First Notes on C

Table of Contents

Level One

What is this text?

This is a set of notes introducing the programming language C. It is intended to explain the ideas involved, for people new to C. It is not a *reference work* - it does not cover every aspect of the language, in every version, nor all standard library functions, nor any third party libraries. Such a work would be tens of thousands of pages long, need to be updated every few months, and be incomprehensible to a beginner. It provides links to some sources of reference material.

Please email corrections to w.w.milner@gmail.com.

Level one and two

If we put the full details of everything to start with, it will make no sense. So in level one we outline common aspects of C, to enable the reader to practice writing and debugging lots of code.

Once some concrete practical experience has been obtained, we deal with more abstract ideas in level two.

Why learn C?

C began around 1970 with Brian Kernighan and Dennis Ritchie as they developed Unix. So it is 50 years old. Why study it now?

- It remains a very popular and widely-used language. Tiobe publishes a comparison of language use, and C has been in the number 2 position, after Java, for decades.
- Its syntax was the basis for many other languages, including C++, Java and Javascript, so learning C gives you a start in any of those.
- C uses raw exposed pointers, and these are more concrete and understandable than the more abstract approach in C++ and Java. If you are a beginner 'raw exposed pointer' makes no sense – just believe me.

C and C++

Many sites talk about 'C/C++', as if there existed some hybrid language. There is no such thing. C and C++ are very different languages, with different rules and different ideas. C++ includes OOP and the STL, which makes it a very large language.

This text is about C. It is not about C++.

Trying to use C++ if you do not know C is like trying to run a 100 meters sprint when you have not yet learnt to walk.

Tests

Do not just read this book.

Do not try to ‘learn’ it. Do not try to memorize it. Try to understand it.

By writing your own code.

Try to do all of the ‘tests’.

Reference and other help

A key text is 'K and R'. This is 'The C Programming language' by Kernighan and Ritchie, second edition (1978). You may be able to Google a pdf version. This is a central text in Computer Science. Stroustrup's 'The C++ Programming Language' relates to it, as does 'The Java Programming Language' by Arnold, Gosling and others. It does not cover the later standards, but it should be read.

comp.lang.c was a usenet newsgroup - a old type of text-only Internet forum predating the world-wide web. It continues as a Google group: https://groups.google.com/forum/#!forum/comp.lang.C

A set of frequently-asked questions from comp.lang.c is here

and a set of notes by Steve Summit here

Reference material should cover everything, but is not intended to explain things - it assumes you know the subject. A typical extract is 'A C program is a sequence of text files (typically header and source files) that contain declarations.' For a beginner this is useless, because they do not know what a text file is, what header and source files are, nor what declarations are.

The reference manual for gcc, a widely-used compiler, is here.

C has a standard, under ISO/IEC authority. Many of these documents are not free. A set of links to some versions, with some free ones are here.

A slightly less formal set of reference material is here.

Basic ideas

Idea of a compiler

Digital devices (desktop computers, laptops, notebooks, tablets, phones and so on) work because they contain a *processor*. Actually they may contain several processors, each one multi-core. The idea is the same. A processor is a chip which can recognise and carry out a set of instructions, such as add, subtract, move and compare *binary patterns*. Each instruction is itself a binary pattern, probably 32 or 64 bits long. A computer program contains many of these instructions.The program is normally stored in a file, and loaded into memory (RAM) before the processor executes it.

This type of program is called *machine code*. Coding in machine code is not a good idea, because:

1. It is impossible to remember the binary codes
2. Bugs would be very common
3. Coding would be very slow.
4. It would only work on one type of processor.

Instead of machine code, we do this:

We write programs in a *high level language*, such as C, C++, Java, Javascript, Kotlin, and hundreds of others. These do not use binary patterns. They use English words and make sense to a human.

1. We store the program in a text file (called *source code*)
2. We use a *compiler* to translate source code into machine code. A compiler is just a program itself, which reads source code, applies rules of syntax to it, and if possible, outputs the machine code in an executable file. If it is not possible, because of syntax errors, it outputs an error message.

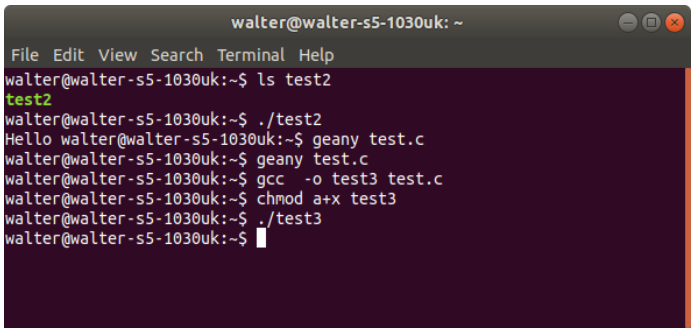
This can be done in other ways. In Javascript we use an *interpreter*, normally inside a web browser, and this takes one line of Javascript code, executes it, and moves on to the next. In Java the code is translated into bytecode, which is part way between source code and machine code. Then bytecode is executed in a Java runtime environment, which is another piece of software which is mostly an interpreter.

But in C we use a compiler.

Operating Systems

As well as a processor, a digital device needs an operating system (OS). Examples of current common OS are Windows (in several versions), and Linux (distributed in various packages such as Ubuntu, Debian, Suze, Redhat and so on, and used as the kernel on Android and MacOS). The OS is a bundle of software which is needed to make the hardware usable. An OS provides software to make a file system work, to provide security, to make multi-tasking work (so several pieces of software can run at the same time) and so on.

The OS will provide a *command line interpreter*. This allows the user to type in commands



Using a CLI

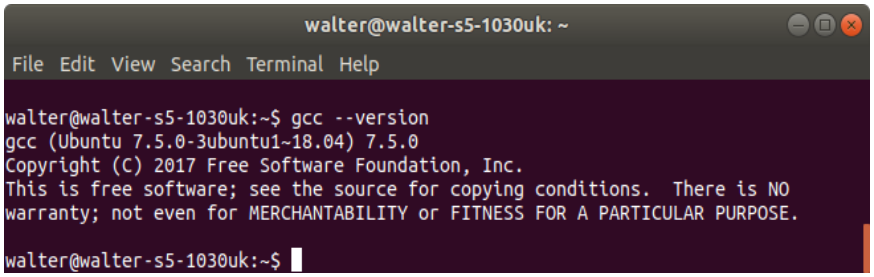
and get output as test.

The OS may also have a graphical user interface (*GUI*) with windows, icons, buttons, text boxes, a mouse, use of a touch screen and so on. Windows is named after its GUI. Linux can use a set of alternative windows managers.

Install a compiler

You need a C compiler. Which one? There are many free alternatives. Google 'C compilers' to get lists like https://en.wikipedia.org/wiki/List_of_compilers#C_compilers. In these examples we will use gcc, from the Free Software Foundation. Use a different compiler if you want - but you need to read its documentation to find out how.

If you are using Linux, you probably already have gcc. To find out, open a terminal (probably by Ctrl_Alt_T, depending on which Linux), and type in gcc -version:



Checking for gcc

If you get output like this, you have gcc installed.

On Windows you need to download and install it. Google for something like:

https://gcc.gnu.org/install/binaries.html

You want the binary - in other words, the program as an executable file. You could download source files, but then you need to compile them, and you do not have a compiler. You must select the version for your OS. The version for Windows is called MinGW.

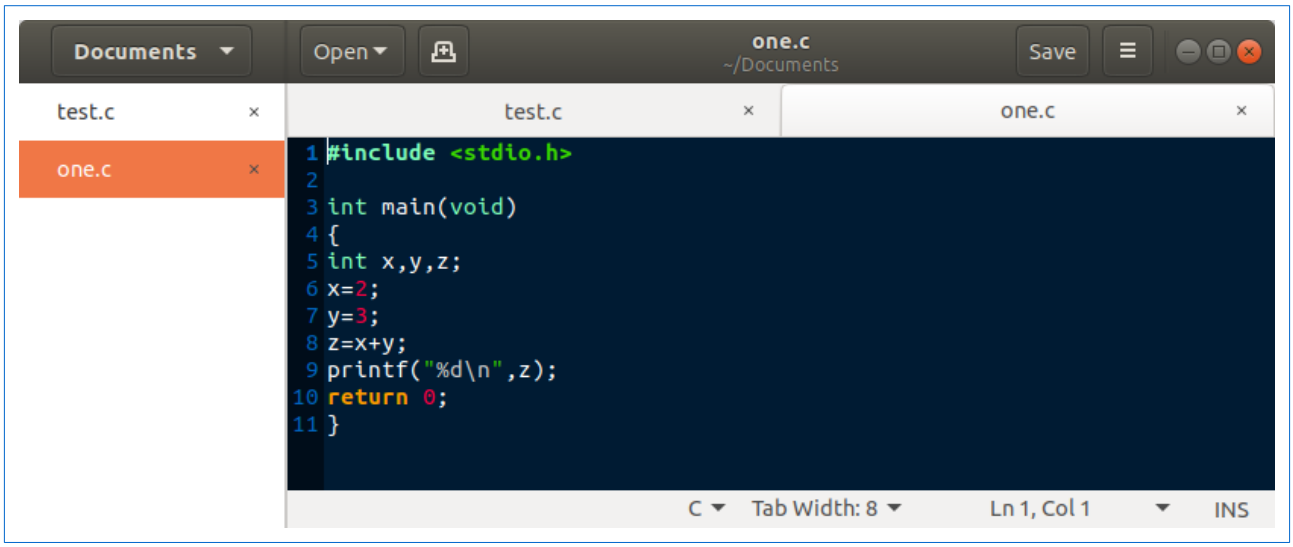
Write a program

High level language source code programs are text files, so to write a program, you need a *text editor*. On Windows you can use notepad, or install others. On Linux you can use nano or geany or gedit or similar.

