



# School of Physics

## Introduction to C

Comments and questions to [John Rowe](#).

### Brief reflection on the turtle

Review the questions at the end of Mr Turtle.

- What features did you find surprising or fun?
- What features do you think would make it easier to do complicated things?

The instructions that make up our program are conventionally referred to as "code", as in "code of conduct".

### The C programming language

C is a general-purpose programming language which according to [one widely quoted but not very meaningful index](#) is the second most popular programming language in the world today. It has a philosophy of **simplicity** and **consistency**.

As with all programming languages the purpose of C is to **unambiguously express an algorithm** in a way that is easy for us to understand and modify.

C's popularity has spawned several off-shoots which are, with very few exceptions, *super-sets* of the original language, i.e. any valid C program will also be a valid program in any of those languages. These include C++, C# (C-sharp) and Objective C, the main programming language used on the iPhone and iPad. When looking for information on the Web or buying a course book **be sure it is about the C programming language, not C++ or one of the other off-shoots**.

On the other hand, we can also use a C++ compiler as a C compiler provided we are careful only to use its C features, not any of the C++ extensions. (A **compiler** is a program that converts the human-readable program we have written into the executable program that will run on our computer.)

Text which is indented relative to the main page are **notes**. These are mid-way in importance between the main text and asides like this. You should vaguely remember you have read them but they are not of primary importance.

## Our first C program

In C, as in many languages, the statements that implement our algorithm are grouped into **functions** which implement a well defined task. (In Logo these are called "procedures", other names can include "routines" or "subroutines".) Traditionally, our first C program looks like this:

```
/*
 * My first C program. Successfully prints the phrase:
 * "Hello, world"
 */
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

The C programming language was developed at Bell labs where they also invented the transistor and discovered the cosmic background radiation left over from the Big Bang.

Though simple, this program illustrates the three components found in any C program: Comments, pre-processor directives and C code proper.

### Comments

The first thing we see is a **comment**, which is there purely for our benefit:

```

/*
 * My first C program. Successfully prints the phrase:
 * "Hello, world"
 */

```

Text starting with `/*` and ending with `*/` is ignored by the compiler (strictly speaking it is replaced by a single space but C doesn't worry much about spaces anyway). It's there to give us the context of what's going on and to help us understand anything that can't be made clear in the program itself.

Comments at the start of programs and just before functions are extremely useful as it's easy to be *slightly* unclear or ambiguous about what a function does. Comments inside of functions are less useful.

For multiline comments we should adopt a consistent style that makes it easy to see at a glance that it's a comment. The style above is a common one, where we have put an asterisk (`*`) at the start of every line and indented the end-of-comment indicator `*/` by one so they all line up. Remember, it's a *comment* so we can put what we like in there and format it as we like, so choose a style that is clear to you and use it consistently.

C also allows single line comments starting with `//`: The `//` and everything after it up to the end of the line is ignored. We could have equally written the above comment:

```

//
// My first C program. Successfully prints the phrase:
// "Hello, world"
//

```

Single-line `//` comments are a later addition to the C language

## # Directives

Next we see the line:

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

Lines like this that start with a hash (`#`) are called pre-processor **directives**. In this case the directive is to find and include a file called `stdio.h`.

The word `stdio` is short for **STANDARD Input/Output**, i.e. a collection of standardised functions available to us that will read from, and write to, the screen and/or files on our computer. Somewhere in the depths of our computer will be a file called `stdio.h` (I strongly recommend you don't try to read it!) and this command **includes** that file in our program when it is compiled. This file `stdio.h` contains all the definitions necessary for us to use these standard input and output functions (we will see one of them in the main program). It's normal to put all of our `#include` statements at the top of the file.

We will use this construct quite often as the standard library is divided into convenient groups, each with its own include file. This is also the method by which we can use "third-party" libraries (i.e., software written by other people) for tasks such as graphics, advanced mathematical analysis, etc.

It is also used when our program becomes large enough to require splitting between several files and we need to keep various definitions, etc. consistent between them.

## The main() function

Finally we come to the actual C code itself. The construct:

```

int main() {
    ...
}

```

Throughout this course we will use the ellipsis ... to indicate omitted code.

indicates that we have written a function called **main** which will return an integer value to whatever called it. The function `main` has a special place in C in that the program runs by calling and executing `main`, and hence any functions contained within it, then quitting. The value returned by `main` is then passed back to whatever ran the program to indicate whether the program succeeded or failed. The convention is for a program to return zero upon success and a non-zero value on failure.

The actual content of the function lives in between the matched pair of **braces** `{ }`, or curly brackets. The first statement:

```
printf("Hello, world\n");
```

calls the **printf** function which "prints" the output, not to a printer but to the default output device, usually the screen. (In the early days of computing the default output device was a printer!) The "f" at the end of `printf` stands for "formatted" and we shall see later how to use it to print numbers, etc to the screen as well as plain text.

The **parentheses**, following the word `printf` enclose the **arguments** which are passed to the `printf` function. In this case there is just one, a **character string** enclosed within **double-quotes**. (They have to be double quotes, single quotes are used for something else.)

The actual character string itself is fairly straightforward except for the funny `'\n'` at the end which stands for "new line".

There are several of these so-called escape characters, consisting of a backslash (`\`) followed by a letter. These include `'\t'` (tab) and `'\a'` (alert, which sounds the beep). We also need to use a `'\'` character when we need a literal backslash or double quote inside a character string:

```
printf("\"Hello\", he said\n");
```

The statement finishes with a semi-colon (`;`). The individual statements within a `{ ... }` block all end in a semi-colon and if they are very long they can be split over several lines for clarity.

The second and final statement:

```
return 0;
```

returns from `main` in both senses of the word: the execution of `main` immediately stops and the value zero is returned. If we had put any statements after the `return` statement they would never be reached and, if we were using a nice compiler, it would warn us. Notice again the semi-colon at the end of the statement.

## Summary

C source files typically consist of three elements, handled in this order:

**Pre-processor directives** (`#include<stdio.h>`) modify the source code, in this example by finding the file `stdio.h` and "pasting" its contents into the file at this point. Pre-processor directives are unusual as they are one-per-line.

**Comments** (`/* Hello mum! */`), which exist for human readers only, are replaced by spaces.

**C code**: everything that reaches this stage, both our own code and that `#included` by the pre-processor, should be valid C code which is **compiled** into binary files that our processor can understand.

- The program runs the **main()** function which returns zero to indicate success or and a non-zero integer for failure.
- The body of the function is enclosed inside `{ ... }`.
- **printf("format-string");** prints to the screen.
- Individual **statements** end in a semi-colon.
- **return value;** returns from the function and returns *value* to whatever called it.

Assuming that our code survived these stages and compiled successfully, it will now be passed to the final stage where it is **linked** together with any other source files we may have written and the library functions we have called to create the final executable program.

You don't need to know the detail of the order of these stages, but you do need to know the functions of the three typical components of a C source file.

The pre-processor doesn't actually modify our file, it makes a temporary copy of its contents and modifies the copy.

When we are writing large programs it is convenient to group related functions into separate files rather than have one huge source file.

## Making our program easier to understand

In the preface to this series we emphasised the importance of avoiding and finding mistakes. Amongst other things this means having code that is really easy to understand.

## Mega-principle

Whenever we are faced with a choice, our first question should always be:  
**"which choice will be the clearest and give me the least chance of making a mistake?"**.

The first application of this principle is the fact that clarity of our code is enormously helped by **indentation**.

Compare a set of bullet points printed twice, with and without indentation: :

### Pets: pros and cons

- Cat
  - Easy to look after
  - Kills birds
- Dog
  - Friendly
  - Needs daily walk
- Great white shark
  - Looks great.
  - Feeds itself
  - Has a nasty nip:
    - Big pointy teeth
    - Can open jaws *really wide*
  - Not ideal first pet

### Pets: pros and cons

- Cat
  - Easy to look after
  - Kills birds
- Dog
  - Friendly
  - Needs daily walk
- Great white shark
  - Looks great.
  - Feeds itself
  - Has a nasty nip:
    - Big pointy teeth
    - Can open jaws *really wide*
  - Not ideal first pet

Our code should be indented in the same way and for the same reason; to indicate its structure. If we look at the example program above we will see that all the lines start flush at the left except for the contents of the `{ ... }` block which are indented by the same amount. In this case it's two spaces, four spaces is also common. Notice how the closing `"}"` is brought back to the left, level with the code outside of the `{ ... }` block.

When we come to write functions containing loops and conditionals ("`if` statements"), their contents will also be indented by the same amount, thus making the logical structure of our function clear at a glance.

We also left a couple of blank lines before the start of the function and this is also good practice. Minor sub-divisions within a function can be denoted by a single blank line.

**This is extremely important** and teams of programmers have instructions ("style guides") that specify exactly how code should be formatted.

In general C doesn't care about white-space at all outside of character strings, although the compiler will get upset if we try to break up words ("`printf`") or numbers ("`1.2 34`").

Also remember that we will sometimes need to print our program, so follow the google guidelines and limit our lines to eighty characters. We can break long statements over several lines, making sure that the continuation lines are themselves properly indented.

We are also allowed to have more than one statement on the same line but again this is highly disapproved-of.

**The markers will be instructed to be extremely strict on this** as it's very important and not hard to get right.

The use of so-called **white space**, including blank lines, to visually indicate structure has been used in books for centuries.

The google style guide specifies that `{ ... }` blocks are indented by **two** spaces and that lines are a maximum of **eighty** characters long.

## Layout summary

- Indent the contents of `{ ... }` blocks by either two or four spaces, always using the same amount

- Bring the final `}` back in line with the outside of the block.
- Leave two or three blank lines between functions.
- Leave a single blank line between a function when we wish to emphasise internal divisions.

On the Macs lines will automatically start lined up with the line above. The **tab** key will move the text over by four spaces and the **delete** will move you back four spaces. Don't try to use spaces!

## Arithmetic and variables

The above code has one huge omission: it has no data, no numbers, etc. Now that we have learned the basics of what a simple program looks like it is time to do some useful calculations.

You should at least be aware that computers use "binary", or base-2 arithmetic rather than the "decimal" or base-10 that we are used to, that "binary digits" (zeros or ones) are called "bits" and that an ordered set of eight bits is called a "byte", although you will be relieved to hear that you don't need to be able to do binary arithmetic to program a computer.

As discussed in the preface, algorithms have words and phrases which are used as "place-holders" for the actual numbers:

- The **cost\_of\_the\_petrol** is the **cost\_per\_litre** multiplied by the **number\_of\_litres\_sold**.
- The **cost\_of\_the\_baked\_beans** is the **cost\_per\_tin** multiplied by the **number\_of\_tins\_sold**.

In computer programming these "place holders" are known as **variables**. The above example reminds us that some things (tins of beans, people) are treated as integer units, whereas others (petrol, distance) are allowed to be fractions. In the latter case all measurements are by necessity approximate.

Like most programming languages C reflects this by allowing two categories of variables: **integers** and **floating-point**, i.e. non-integers. (The name "floating-point" comes from the fact that in numbers such as 1.234 , 12.34 , 123.4 etc. the decimal point "floats" from left to right.) Floating-point calculations are very important for scientists and engineers so be sure to remember what the term means!

Variables have names which start with a letter (the underscore "\_" counts as a letter) followed by zero or more letters or digits. Conventionally only lower-case letters (and digits) are used, with upper-case letters being used for named constants.

**The compiler doesn't care what the variables are called**; the names are simply there to make things clearer for us, and the people who have to read our code later on.

### Floating-point variables

Whenever we take a measurement or use a calculator we should be familiar with the fact that using a finite number of decimal places (or binary places for a computer) limits the accuracy of the calculation. Most computers help deal with this conundrum by offering a choice of two precisions: *single precision* (four bytes per variable) and *double precision* (eight bytes per variable). In C these are known, somewhat inconsistently as **float** and **double** respectively.

We suggest using **doubles** and it's vaguely useful to remember they use eight bytes each.

C adopts a "better safe than sorry" approach and by default does most of its arithmetic in double precision anyway.

Just like on a calculator **floating-point calculations are always approximations to the mathematically correct result**.

### Declaring variables

Unlike in our turtle example, variables have to be **declared** before they can be used. The following snippet declares and then uses three doubles:

```
double materials, labour, total_cost;

materials = 9.4;
labour = 11.3;
```

The use of four and eight bytes for **floats** and **doubles** respectively is an optional appendix to the C99 standard. You will find it on everything from an iPhone upwards, but you may not find it on your mobile phone.

Some languages allow us to use a variable without declaring it but this causes problems with typing mistakes:

```
s1de = 7.3;
```

```
total_cost = materials + labour;
```

```
area = side*side;
```

You will notice that declarations also end in a semi-colon.

### Constants

Constants work pretty much as we would expect. Scientific notation is available using "E $n$ " for "ten to the power  $n$ ", where  $n$  is an integer. ('E' stands for **E**xponent.) In the following examples the numbers on the same row have the same value:

1.2345E2	123.45
1.2345E-2	0.012345
-1.0E1	-10.0

### Arithmetic expressions

Again, these work just as we would expect using the symbols "+ - \* /" for the arithmetic operations plus, minus, multiply and divide. They have the standard arithmetic precedence (\* / bind more closely than + -) and left-to-right evaluation. Parenthesis ( ) have the usual grouping effect. As above, in the following examples expressions on the same line have the same value **to within rounding errors**:

$x * y + z$	$(x*y) + z$	
$1.5 - x - y$	$(1.5 - x) - y$	
$x/y/z$	$(x/y)/z$	$x/(y*z)$

**Thirty-second discussion:** turn to your neighbour and discuss which of the three ways of writing the final expression ( $x/y/z$ , etc.) is the clearest. **Remember the mega-principle.**

Having calculated an expression we can use it as the argument to a function such as:

```
sqrt(x*x+y*y)
```

(no prizes for guessing what the `sqrt ( )` function does!). The most common use is probably the one we one we showed above, putting it on the right-hand side of "**variable =**".

**Example:** if we are writing a program dealing with a plane of gradient 1 in both the x and y directions (i.e. an absolute gradient of  $\sqrt{2}$ ), and we want to know the distance of a point on the plane from the

origin, then our code might look like:

```
double x, y, z, distance;

x = 2.1;
y = 1.3;
z = x + y;
distance = sqrt(x*x + y*y + z*z);
```

### Arithmetic assignment

It is important to realise that the statements with equals signs above such as:

```
z = x + y;
```

are **one-off arithmetic assignments**, not lasting mathematical relationships. (Remember, we are dealing with **algorithms** which are sequences of instructions or actions to achieve the desired result.)

Suppose we were to modify the example by later changing the value of **x**:

```
double x, y, z, distance;

x = 2.1;
y = 1.3;
z = x + y;
distance = sqrt(x*x + y*y + z*z);
```

z will not be exactly 3.4 because the computer works in binary and neither 1.3, 2.1, or 3.4 are expressible in a finite number of "binary places".

```
x = 10.0;
```

When it gets to the statement "x = 10.0;" the compiler will not think to itself "z equals x + y, so z is now equal to 11.3 and I must recalculate the distance". The assignment of z was a **one-off action** so z will remain (approximately) equal to 3.4 until we explicitly change it, and the same applies to distance.

Of course the compiler doesn't "think" at all and it certainly doesn't realise the significance of the word "distance". For all it cared we could have called the four variables (x, y, z and distance) "variable1", "variable2", "variable3" and "i\_feel\_like\_a\_banana". But the program would have been [much less clear to us](#).

## Standard mathematical functions

Putting the the following compiler directive at the top of our file:

```
#include <math.h>
```

(note the Americanism it's **math** not maths) gives us access to all of the usual mathematical functions such as:

Some useful mathematical functions in math.h	
function(s)	Notes
sin(x) cos(x) tan(x)	in radians
asin(x) acos(x) atan(x)	inverse sin (arcsin), etc.
sinh(x) cosh(x) tanh(x)	hyperbolic sine, etc.
asinh(x) acosh(x) atanh(x)	inverse hyperbolic sine (arcsinh), etc.
sqrt(x)	
log() log10()	natural log, log to base 10
pow(x,y)	$x^y$

All are doubles.

## Printing numbers to the screen

We have already seen the `printf()` function print a constant string of characters to the screen. To print expressions such as numbers to the screen we put a **format specifier** in the string. This consists of a percent character "%" followed by a letter to denote the type of the value to be printed. As you would expect, the format "%f" expects a floating-point argument.

For example, suppose we have two double variables, **a** and **b**. To print the value of **a + b** to the screen we say:

```
printf("a plus b equals %f\n", a + b);
```

Notice:

1. Inside the format string we see the characters "%f".
2. We have added a second argument to `printf()`, "a + b", which is separated from the first argument (the format string) by a comma.
3. The expression "a + b" is a **single** argument: the computer will calculate the value **a + b** when the statement is executed.
4. As you might expect, the output is the string with "%f" replaced by the value of **a + b**

Notice the difference: `\n` is used to represent a character which we can't conveniently type into the string (in this case "new-line"); `%f` is used to indicate another argument, in this case a floating-point

number.

There are three possible formats for printing a floating-point number depending on how exactly we want it to be displayed

- **%e** This prints a floating point number always using the exponent form: `1.2345e1`
- **%f** (seen above). This prints a floating point number in the form `12.345`
- **%g** This prints a floating point number making an intelligent choice between the above two formats depending on the value of the arguments. This is normally the best choice.

We can modify the above example to use `%g` and also to print out the values of `a` and `b`:

```
printf("%g plus %g equals %g\n", a, b, a + b);
```

Notice three "`%g`"s means we need three floating point arguments following the format string, separated by commas. The output is the format string with the value of `a`, the value of `b` and the value of `a + b` replacing the three occurrences of "`%g`".

## A complete example

The following example demonstrates a well-known trigonometric identity:

```
/*
 * Demonstrate the trigonometric identity
 * cos(a+b) equals cos(a)*cos(b) - sin(a)*sin(b)
 */
#include <stdio.h>
#include <math.h>

int main() {
    double a, b, cos_a_plus_b;

    a = 0.707; // Random values
    b = 0.1234;

    cos_a_plus_b = cos(a) * cos(b) - sin(a) * sin(b);

    printf("I calculate cos(%g) to be %g\n", a + b, cos_a_plus_b);
    printf("with an error of %g\n", cos_a_plus_b - cos(a+b));

    return 0;
}
```

### Things to notice

- The (fairly short) comment at the top tells us what is going on.
- We include the file `<stdio.h>` to use the `printf()` and the file `<math.h>` for `sin()` and `cos()`
- We have used the same variable names "`a`" and "`b`" inside the code as we did in the comment. Usually the context gives us an idea of a sensible name for a variable. (We could have used `alpha` and `beta` as our variable names instead.)
- Observe the ridiculously obvious name "`cos_a_plus_b`". We could have called it "`cos_a_b`" or "`cosab`" but you should avoid generic names like "`result`" or worst of all "`c`"! What would those names have told you?
- **We have included a test** and that we have made the test equal to zero for the precise case. This means that the test actually tells us whether it has worked OK, **that's what a test is for!**
- We have used correct indentation and blank lines.

Test cells that should be zero are also a great check to have in spreadsheets.

On my machine the program prints:

```
I calculate cos(0.8304) to be 0.674581
with an error of 1.11022e-16
```

with an error of  $-1.11022e-10$

## Summary of floating point variables and expressions

- Use type **double** variables for floating-point numbers.
- Variables must be declared before use.
- Variable names should start with a letter, then have more letters and digits, all lower-case.
- The compiler doesn't care about names but we should: choose a fairly short name that tells you what it means.
- **#include <math.h>** lets us use `sin()`, etc.
- Use **%g** in `printf()` to print a floating-point number (`%e %f` do too).
- Arguments to functions are separated by commas.
- C always passes **values of expressions** to functions, never variables.
- test things out!

## Afterwards

- Revise these notes, especially the three summaries.
  - Repeat the mega-principle to yourself so many times that you wake up having dreamed about it.
-