x^2), one should perform the multiplication explicitly (`x*x`).

Simple Logical Expressions

```
7  while (c <= high)
```

- Are used to carry out branches (`if` statement) and loops (such as `for`, and `while`).
- Evaluate to either *true* (non-zero `int`) or *false* (zero).

Logical Operators

x	>	y	is x greater than y?
x	>=	y	is x greater than or equal to y?
x	<	y	is x less than y?
x	<=	y	is x less than or equal to y?
x	==	y	is x equal to y?
x	!=	y	is x different to y?

Using == Safely

The danger of the easily-made typo:

```
if (x = 3) {Statement;}
```

is it will **always** return true and execute the statement, it will also overwrite `x` with `3`. This is not only undesirable as it is will not be testing the desired expression, but it is valid code so will not always throw an error - making debugging very tricky.

A Preventative Measure

If the variables `x` and `3` were to be swapped, such as:

```
if (3 == x) {Statement;}
```

Then if `=` was used rather than `==` it would cause an error as a value cannot be assigned to 3, but it keeps the expression logically equivalent. Getting into the habit of using the variables this way round can save hours of debugging!

Using == Safely

The danger of the easily-made typo:

```
if (x = 3) {Statement;}
```

is it will **always** return true and execute the statement, it will also overwrite `x` with `3`. This is not only undesirable as it is will not be testing the desired expression, but it is valid code so will not always throw an error - making debugging very tricky.

A Preventative Measure

If the variables `x` and `3` were to be swapped, such as:

```
if (3 == x) {Statement;}
```

Then if = was used rather than == it would cause an error as a value cannot be assigned to 3, but it keeps the expression logically equivalent. Getting into the habit of using the variables this way round can save hours of debugging!

Compound Logical Expressions

We can create compound logical expressions using the following operators:

- `||` is a *logical or*. `le1 || le2` returns false if both `le1` and `le2` are false and true otherwise.
- `&&` is a *logical and*. `le1 && le2` returns true if and only if both `le1` and `le2` are true.
- `!` is a *logical not*. `!le1` returns the opposite of `le1`.

Here are two identical examples:

- `(x < 100) && (x%2 == 0)`
- `(x < 100) && !(x%2)`

Flow Control - `if`

Executes block(s) of code depending on the evaluation of a logical expression.

Simple `if`

`if` (*logical expression*) {*statements*;}

`if`, `else if`, `else`

```
if (logical expression)
    {statements;}
else if (logical expression)
    {statements;}
else if (logical expression)
    {statements;}
else
    {statements;}
```

Flow Control - `while`

A `while` loop is used to repeatedly execute code as long as a logical expression is true.

Structure

```
while (logical expression)
    { statements ;}
```

- If *logical expression* is false, then the *statements* are never executed.

Flow Control - `do {} while ()`

We place the *logical expression* after the *statements* giving us:

Structure

```
do {statements;}
while (logical expression)
```

- The *statements* are executed at least once.

`do while` or `while`?

Generally I prefer `while` over `do while`, as it forces me to initialise variables properly.

Flow Control - `for` loop

```
for ( start expression ;
      logical expression ;
      step expression)
      { statements ;}
```

- Print out ten numbers:

```
for (x=0; x < 10; x = x + 1)
   printf("x = %d\n", x);
```

- Keep looping indefinitely (printing out dots)

```
for (;;) printf(".");
```

Flow Control - `switch - case`

We can selectively execute code based on a value, using the following:

```
switch (integer_statement) {
case integer_value1: statements1; break;
case integer_value2: statements2; break;
case integer_value3:
case integer_value4: statements3; break;
default: statements4; break;}
```

- Execution starts at either one of the `case`'s or at `default`.
- Execution stops at the end } or at `break`.
- `case`, `default` and `break` are optional.

Some Loop Control Features

Execution of code inside a loop (`do`, `while`, `for`) can be manipulated by the following statements.

`break;`

Break out of the current loop. Any statements in the loop following the `break` are ignored and the loop condition automatically evaluates to false, ending the loop.

`continue;`

Jump to the end of the current loop (effectively ignoring everything below the `continue` statement. Whether or not the loop continues executing depends on the loop condition.

`printf` - declared in `<stdio.h>`

As seen in the examples, the `printf` function can be used to print out variables. The function *prototype* takes the form:

```
int printf(char * formatString, ...)
```

`formatString`

The first argument of `printf` is the *format string*, this specifies how many variables need printing out, how they are to be printed, and in what order.

`...`

This is C shorthand for *variable number of arguments*.

Return value: `int`

`printf` returns the number of characters printed.

`printf` - declared in `<stdio.h>`

As seen in the examples, the `printf` function can be used to print out variables. The function *prototype* takes the form:

```
int printf(char * formatString, ...)
```

`formatString`

The first argument of `printf` is the *format string*, this specifies how many variables need printing out, how they are to be printed, and in what order.

`...`

This is C shorthand for *variable number of arguments*.