In this example we use gedit. Start the editor and copy and paste this program:

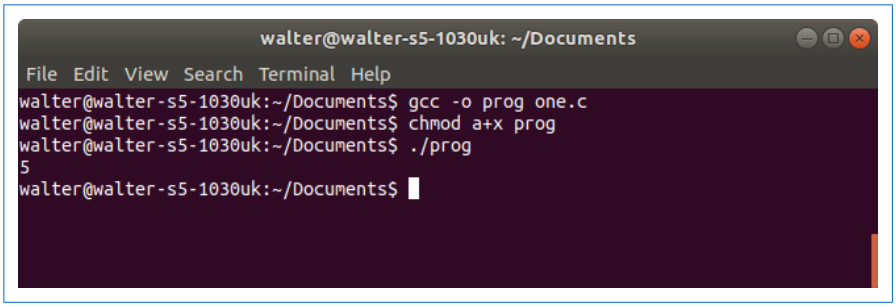
```
#include <stdio.h>

int main(void)
{
int x,y,z;
x=2;
y=3;
z=x+y;
printf("%d\n",z);
return 0;
}
```

Actually copy and paste it, rather than trying to type it in, because you will make mistakes. In gedit it will look like:



Save it, into a file named one.c

Then type in the following commands



What is happening?

`gcc o prog one.c`
runs the gcc compiler, telling it to compile the file one.c, and output the result to a file name prog.

`chmod a+x prog`
change the *permissions* on file prog, to make it an *executable* file. Then

`./prog`
actually executes it - getting the output 5.

How does the program work? We repeat it with comments:

```
#include <stdio.h> // needed to do output - stdio is standard input and output

int main(void) // execution starts here
{
int x,y,z; // declare 3 variables, x y znd z. An int is a whole number
x=2; // make x 2, y 3 and add them up
y=3;
z=x+y;
printf("%d\n",z); // output the value of z
return 0; // finished
}
```

A comment is text in a program which the compiler will ignore.

```
// This is a C line comment
/* Or a comment can
run over several lines
```

```
like this */
```

Reserved words and identifiers

A *reserved word* (or *keyword*) is a word which is *part of the C language* – such as void or return. There are only 30 or 40 reserved words (depending on the version) in C.

An *identifier* is the name of a variable (like x or y) or a function (like printf) or something else which the programmer can *choose the name of*.

- You cannot use a reserved word as an identifier
- You must meet some obvious rules – like no space in an identifier
- You should choose an identifier name which describes exactly what it is for. As an example if a variable is used to count files, it should be named fileCount.

Compile-time and runtime

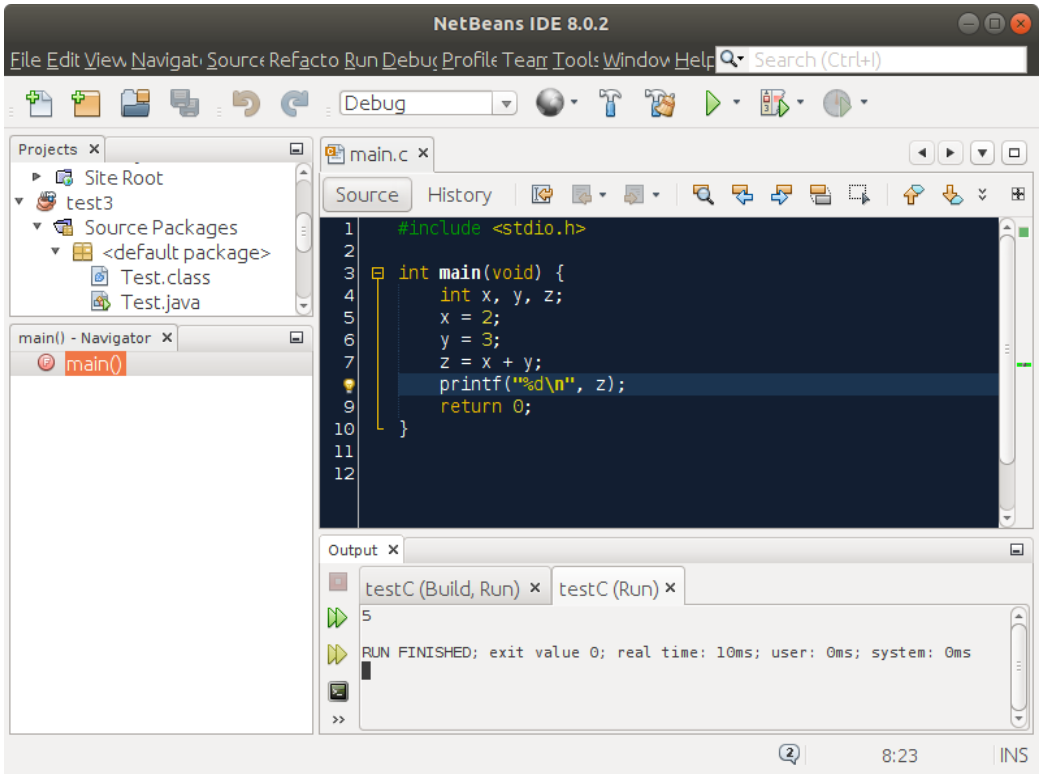
Compile-time means when we compile source code.

Runtime is when we run compiled code.

Different things happen at compile and run time.

Use of an IDE

Instead of using a text editor at the command line, we can use an integrated development environment (IDE). An IDE provides access to a



Netbeans with the C plugin

set of tools you need to develop code - a text editor, a compiler, a debugger and many other facilities.

IDEs can usually be set up to work with different languages and compilers. Some options are Netbeans, Eclipse and CodeBlocks.

This shows the same program in Netbeans

The output appears in the 'console' within the IDE.

Currently (April 2020) the latest version of Netbeans (Apache 11.3) has lost direct C support. Go Tools.. Plugin.. Settings and enable Netbeans 8.2 Plugin Portal to get access. Then go Available Plugins and install the C one.

Test

1. You need access to a desktop or laptop with a C compiler. Maybe you can use a school or college or university machine. If not you need something which will run a C compiler. That does not need to be very powerful or expensive.
2. You need to be able to work at the command line, so you need to know how to get to one, and do things like change directory, list files and folders in the current directory, delete files and so on. This is different for Windows and Linux. Google and learn for your OS.
3. Check if you have a C compiler. If not, install one. If you have Linux, you probably have gcc.
4. Copy, compile and execute the program given here.
5. Change it so it adds 13 and 15.

Declarations and assignments

Basic code template

To start with, our programs will have this structure:

```
#include <stdio.h>

int main(void) {
   // your code goes here..

    return 0;
}
```

Declarations

Our programs will have variables, which are pieces of data which usually change as the program runs. Variables have names and are held in memory.

Variables must be *declared before use*. For example this program:

```
#include <stdio.h>

int main(void) {
    x = 9;
    return 0;
}
```

will not compile. When we try to, we get this error message:

```
main.c: In function ‘main’:
main.c:4:5: error: ‘x’ undeclared (first use in this function)
     x = 9;
     ^
```

We fix it by *declaring* the variable:

```
#include <stdio.h>

int main(void) {
    int x;
    x = 9;
    return 0;
}
```

Why is that a rule? One reason is that it helps to prevent errors from mis-typed variable names. If we had typed xx by mistake where we meant x, the compiler will point out this error because xx has not been declared. It is much easier to find bugs if the compiler will point them out to us.

Test

Try out these two code fragments.

Code layout

C is *case sensitive*. So int and Int are different – and Int will not be understood.

Whitespace means spaces, tabs and newlines. The compiler first replaces all whitespace with a single space – so we get

```
#include <stdio.h> int main(void) { int x; x = 9; return 0; }
```

and your code could be like that. But that is a bad plan!!

The important thing is how *readable* it is – how clear and easy to undertsand, for a human being.

So set it out as we have done here.

You cannot put spaces in the middle of words – so main not ma in.

Types

In

```
    int x;
```

the 'int' is a *type*. An int is a whole number. Some C types are:

short, int, long

These are whole numbers, stored in different numbers of bytes - short is the smallest, and long is the biggest. Using a small type saves memory, but the range (smallest and biggest value) is limited.

float, double

These are numbers with decimal parts, stored as *floating point* format. A double uses more memory than a float, but has a bigger range. Arithmetic on floating point is *slower* and *not completely accurate*.

Representation

Everything in a digital system – from a mobile phone to a supercomputer – is in the form of binary patterns, streams of bits like 1100 0101. This includes program code, text, numbers, pixels, sound files, movies and everything else.

Representation means *how* that is coded into binary patterns.

For whole numbers, some standard forms of representation are called sign and magnitude, one's complement and two's complement.

For floating point, two methods are binary-encoded decimal and IEEE754 exponent and mantissa format.

Standard file formats, like jpeg and gif and avi and mp3, set out how the bits in a file are used.

The C language does not specify any method of representation. Unlike Java, it is *implementation-defined*. In other words if we take the same C program, and compile it for use on different platforms, data might be represented in a different way.

But we need to keep in mind the idea that all this stuff is actually sets of binary patterns.

Some notes on number bases, bits and so on are here.

For constant values use const

We can declare and assign them for example

```
const int BUFFER_SIZE = 1000000;
```

This creates a variable of type int, with a name BUFFER_SIZE, and a value 100000.

Declaring it to be const means it is a constant, and code cannot assign a new value later on (or the compiler will flag it as a syntax error).

But why not just use 100000 in our code, instead of BUFFER_SIZE? Because

- It is more readable = makes more sense. Someone would ask 'why 100000', while if they see BUFFER_SIZE they know it is.. well, the size of the buffer.
- It is easily changed. If we want to try it out with a buffer size of 200000 we need only change one line, but if we have written 100000 in code throughout the program, we must find and change every occurrence.

It is usual to name constants UPPER CASE.

Output - printf

The printf function is not part of the *C language* - it is part of the *standard library*. To use it we need

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

We explain this later.

The function name means 'print, formatted'. It is designed to output zero or more data values to the console or command line, as a stream of characters, together with a string.

An example with no data is

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

The string to be output is in "double quotes". The \n is a *control character*, a new line character. So this outputs Hello World then goes on to the next line.

If we want to output data, we need to tell it what data, and how to format it. For example

```
    int x=9;
    printf("x = %d\n", x);
```

The %d is a *format specifier*, which says how to convert the data into characters for display. The data (x) replaces the format specifier in the string (%d means decimal integer) so this outputs

x = 9

For a double floating point type we use %lf

```
    double y=6.71;
    printf("y = %lf\n",y);
```

We can control the format like this

```
    double y=6.71;
    printf("y = %3.1f\n",y);
```

The 3.1 means a total width of 3 characters, with 1 decimal place, so we get

```
y = 6.7
```

Suppose we use the wrong format specifier?

```
    double y=6.71;
    printf("y = %d\n",y);
```

Then we get a *warning*:

```
main.c:6:18: warning: format ‘%d’ expects argument of type ‘int’, but argument 2
has type ‘double’ [-Wformat=]
     printf("y = %d\n",y);
                  ~^
                  %f
```

and weird output:

```
y = -404793432
```

C assumes you know what you are doing, and does what you tell it. At y is stored a sequence of bytes in floating point representation. If we tell it to treat those bytes as an int, we will get an incorrect output.

And a warning (not a syntax error). In general, if you get a warning, you have done it wrong and need to fix it.

More on printf later.

Assignment statements

Something like

```
    x = 9;
```

is an *assignment statement*. When executed, the right-hand side (9) is evaluated (pretty easy) and the result is stored in memory where the left-hand side is kept.

This is one kind of statement. Statements have a semi-colon ; at the end.

The left-hand side must be a variable name, nothing more. If we say

```
    x+1=9;
```

we get a syntax error:

```
main.c:5:8: error: lvalue required as left operand of assignment
     x+1=9;
        ^
```

An lvalue is something which can be on the left.

Expressions

We can also say something like:

```
#include <stdio.h>

int main(void) {
    int x;
    int y;
    y=1;
    x=4*(y+9);
    printf("%d\n",x); //40
    return 0;
}
```

Something like

4*(y+9)

is an *expression* - something which will be evaluated at runtime.

An expression can include *operators*, like * and + and -, and *operands*. An operand is a variable (like y) or a constant (like 4 or 9).

The usual operators are *, /, + and -. These obey BODMAS and you can use (round brackets) to change that.

% is the remainder operator. So 13%3 is 1,because 12 divided by 3 is 4, remainder 1. 6%3 is 0, 17%5 is 2.

/ works differently on ints and floats. For example

```
int x;
x=10/3;
printf("%d\n",x); // 3
double d;
d=10.0/3.0;
printf("%lf\n",d); // 3.3333
```

so / on ints like 10/3 is the quotient. 3 divided into 10 goes 3 times. We have a 1 remainder, but that it discarded.

/ on floats and doubles gives a double result.

% is remainder, on ints. So

13 % 3 is 1
15 % 3 is 0

8 % 2 is 0

9 % 2 is 1

Test

Write programs that

1. Output your name

2. Declare int variable named one and two. Make one 45. Make two double one plus 17. Output two to check it.

3. Calculates the circumference of a circle radius 3.5 cm.

Type casts

A type cast changes the type of something.

In C we cannot change the type of a variable. But we can sometimes change the type of a value. For example:

```
int i=3;
double d = i;
```

This casts the value of i, an int, to type double.

Or

```
double d = 3.0;
int i=d;
```

casts a double to an int. In fact

```
double d = 3;
```

is a type cast from an int (3) to a double. This is different from

```
double d = 3.0;
```

because 3.0 is already a double value.

These are all *implicit type casts*. This is when the code produces a type cast without actually saying so. An *explicit type cast* is for example

```
int i=3;
double d = (double) i;
```

where (double) means 'change the value on the right to double type'.

Use explicit type casts, to show you know what you are doing.

How does the cast happen?

```
int i;
double d = 3.1;
i = (int) d;
printf("%d\n", i); // 3
d = 3.9;
i = (int) d;
printf("%d\n", i); // 3
```

So in fact this is a *truncation* – it just cuts off the decimal part. This is why type casts need to be used with caution – we may be losing data.

To do *rounding* – change to the nearest integer – add 0.5:

```
int i;
double d = 3.1;
i = (int) (d+0.5);
printf("%d\n", i); // 3
d = 3.9;
i = (int) (d+0.5);
printf("%d\n", i); // 4
```

To round to the nearest hundred, say – divide by 100, round, then multiply by 100:

```
int i;
double d = 3278.4;
double temp = d / 100.0;
i = (int) (temp + 0.5);
i =i* 100;
printf("%d\n", i); // 3300
```

Short-cut operators

We often want to do things like multiply a value by 3. We can do this by

```
x=3*x;
```

but a better version uses a *short-cut operator*

```
x*=3;
```

which means the same thing.

Why is it better? Because it probably compiles to faster machine code. Others are

Short-cut	Means
x+=4;	x=x+4;
x-=9;	x=x-9;
x/=5;	x=x/5;

Pre and post increment

We often want to increase a value by 1. As in

```
x=x+1;
```

We can say

```
x+=1;
```

but better (faster) is

```
x++;
```

This is *post-increment*. Why?

```
    int x=7;
    int y;
    y=x++;
    printf("x=%d and y=%d\n",x,y);
```

outputs

x=8 and y=7

In

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

we increment x. But we also use the value to assign to y. The question is - is that the old or the new value? Answer is the old value. So

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

means use x, then *after that*, increment it.

Pre-increment is ++x - meaning increment it, *then* use the new value:

```
    int x=7;
    int y;
    y=++x;
    printf("x=%d and y=%d\n",x,y);
```

```
x=8 and y=8
```

Characters

A character is a symbol used in text. Characters include letters of the alphabet, digits 0 to 9, punctuation marks like ?, symbols like # and many more. A space is just another character.

A character set is a set of characters. Different character sets are used, with names like EBCDIC, ASCII and Unicode. In each character set, each character has unique integer - its *codepoint*. For example in ASCII the codepoint of character A is 65, B is 66, C is 67, 0 is 48, a space ' ' is 32.

The *C language does not use any particular character set*. In other words it is implementation dependent. One system might use ASCII, another Unicode. ASCII is very common – but you cannot assume it.

Digital systems store characters as their codepoints. In C, the character type is named 'char'. So a char is actually a small integer.

For example

```
#include <stdio.h>

int main(void) {
    char char1 = 'A';
    printf("%c\n", char1); // %c format as char - get A
    printf("%d\n", char1); // %d format is int - get 65
    char char2=char1+1;    // some arithmetic on a char
    printf("%c\n", char2); // get B
    printf("%d\n", char2); // get 66

    return 0;
}
```

A *string* is a sequence of characters. We look at strings later.

Test

Write programs to find out

1. Can you use pre-increment on a char type? What does it do?

2. If you output the chars ‘A’ ‘B’ and ‘C’ as integers, what do you get? Why?

Conditionals

We often want code to do different actions in different cases - for example do one thing if x is greater than 10, and something else if x is less than 10. This type of statement is a conditional, or an 'if statement'.

The structure must be like

```
if ( something true or false )
{
.. code to do if true
}
```

For example:

```
#include <stdio.h>

int main(void) {
    int x = 20;
    if (x > 10) {
        printf("Bigger than 10 \n");
    }
    return 0;
}
```

This might seem pointless - we can see x is bigger than 10, so why bother? In real code the condition (like x>10) would depend on some other computation, maybe based on data input or reading a file, so at compile-time we do not know whether it is true or false.

Check:

Must be round brackets, so

if [x>10]

is a syntax error

No semi-colon, so

if (x>10);

is wrong

This uses a *block or compound statement*:

```
if (x > 10) {
        .. this is a block
    }
```

and we can have several statements in a block, like

```
if (x > 10) {
        a=9;
        z=7*a;
        b=x+y;
    }
```

Suppose the condition is false? Then the block is not executed, and we just go on to the next statement. But maybe we want to do something if it is true, and something else if it is false? Then we can say:

```
if (x > 10) {
        printf("Bigger than 10 \n");
    }
else
   {
    printf”Its not \n”);
   }
```

Test

Write a program which assigns a value to an int variable y, then outputs whether or not it is even (use %)

True and false

False is represented by 0, and any non-zero value is true. For example:

```
int main(void) {
    int x = 1 <10;
    printf("%d\n",x); // 1 - its true
    x= 5<4;
    printf("%d\n",x); // 0 - its false
    return 0;
}
```

This line

```
    int x = 1 <10;
```

is pretty unusual, but OK.