Return value: `int`

`printf` returns the number of characters printed.

`printf` - declared in `<stdio.h>`

As seen in the examples, the `printf` function can be used to print out variables. The function *prototype* takes the form:

```
int printf(char * formatString, ...)
```

`formatString`

The first argument of `printf` is the *format string*, this specifies how many variables need printing out, how they are to be printed, and in what order.

`...`

This is C shorthand for *variable number of arguments*.

Return value: `int`

`printf` returns the number of characters printed.

`printf` - declared in `<stdio.h>`

As seen in the examples, the `printf` function can be used to print out variables. The function *prototype* takes the form:

```
int printf(char * formatString, ...)
```

`formatString`

The first argument of `printf` is the *format string*, this specifies how many variables need printing out, how they are to be printed, and in what order.

`...`

This is C shorthand for *variable number of arguments*.

Return value: `int`

`printf` returns the number of characters printed.

`printf` - declared in `<stdio.h>`

We call `printf` as follows:

```
printf(formatString, var1, var2, ..., varN);
```

where,

`formatString`

The format string tells `printf` how many variables need printing. A format string can contain *format specifiers*, these tell `printf` exactly how to print out each variable, some examples:

`"%6d"`	print out an integer (6 characters wide).
`"%g"`	print out a floating point number.

`var1, ...`

`printf` accepts a variable list of arguments, which can be of different type. Care must be taken to match `formatString` with the variables.

`printf` - declared in `<stdio.h>`

We call `printf` as follows:

```
printf(formatString, var1, var2, ..., varN);
```

where,

`formatString`

The format string tells `printf` how many variables need printing. A format string can contain *format specifiers*, these tell `printf` exactly how to print out each variable, some examples:

`"%6d"`	print out an integer (6 characters wide).
`"%g"`	print out a floating point number.

`var1, ...`

`printf` accepts a variable list of arguments, which can be of different type. Care must be taken to match `formatString` with the variables.

Special Characters

- The backslash \ character in C has a special meaning, it is known as the *escape character*.
- We combine the escape character with other characters, to form an *escape sequence*, here are some examples:

`\n`	New line
`\t`	Tab
`\b`	Backspace
`\r`	Carriage return
`\a`	Bell
`\f`	Form feed (new page)
`\\`	\
`\"`	"
`\'`	'

scanf() - Reading Data from Standard Input

For two variables `A` and `B`, both of type `double`, we use:

```
scanf("%lf %lf", &A, &B);
```

- where the `%` represent *format specifiers*

Format Specifiers

Consist of a `%`, a numerical width specification and a field code:

`d`	`int`	`g`	`float` (general form)
`u`	`unsigned int`	`lf`	`double` (fixed form)
`f`	`float` (fixed form)	`le`	`double` (exponential form)
`e`	`float` (exponential form)	`lg`	`double` (general form)

- and the `&` represents the *address* of the variable in memory. This is known as a *pointer reference operator*.

`scanf()` - Reading Data from Standard Input

For two variables `A` and `B`, both of type `double`, we use:

```
scanf("%lf %lf", &A, &B);
```

- where the `%` represent *format specifiers*

Format Specifiers

Consist of a `%`, a numerical width specification and a field code:

`d`	`int`	`g`	`float` (general form)
`u`	`unsigned int`	`lf`	`double` (fixed form)
`f`	`float` (fixed form)	`le`	`double` (exponential form)
`e`	`float` (exponential form)	`lg`	`double` (general form)

- and the `&` represents the *address* of the variable in memory. This is known as a *pointer reference operator*.

Why the `&A` in `scanf()`?

- Functions in C can return only one value.
- Sometimes we want more than one value to change.
- If we tell `scanf` *where* the variables are in memory, `scanf` can change them itself.

The ability to manipulate memory directly is what makes C so powerful (and potentially dangerous).

Pointers

A *pointer* is a variable that stores a memory location, they are declared as follows:

```
double * ptrA;
```

& - Pointer reference operator

Returns the memory address (pointer to) of a variable.

```
ptrA = &A;    // ptrA points to A
```

* - Pointer de-reference operator

Converts a memory address to a variable:

```
* ptrA  = 1.234;       // A is now 1.234
```

Pointers

A *pointer* is a variable that stores a memory location, they are declared as follows:

```
double * ptrA;
```

& - Pointer reference operator

Returns the memory address (pointer to) of a variable.

```
ptrA = &A;    // ptrA points to A
```

* - Pointer de-reference operator

Converts a memory address to a variable:

```
* ptrA  = 1.234;      // A is now 1.234
```

Summary

- C is a cross-platform, compiled language which can produce results much quicker than other methods/languages.
- We use an IDE to write source, compile, link and debug our C programs.
- The basic structure of a C program has been demonstrated.
- There are two categories of number in C: integers and floating point numbers.
- We have seen how logic and statements can control the flow of a program.
- `printf` and `scanf` will write and read from the console respectively.