Logical operators

An arithmetic operator is for example * or +. That is, something which does arithmetic.

A logical operator gives a result which is true or false - like < for less than:

```
if (x<10)...
```

Here they are:

Operator	Means	Example
<	less than	
>	greater than	
= =	equal to	not =
=<	equal to or less than	
=>	equal to or greater than	
!=	not equal to	
&&	and	(x<y) && (z==9) x less than y AND z equal to 9
||	or	(x==y) || (y==z) x equals y OR y equals z
!	not	!((x==2) && (x==3)) not(x is 2, AND x is 3)

= is assignment. = = tests for equality.

Humans are poor at logic. !((x==2) && (x==3)) for example. x cannot be both 2 and 3 at the same time. So (x==2) && (x==3) is always false, and !((x==2) && (x==3)) is always true.

Test

1. Google de Morgan's Laws

2. Write a program showing that they work

Loops

Suppose we want to:

- Delete every file in a folder, or
- Set every pixel in a row blue, or
- Increase the price of every item in a supermarket by 5%

Each of these means repeating actions, which is a very common requirement. We do this by using loops, which *repeat code*. C has three kinds of loops.

while..

For example

```
#include <stdio.h>

int main(void) {
    int x = 1;
    while (x < 11) {
        printf("%d ", x); // 1 2 3 4 5 6 7 8 9 10
        x = x + 1;
    }

    return 0;
}
```

This has

- Initialisation : x=1
- What to change every time: x=x+1;
- When to continue: while (x<11)

A while loop might execute zero times. In this example, if x started at 20, the loop would not execute at all, so the condition is false the first time in.

Test

Write programs with while loops that

1. Output your name 10 times

2. Outputs 5 6 7 8 9 10

3. Adds up 3,6,9,12.. 36

do.. while

For example

```
#include <stdio.h>

int main(void) {
    int x = 1;
     do {
        printf("%d ", x); // 1 2 3 4 5 6 7 8 9 10
        x = x + 1;
    } while (x < 11);

    return 0;
}
```

So this is very similar, except that a do while will always execute at least once - since we only check the condition at the end of the first time through.

Test

Write programs with do while loops that

1. Output your name 10 times

2. Outputs 5 6 7 8 9 10

3. Adds up 3,6,9,12.. 36

for

```
#include <stdio.h>

int main(void) {
    int x;
    for (x=1; x<11; x=x+1)
    {
        printf("%d ", x); // 1 2 3 4 5 6 7 8 9 10

    }
    return 0;
}
```

In the for loop header

```
    for (x=1; x<11; x++)
```

we have the 3 parts:

- initialisation, x=1;
- when to continue, x<11
- what to change, x=x+1

We often use post-increment:

```
    for (x=1; x<11; x++)
    {
```

```
        printf("%d ", x); // 1 2 3 4 5 6 7 8 9 10

    }
```

We can also declare the loop counter in the for:

```
int main(void) {

    for (int x=1; x<11; x++)
    {
        printf("%d ", x); // 1 2 3 4 5 6 7 8 9 10

    }
    return 0;
}
```

Test

Write programs with for loops that

1. Output your name 10 times

2. Outputs 5 6 7 8 9 10

3. Adds up 3,6,9,12.. 36

break and continue

These 2 instructions work in comnnection with loops.

Break breaks out of the current loop

continue skips the rest of the loop body and repeats

For example:

```
    const int FOREVER=1;
    int index=0; // counter
    while  (FOREVER)
    {
        index++;
        if (index==15) break; // finish when 15
        if (index%3 != 0) // skip rest if not multiple of 15
            continue;
        printf("%d\n", index);
    }
```

so this outputs 3 6 9 12

switch

Related to an if is a *switch statement* (not a loop). This gives you a set of alternate statement blocks, depending on the value of some variable. For example:

```
    int x;
    int y;
    x=0;
    switch (x)
    {
        case 0:
            printf("x is 0\n"); // single statement
        case 1:
        { // or a block
            printf("x is 1\n");
            y=2;
```

```
        }
        default:
            printf("x is not 0 or 1");

    }
```

The idea is that if x is 0, it will do the first printf. If x is 1, it will do the block. And if it is anything else, it will do the default.

But – switch has a *fall-through behaviour*. If we run this the output is

x is 0

x is 1

x is not 0 or 1

This is because when any case is true, it and *all following cases* execute.

Because of this, it is usual to end each case with a break:

```
    int x;
    int y;
    x = 0;
    switch (x) {
        case 0:
            printf("x is 0\n");
            break;
        case 1:
            printf("x is 1\n");
            y = 2;
            break;
        default:
            printf("x is not 0 or 1");

    }
```

The 'controlling expression' (x in this example) needs to be an integer type.

How to write code

You need:

1. To know what the code is supposed to do. The jargon is a *requirements specification*. That just means – what is it supposed to do.

2. That is, what *output should be produced, for a given input*. Not *how* it should do it – simply *what* it should do. Some output would be correct. Others would be wrong. You must know what the correct output is, or you have no chance of writing the code.

3. You must find a way of doing that, in terms of computer processes. The jargon would be an *algorithm*. That just means – how? Not in terms of C, but how could you do it with pencil and paper. We can use loops, conditionals and assignments.

4. If its a big problem – split it into stages – smaller problems. This is called *problem decomposition*. Code one stage at a time.

5. Each stage must be *small*.

6. You must *test* each stage.

As an example – suppose the problem is to output the sum of the even numbers between 2 input limits.

So if the input is 2 and 6, the output should be 2+4+6 = 12.

If the input is 1 and 9, the output should be 2+4+6+8 = 20

How to do that? We could do it in two stages:

1. Firstly just go through the even numbers in a loop, then

2. Add up those numbers

How to do (1)? We could use a while loop. How to just get the even numbers? We could either go through all numbers, and check if each is even and ignore it if it is not, or go up in steps of two.

We will go up in steps of 2. But suppose the input is odd – like 1 and 9? We could adjust that to 2 and 8, then go 2 4 6 8:

```
   int start = 1;
   int end = 9;
   if (start%2==1) start++; // start is odd
   if (end%2==1) end--; // end is odd
```

Is that correct? Or not? We *test it*:

```
   int start = 1;
   int end = 9;
   if (start%2==1) start++;
   if (end%2==1) end--;
   printf("%d %d", start, end); // 2 8
```

Now we loop up in steps of 2:

```
..
int index=start;
  while (index <= end)
      index+=2;
```

Is that correct? We *test it*:

```
   int index=start;
   while (index <= end)
   {
       printf("%d", index);
       index+=2; // 2 4 6 8
   }
```

Now we go on to stage 2 – adding them up. We need another variable, initialised to 0:

```
   int sum=0;
   int index=start;
   while (index <= end)
```

```
    {
        sum+=index;
        index+=2;
    }
```

Is that correct? We test it:

```
    int sum=0;
    int index=start;
    while (index <= end)
    {
        sum+=index;
        index+=2;
    }
    printf("%d", sum);    // 20
```

and compare that with the expected output of 2+4+6+8 = 20.

Testing

Testing is a major part of software development. To test code means to find the bugs in it.

Code can fail in three ways:

- It fails to compile. That means there will be an error message, telling you where the error is and what it is. Compile-time syntax errors are easy to fix – the compiler tells you what the problem is. Just read the error message.
- It explodes when you run it. In C this will be a segmentation fault. This is harder to fix, because you have no clue as to what the problem is. It will be something you wrote, between now and the last time you ran it and it worked. So run the code frequently.
- It runs with no errors – but the output is incorrect. Again this is hard to fix. It means you have an algorithm error – your method of solution is wrong.

Code can be tested by designing input data and expected output, and checking that against actual output. For example in this case:

Input	Expected output	Actual output	Case
2,8	2+4+6+8 = 20		Normal input
3,9	4+6+8=18		Odd numbers
4,4	4		Start=end
6,4	0		Start > end

Debugging techniques

All bugs are written by programmers.

If your code has a bug – start by admitting it has a bug. People often start denying there is a bug.

Use logic to work out *where* the bug is, then *what* it is, then find a *solution*.

Ideas -

Print debugging – print out key values at key points. Compare what they are with what you think they should be. If they differ, work out why.

Commenting out code. Put sections of your code as comments. Some of your code makes it go wrong – the idea is to find out which part. You comment it out so you can easily put it back when fixed.

Using a *debugger*.

A debugger is a piece of software intended to help debugging code. Most IDEs have a debugger available. Typical tools are:

- Run to cursor. Execute code at full speed up to the cursor, then pause there.
- Set a breakpoint. Run your code at full speed up to the breakpoint, then pause there.
- Set watch variables. This is one or more variables which you can track the value of, and see the actual values and compare them with what you think they should be. Like print debugging
- Single step. One instruction is executed at a time on each key press.
- Stepover. This means to step over functions – just execute function calls in a single step, not..
- Stepinto. Go into function calls and single step through the code inside the function

Coding style

The web is full of rules about coding style. An early one is Indian Hill. Here are a few suggestions:

- Choose identifier names carefully, to simply say exactly what they are for.
- Avoid i and l. The characters 1, |, !, I, [and l look too similar. Use index and length or suchlike.
- Use comments to explain your code if it is at all obscure and not obvious.
- One step at a time – one action per statement

- One statement per line

Arrays

We often want to use a set of related data values, not just one. This means we need to use some *data structure*.

A simple data structure is an *array*. This is a set of values accessed through an *index*:

```
    int numbers[4]; // declare the array
    numbers[0]=7;   // put some values in
    numbers[1]=64;
    numbers[2]=13;
    numbers[3]=6;

    printf("%d\n",numbers[2]); // 13
```

So now numbers is a block which can hold 4 ints. In

```
    numbers[2]=13;
```

the index is 2. We store 13 in the array element with index 2.

In C:

1. an array has fixed size, fixed at compile-time (cannot grow or shrink)
2. has elements all the same type
3. starts with index 0 (not 1) so numbers[4] goes from 0 to 3 inclusive

We must use [square] brackets for an index.

Suppose we do this:

```
    int numbers[4];
    numbers[4000] = 6;
```

we get:

```
RUN FINISHED; Segmentation fault; core dumped; real time: 240ms; user: 0ms;
system: 0ms
```

We have made space for 4 ints, then tried to write into memory 4000 spaces further along. A *segmentation fault* is what the OS does when it finds one process (our program) tries to write into memory owned by another process.

Initialisation

We can do this:

```
    int numbers[4] = {2,3,4,5};
    printf("%d\n",numbers[1]); // 3
```

to initialise an array with starting values.

Loops and arrays

We very often process arrays in loops. We give several examples:

Add up an array

```
    int numbers[4] = {2,3,4,5};
    int total=0; // initialise
    for (int index=0; index<4; index++) // for each index
    {
        total+=numbers[index]; // add element into total
    }
    printf("%d\n",total); // 14
```

The variable 'total' means the total so far.

1. We initialise that to zero.
2. Then the for loop goes through all array index values, and
3. For each one, adds it into the total

Find biggest in an array

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

int main(void) {
    int numbers[5] = {2,32,4,55,9};
    int biggest=INT_MIN;
    for (int index=0; index<4; index++)
    {
        if (numbers[index]>biggest)
            biggest=numbers[index];
    }
    printf("%d\n",biggest); // 55

    return 0;
}
```

The idea here is that biggest means the largest element found so far. We go through the array and

```
        if (numbers[index]>biggest)
```

checks whether we have found a bigger one. If so, we change biggest to this one.

How to initialise biggest? It needs to be less than any value we might find. INT_MIN is the smallest possible int on this system, so we start it at that. INT_MIN is in the header file limits.h, so we #include that.

Here is another way:

```
#include <stdio.h>

int main(void) {
    int numbers[5] = {2,32,4,55,9};
    int biggest=numbers[0];
    for (int index=1; index<4; index++)
    {
        if (numbers[index]>biggest)
            biggest=numbers[index];
    }
    printf("%d\n",biggest); // 55

    return 0;
}
```

Here we initialise biggest with the first array element, and start the loop at the second element.

Output even numbers

```
#include <stdio.h>

int main(void) {
    int numbers[5] = {2,32,4,55,9};

    for (int index=0; index<4; index++) // go through the array
    {
        if (numbers[index]%2 == 0) // if even..
            printf("%d ", numbers[index]); // 2 32 4
    }
    return 0;
}
```

Reverse an array

The first stage is how to exchange two values. We need to use a third storage place:

```
#include <stdio.h>

int main(void) {
    int x=9;
    int y=4;
    int temp; // temporary storage location
    temp=x; // copy old value x
    x=y;
    y=temp; // get old x into y
    printf("x=%d and y=%d\n",x,y); // x=4 and y=9
    return 0;
}
```

Then we use this idea on an array. We go through the array, exchanging each one with the corresponding value at the 'other end' of the array. We only go half-way through:

```
#include <stdio.h>

int main(void) {
    int data[10] = {6,1,7,2,6,5,6,2,8,8};
    for (int start=0; start <5; start++) // go half way through
    {
        int other = 9-start; // index at other end
        int temp; // we exchange start and other
        temp=data[start];
        data[start]=data[other];
        data[other]=temp;
    }
    // check
    for (int index=0; index<10; index++)
        printf("%d ",data[index]); // 8 8 2 6 5 6 2 7 1 6

    return 0;
}
```

Test

Write programs to:

1. Find the smallest number in an array

2. Output all odd values

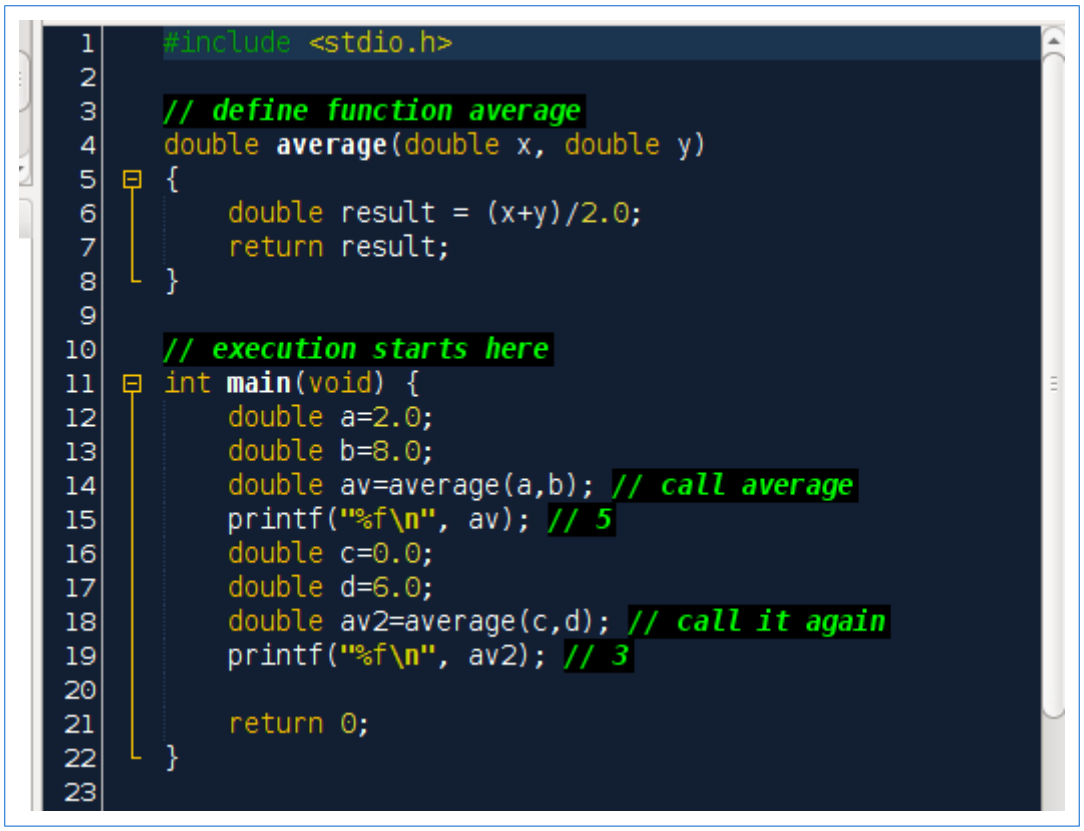
3. Reverse an array twice and show you get the original.

Functions

In C, the basic unit of code is a block called a function. So far we have just had one function, named main. Here is a program with two functions:

```
#include <stdio.h>

// define function average
double average(double x, double y)
{
    double result = (x+y)/2.0;
    return result;
}

// execution starts here
int main(void) {
    double a=2.0;
    double b=8.0;
    double av=average(a,b); // call average
    printf("%f\n", av); // 5
    double c=0.0;
    double d=6.0;
    double av2=average(c,d); // call it again
    printf("%f\n", av2); // 3

    return 0;
}
```

Here is the same in Netbeans, with line numbers:



We check through several aspects of this:

Function execution sequence

The code starts running at line 11. C code always starts running at the special function named *main*. We run on to lines 12, 13 and so on. Line 14 refers to the function named average. So here we *call* average. That means execution switches up to line 4, where

average starts, and runs to 7, where there is a *return* statement. That means we return to where it was called from - that is line 14.

Then we continue from there, lines 15,16,17 and 18. Here we have another call to average, so it switches up to 4 again, and returns at 7. It goes back to 18 (not 14).

Then 19 and 21, where the return ends main, and so ends the application.

Data flow

We pass data *into* the function average, and get data back *from* it. Data goes *into* the function through the *parameters* (also called *arguments*) named x and y:

```
double average(double x, double y)
```

Check - we need to give the types of the parameters. The function does some computation, then data comes out of the function by the return statement:

```
    return result;
```

so the returned value is the alue of the variable named result. What happens to the returned value? At line 14, it is assigned to av

```
  double av=average(a,b);
```

The function is called again at line 18:

```
  double av2=average(c,d);
```

Here variables c and d are passed *in*, and the returned value back *out* this time is assigned to av2.

Local variables

In function average, we declare and assign a variable named result:

```
    double result = (x+y)/2.0;
```

In main, we declare variables named a and b

```
int main(void) {
    double a=2.0;
    double b=8.0;
    ..
```

These variables are local to those functions. This means they only have use within those functions. We check by altering the code:

```
// define function average
double average(double x, double y)
{
    double result = (x+y)/2.0;
    return result;
}

// execution starts here
int main(void) {
    double a=2.0;
    double b=8.0;
    double x=3.0;
    double result=9.0;
    double av=average(a,b); // call average
    printf("%f\n", av); // 5
    printf("x=%f result=%f\n",x,result);  // 3 9
```

```
    double c=0.0;
    double d=6.0;
    double av2=average(c,d); // call it again
    printf("%f\n", av2); // 3

    return 0;
}
```

We have declared variables x (used as a function parameter) and result (used as a local variable) in main, as well as average. But they keep those values. They are not affected by changes to them in average. So local variables in one function are unrelated to those in other functions, even if they have the same name.

Argument names do not matter

The function has parameters named x and y:

```
double average(double x, double y)
```

These are sometimes called *formal parameters*. When the function is called, the parameters are named a and b (the *actual parameters*):

```
    double av=average(a,b);
```

The names x and y, and a and b, do not matter. All that matters is the order. The value of a is passed to x, and b is passed to y.

Later on we say:

```
    double av2=average(c,d);
```

This time c goes to x, and d goes to y.

Function syntax

The function header is:

```
double average(double x, double y)
```

The name of the function is average. This identifies it, and we cannot have two functions with the same name (but we can call a function many times).

The parameters are in a list in round brackets, separated by commas, saying the type of each parameter.

The header starts with the return type. This needs to match the type of the value returned, and the type the returned value is assigned to at the calls.

Some functions do not return a value. Instead they just do some computation and change something (like deleting a file) and do not need to send any data back. These functions are declared to be type *void*.

Why use functions?

1. To *re-use code*. In our example we calculate the average twice. We do not need to repeat the same code twice.

2. To *simplify tasks*. If our main has a million lines of code - that is impossible to manage. We split the problem into parts, each part being handled by one function. A function can call other functions, so we can further divide tasks if needed. We break them down into small simple tasks.

3. One function can be used in *different applications*. For example, we might write a function that deletes a file. We could use that function in many applications.

4. A set of functions which would be useful in many applications is a *library*.

5. An *OS* contains in effect a very large library of standard functions (like deleting a file).

If a block of code is more than 20 or so lines, it is probably too complex and should be split into smaller functions.

Test

Write programs with functions that:

1. Find the average of 3 numbers

2. Returns the smaller of 2 numbers

3. Returns 1 if its 2 arguments are equal, and 0 if they are not.

4. Prints your name. Call it 10 times.

5. Return 1 if its int argument is prime, and 0 if it is not.

Scope and global data

Here is a different version:

```
#include <stdio.h>

    double a=2.0;
    double b=8.0;

double average(double x, double y)
{
    double result = (x+y)/2.0;
    return result;
}


int main(void) {

    double av=average(a,b); // call average
    printf("%f\n", av); // 5
    a=6.0;
```

```
    av=average(a,b);
    printf("%f\n", av); // 7

    return 0;
}
```

Here a and b are not declared inside main, so they are not local variables. They are declared at the start of the code, outside any function, so they are *global*. We can use their values anywhere, in main or any other function, and also change them from any other function, as shown here.

Another version:

```
int main(void) {
    double a;
    a = 2;
    b = 6;
    double av = average(a, b); // call average
    double b;
    printf("%f\n", av); // 5


    return 0;
}
```

This will not compile - it tells us b is not declared. In fact it is, but after it is referred to. And the C compiler just looks at code once, from the top down.

The *scope* of a variable is the region of source code where the variable can be used. For a variable declared globally, it is from the declaration to the end of the code. For a local variable, it is from declaration to the end of the function.

For example

```
#include <stdio.h>
double a; // scope of a starts here

double average(double x, double y) {
    double result = (x + y) / 2.0;
    return result;
}

int main(void) {

    a = 2;
    double b; // scope of b starts here
    b = 6;
    double av = average(a, b); // call average
    double b;
    printf("%f\n", av); // 5
    return 0;
} // b scope ends here
// a scope ends here
```

In fact scope can be narrower than a function - it can be local to a block:

```

double average(double x, double y) {
    double result=0;
    { // new block here
        double value1=result+x; // scope of value1 starts here
        value1+=y;
        value1/=2;
        result=value1;
```

```
    } // value1 ends here, with block
    printf("%f\n", value1);  // no
    return result;
}
```

Local scope is good. Why?

We do not need to remember the names of all the variables used in an application, in case of a clash. If they are local, they do not affect each other.

It means functions are largely self-contained. Data goes in through parameters and out through return values. This means that a bug in one function will not make another function go wrong. Functions can be written and tested one at a time.

Test

1. Explain to a fellow student what *global* means.

2. Explain what *scope* is.

Function declarations

In fact we need to declare functions as well as variables. For example:

```
#include <stdio.h>

int main(void) {
    int a=7;
    int b=9;
    printf("%d\n", bigger(a,b));
    return 0;
}

int bigger(int n, int m)
{
    if (n>m) return n;
    else return m;
}
```

When we compile this we get the warning:

```
main.c:8:20: warning: implicit declaration of function ‘bigger’ [-Wimplicit-
function-declaration]
     printf("%d\n", bigger(a,b));
                    ^~~~~~
```

As it compiled this from the top down, it reached the use of function bigger before it was declared and defined. We fix the warning by declaring it first:

```
#include <stdio.h>

int bigger(int n, int m); // declaration or prototype

int main(void) {
    int a=7;
    int b=9;
    printf("%d\n", bigger(a,b));
    return 0;
}

int bigger(int n, int m) // definition
```

```
{
    if (n>m) return n;
    else return m;
}
```

We often put function declarations in header files - see later.

A function declaration is alos called a *prototype*.

Recursive functions

This is a function which calls itself.

As an example - the factorial function is the product of all the integers down to 1. So factoral 3, written 3!, is 3X2X1=6, and 4!=4X3X2X1 = 24

```
#include <stdio.h>

int fact(int n)
{
    if (n==1) return 1;
    return n*fact(n-1);

}

int main(void) {

    for (int x=1; x<6; x++)
        printf("%d %d\n", x, fact(x));
    return 0;
}
```

In function fact, we check if n is 1. If so, the result is 1, and the function call ends.

If it is not, the result is n times the factorial of the next smaller number.

This is recursive because fact calls fact.

Recursive functiosn usually have a form like this - check a small simple case (n is 1) and directly return the answer. Otherwise return an expression calling the function on a smaller parameter.

We need to make sure the function will not call itself forever - an infinite recursion.

Test

The Fibonacci sequence is 1,1,2,3,5,8,13.. The first two are 1. After that, the next is the sum of the last two.

Write a recursive function to find the nth Fibonacci number

Pointers

A pointer is a value used to locate another value. It 'points' to something else.

In C, we use actual memory addresses as pointers.

& and *

In C, & means “the address of”

* is 'what is pointed at'. So

```
int main(void) {
    int x=6;   // simple int, value 6
    int * ptr; // ptr is a pointer to an int - a declaration
    ptr=&x;    // the value of ptr is the address of x
    int z=*ptr; // *ptr is what ptr points at - which is x
    printf("%d", z); // 6
    return 0;
}
```

Another example:

```
int main(void) {
    int x = 6;
    int * ptr;
    ptr = &x;

    printf("%p\n", ptr); // 0x7ffdd59f589c
    *ptr = 7;            // store 7 where ptr points to - which is where x is
    printf("%d", x);     // 7
    return 0;
}
```

%p is the format specifier for a pointer. So the address in RAM where x is held was 0x7ffdd59f589c (in hex).

We used the pointer to write 7 into memory - in the address where x is held. So, we changed x.

Pointer arithmetic

For example:

```
int main(void) {
    int x = 6;
    int y = 7;
    int z = 8;
    int * ptr;
    ptr = &x;

    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21d4 6
    ptr++;
    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21d8 7
    ptr++;
    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21dc 8

    return 0;
}
```

This code is dangerous and unreliable.

We have 3 ints in memory, with values 6 7 and 8.

```
    ptr = &x;
```

makes ptr point at the first, x, so 6

```
    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21d4 6
```

so *ptr is 6, as expected, and the actual address of that is 0x7ffda12a21d4

```
    ptr++;
```

Add 1 to ptr

```
    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21d8 7
```

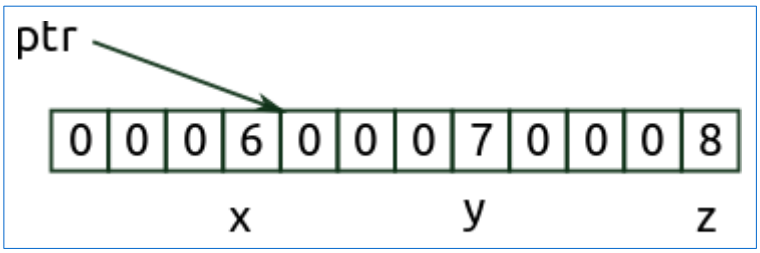
ptr has changed, and now points to 7 - which is y.

Add another 1 to ptr

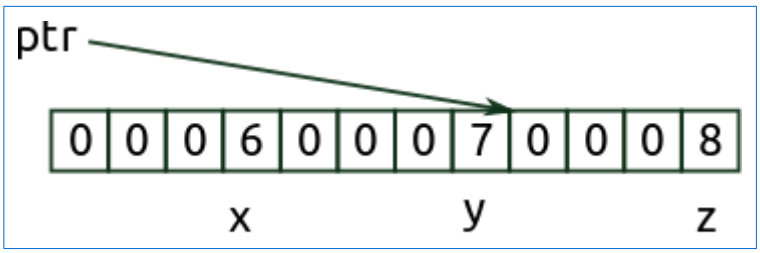
```
    ptr++;
    printf("%p %d\n", ptr, *ptr); // 0x7ffda12a21dc 8
```

which is z.

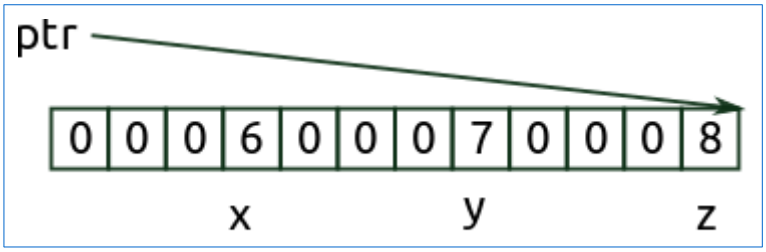
What happened? To start with we had in memory:



Then after the first ptr++



Then



So we get 6 7 and 8.

But in fact the values of ptr are 0x7ffda12a21d4, 0x7ffda12a21d8 and 0x7ffda12a21dc. These do *not* increase by 1 - they increase by 4. Why?

When we declared

```
    int * ptr;
```

we said ptr was a pointer to an int. So that

```
    ptr++;
```

makes ptr point to the *next int*, not the *next byte*. In fact on this system ints are 4 bytes long, so ptr++ adds 4 bytes to the address. This is key, and is why ptr is declared as a pointer to int.

Why is this code *unreliable*? It only works if the compiler puts x y and z next to each other in memory. It could have put them anywhere, in which case it would not have worked.

It is not clear from this what the purpose of pointers is. This next section shows this.

Test

Try this program out on your machine. You will get different addresses, and it may not work at all.

Dynamic memory allocation

Memory allocation means setting aside memory to store data in.

If we have

```
    int x = 6;
```

this allocates enough memory to store an int in, and does so. This happens at *compile-time*.

Dynamic memory allocation means allocating memory at *runtime* - and releasing it.

We need this if we do not know, when writing the program, how many data items we will need. For example in a graphics program we do not know how many shapes the user will want to draw. In an eCommerce application, we do not know how many items the customer will purchase. In a banking system, we do not know how many bank accounts we will have.

We do this by, at runtime, allocating more memory for the storage of data *as we need it*. We also need to *release* memory when it is no longer needed. The system tracks two areas in memory -

memory already in use, holding data, and

free memory, currently spare, available for use if needed.

The total amount of memory is fixed.

The contents of free memory no longer means anything, and is *garbage*.

malloc

This is a standard library function to request memory dynamically. It returns a pointer to the start of the memory obtained. For example:

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

int main(void) {
    int * blockStart;
    blockStart = (int *) malloc(4 * sizeof (int));  // get space for 4 ints
```

```
    *blockStart = 7; // store values in the space
    *(blockStart + 1) = 9;
    *(blockStart + 2) = 12;
    *(blockStart + 3) = 13;
    for (int offset = 0; offset < 4; offset++) // read them back
        printf("%d\n", *(blockStart + offset)); // 7 9 12 13
    free(blockStart); // release memory
    return 0;
}
```

We need the header file stdlib.h to get the declaration of malloc.

The parameter to malloc is how many bytes to get. Here we say 4 * sizeof (int) to get space for 4 ints. This will work, no matter how large an int is.

The return type of malloc is a void *. We use (int *) to typecast it to a type which matches blockStart.

Then we have lines like

```
    *(blockStart + 2) = 12;
```

to use pointer arithmetic to store values in the block, and a loop to read them back.

We use free to release the memory. In this case we are at the end of main, and it does not really matter. But if we repeatedly request memory, and never release it, we have a memory leak and will eventually run out of memory.

In fact malloc may fail if insufficient memory is available. What should happen then depends on the program logic. For example

```
#include <stdio.h>
#include <stdlib.h> // needed for exit

int main(void) {
    int * blockStart;
    blockStart = (int *) malloc(1000000000000 * sizeof (int)); // ask for a lot
    if (blockStart) // is 0 = false if fails
    { // we got the memory
    *blockStart = 7;
    *(blockStart + 1) = 9;
    *(blockStart + 2) = 12;
    *(blockStart + 3) = 13;
    for (int offset = 0; offset < 4; offset++)
        printf("%d\n", *(blockStart + offset)); // 7 9 12 13
    free(blockStart);
    }
    else
    { //no
        printf("Out of memory");
        exit(1); end with status 1
    }
    return 0;
}
```

When run we get

```
Out of memory
RUN FINISHED; exit value 1; real time: 0ms; user: 0ms; system: 0ms
```

calloc (calculated alloc) is very similar to malloc, except it does the multiplication for us. That is for example

calloc(5, sizeof(int));

is the same as

malloc(5*sizeof(int));

calloc also zeros the memory

Pointers and arrays

An array with an index corresponds to a pointer to a memory block with an offset into that block. But the two are not identical.

```
    // normal array use
    char array[]={'a', 'b', 'c'};
    for (int index=0; index<4; index++)
        printf("%c\n", array[index]); // a b c ?
    // normal pointer use
    char * ptr="abc";
    for (int index=0; index<4; index++)
        printf("%c\n", *(ptr+index)); // a b c (0)
```

The string constant ”abc” has a zero after it. This is not true for the char array {'a', 'b', 'c'};

We can use the array start as a pointer:

```
    *(array+1)='x';
    for (int index=0; index<4; index++)
        printf("%c\n", array[index]); // a x c ?
```

If we try to use a pointer as an array we get a segmentation fault

```
    //ptr[1]='x';
```

We can re-assign a pointer, but not an array:

```
   // re-assignment - not allowed for array
   //array={'x','y', 'z'};
   // ok for pointer
   ptr="xyz";
```

We can pass a pointer to a function argument declared as an array:

```
void show(int length, int numbers[])
{
    for (int index=0; index<length; index++)
        printf("%d\n", numbers[index]);
}
```

```
    ..
    int* data = (int*)malloc(3*sizeof(int));
    *data=3; *(data+1)=4; *(data+2)=5;
    show(3,data);
```

Parameter passing

When we pass a parameter to a function, can that function change the parameter? So that the change is seen in the code that calls the function? We find out:

```
#include <stdio.h>

void myFunc(int x)
{
    x=2;
    printf("In myFunc x = %d\n",x); // 2
}
```

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

    int x=1;
    myFunc(x);
    printf("In main x = %d\n",x); // 1

    return 0;
}
```

So the answer is no, even if the formal parameter and the actual parameter have the same name.

Why not?

Pass by value = pass a copy

In C (and many other languages) parameters are *passed by value*. That means a *copy* of the actual parameter is passed to the formal parameter. Then the called function can change the copy, but that will not change the original. This is very simple.

Pass by reference = pass the address

But suppose we want a function to change parameters? A function can have only one return value – suppose we want a function make several changes?

We can do this if the parameters passed are pointers. They will be *copies* of the original pointers. Code in the function can follow the pointers and change what is stored there. We cannot change the pointers. But we can change what is pointed to.

For example, suppose we want a function which exchanges 2 values:

```
#include <stdio.h>

void swap(int* x, int * y)
{
    int temp = *x;
    *x=*y;
    *y=temp;
}

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

    int x=1;
    int y=2;
    swap(&x, &y);
    printf("x = %d  y = %d\n",x,y); // 2 1

    return 0;
}
```

Test

Write and test a function

void show(char * start, int length)

This should output the contents of memory starting at start, byte by byte, for length locations. The output should be as chars (%c) decimal numbers (%d) and hex (%x).

Passing large data structures

Since a copy is passed, we might be concerned that passing a large data structure might be slow and use a lot memory, since we are making a copy.

Normally this is not true, because we do not pass a copy of the structure contents. Instead we pass a copy of a *pointer* to the start.

For example – total an array:

```
#include <stdio.h>

int sum(int size, int data[])
{
    int total=0;
    for (int i =0; i<size; i++)
        total+=data[i];
    return total;
}

int main(int argc, char *argv[]) {
    const int SIZE=100000;
    int bigArray[SIZE];
    for (int i=0; i<SIZE; i++)
        bigArray[i]=2;
    int result=sum(SIZE, bigArray);
    printf("%d\n", result); // 200000

    return 0;
}
```

In C an array does not 'know' how big it is, so we must say

```
int sum(int size, int data[])
```

to pass both the array and how long it is.

But int data[] is not a copy of all 100,000 ints. It is just a copy of the start address of the array.

Runtime data input – scanf

To input data values at runtime, the function scanf (scan formatted) can be used. For example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int x;
    printf("Enter a whole number :");
    scanf("%d", &x);
    printf("\nx is now %d\n", x);

    return 0;
}
```

We want scanf to alter the value of its parameter. So we must pass it the address of the parameter, here &x, so scanf can follow that pointer and change the value there.

Scanf is declared in <stdio.h>

A sample run:

```
Enter a whole number :28

x is now 28
```

In real code such a simple use is no good, since the user may not do what they are told, and we may get:

```
Enter a whole number :what?

x is now 32766
```

In real code we would

1. Input a value as a string
2. Check if the string was a valid int (or whatever)
3. Convert the string to an int if it is
4. Else output an error message and repeat input

Or we might input as command line parameters (see below) or from a text box in a GUI.

Test

Write a program which

inputs an integer

outputs whether it is even or not.

Lifetime

Scope is *where* an identifier can be used. Lifetime is *when*.

A global variable starts when execution starts, and ends when it ends.

A local variable usually starts when the enclosing function starts, and ends when the function ends. The local variable is created on the stack, and the memory used is released at the function return.

So this goes very wrong:

```
int * f()
{
    int d=4;
    return &d;
}
int main(int argc, char *argv[]) {

    int x = *f();
    printf("%d",x);
    return 0;
}
```

This gives a compile-time warning “warning: function returns address of local variable [-Wreturn-local-addr]” and if we run it, we get a segmentation fault. Variable d is created on the stack. We return the address of it. But at function end, the stack gets smaller, and that memory is now garbage.

We can do this:

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

int * f()
{
    int * space;
    space=(int *)malloc(sizeof(int));
    *space=4;
    return space;
}
int main(int argc, char *argv[]) {

    int x = *f();
    printf("%d",x);  // 4
    return 0;
}
```

Here space is a pointer. A copy of that pointer is returned by the function, then space becomes garbage. But the memory it points to still exists – with a 4 in it, and the calling code can fetch that.

Bad news is the memory leak. Function f mallocs space, and it is never freed.

This is better:

```
int main(int argc, char *argv[]) {
    int * where  = f();
    printf("%d",*where);
    free(where);
    return 0;
}
```

static

This is different for static local variables. These start when the enclosing function is first called, and continue until program execution ends (not just function). They *keep the value between calls*.

For example:

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

bool f()
{
    static int runCounter=0;
    runCounter++;
    printf("%d ", runCounter);
    if (runCounter==10) return false; else return true;
}
int main(int argc, char *argv[]) {

    while (f()); // 1 2 3 4 5 6 7 8 9 10

    return 0;
}
```

Test

1. Find, or write, or re-write, a program with a recursive function fib(int n) to find the nth. Fibonacci number.

2. Use a global variable to track how many times the function fib has been called.

3. Find out how many times it is called in finding fib(5)

4. Explain the answer

The standard library

Any C code contains:

- Reserved words (just 30 or 40 of them)
- Identifiers chosen by the programmer.
- Functions and a few other things defined by the programmer

As well as this, there are many things we often want to do which are not part of the language – an example being output to the console using printf.

These are *not* part of the language. Instead they are placed in the *standard library*. This is organised into standard units, and we #include the appropriate header file to provide access to what is needed.

The orginal ANSI C headers were:

<assert.h>	For testing
<ctype.h>	Functions to determine the type contained in character data
<errno.h>	Macros reporting error conditions
<float.h>	Limits of float types
<limits.h>	Sizes of basic types
<locale.h>	Localization utilities
<math.h>	Common mathematics functions
<setjmp.h>	Nonlocal jumps
<signal.h>	Signal handling
<stdarg.h>	Variable arguments
<stddef.h>	Common macro definitions
<stdio.h>	Input/output
<stdlib.h>	General utilities: memory management, program utilities,

	string conversions, random numbers
<string.h>	String handling
<time.h>	Time/date utilities

Check reference material for all details of all functions in all headers in all versions. This is just an overview.

assert

The purpose is for testing – checking things are as we think. We 'assert' (think, but want to check) something is true. If, at runtime, it is not, we get an error message on stderr and execution ends. For example:

```
#include <assert.h>

int main(int argc, char *argv[]) {
    int x = 1 / 3; // deliberate mistake - makes x 0
    assert(x * 3 == 1);
    int y = 4; // more code
    return 0;
}
```

produces at run-time:

```
testc: main.c:7: main: Assertion `x*3==1' failed.

RUN FINISHED; Aborted; real time: 10ms; user: 0ms; system: 0ms
```

ctype

Checks or changes characters – such as

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

int main(int argc, char *argv[]) {
    char c='A';
    if (isalnum(c)) printf("Yes\n"); // is alphanumeric
    if (isalpha(c)) printf("Yes\n"); // is alphabetic
    if (islower(c)) printf("Yes\n"); // is lower case (no)
    if (isupper(c)) printf("Yes\n"); // is upper case
    if (isdigit(c)) printf("Yes\n"); // is digit (no)
    if (isxdigit(c)) printf("Yes\n"); // is hex digit
    printf("%c\n", tolower(c)); // a
    return 0;
}
```

These apply to chars which are 1 byte wide – so not wide chars or Unicode.

errno

Some functions set the value of 'errno' if some kinds of error occur. For example if we try to find the square root of -1, this is a domain error – we can only find the square roots of positive numbers. Two set values are EDOM for domain errors and ERANGE for range errors. For example

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

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

```
    int x=sqrt(-1);
    printf("%d %d\n", errno, EDOM); // 33 33
    if (errno==EDOM) printf("Domain error\n"); // Domain error
    int d = atoi("99999999999999999999"); // convert string to int - but too big
    printf("%d %d\n", errno, ERANGE); // 34 34 - range error
    return 0;
}
```

float

There are 3 floating-point types – float, double and long double.

The ranges and precision of these depend on the implementation.

float.h contains these limits. For example

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

int main () {
   printf("The maximum value of float = %.10e\n", FLT_MAX);
   printf("The smallest positive double = %.10e\n", DBL_MIN);
}
```

limits

This contains the implementation-defined limits of integer types:

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

int main() {

   printf("The number of bits in a byte %d\n", CHAR_BIT);


   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);


   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);

   return 0;
}
```

CHAR_BIT is rather strange. In C, char is always stored in 1 byte. So, sizeof(char) is always 1. But – what is a byte? In C it is the storage used by 1 char (in real life it is 8 bits).

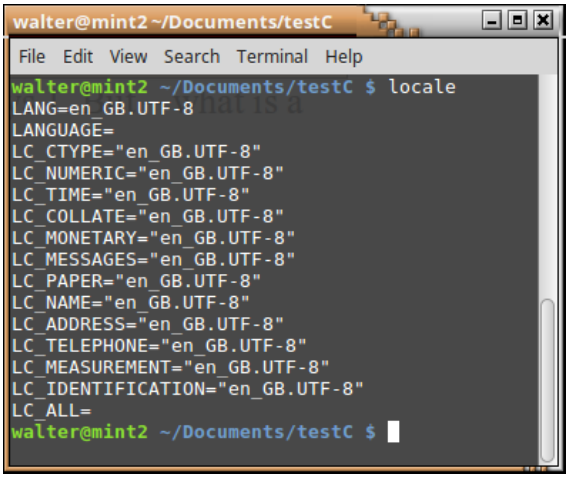
So 1 char occupies 1 byte by definition.

CHAR_BIT is very probably 8. The standards require it to be at least 8.

locale

The ideas behind this are

- Some things, such as currency symbol or date format, vary across the world



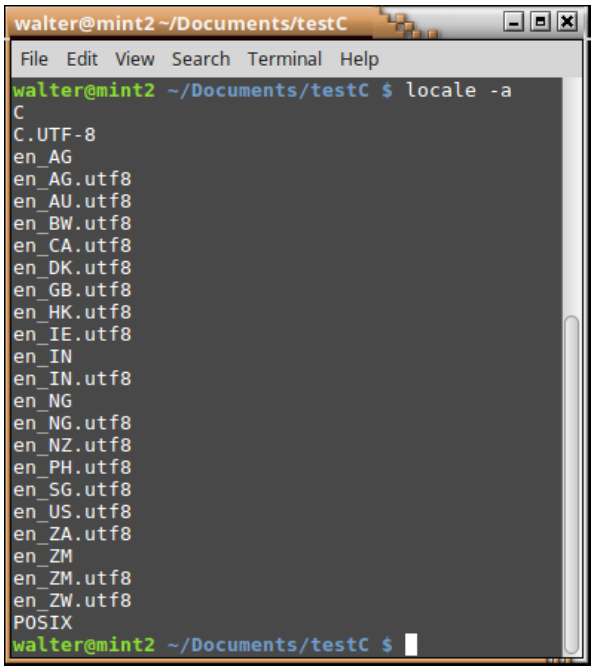
Linux - current locale

- It would be a good idea to be able to set these things as if the program was running in different places
- Or to just change one aspect (like currency alone)

What controls what the default locale is? What determines what different locales are available? The answer is the OS.

On Linux, the locale command tells you the current locale, and some information about it:

and locale -a shows you all available locales on that installation:



Linux - available locales

The locale.h header has a function setLocale to set the current locale, either one aspect or all. It also has a function localeconv to fetch the current locale settings. And it defines a struct, lconv, to hold locale information (see later about structs).

In gcc the default locale is "C", which is mostly nothing. We can set the locale in a C program to be the default from the OS by calling setLocale with "" as the argument:

```
    struct lconv * lc; // declare pointer to suitable
struct
    setlocale(LC_ALL, ""); // set locale to be system default
    lc = localeconv(); // fetch what we have
    printf("%s\n", lc->currency_symbol); // £
    printf("%s\n", lc->decimal_point); // .
    printf("%s\n", lc->thousands_sep); // ,
```

This is non-intuitive. There is

- The locale set in the OS – enGB or whatever
- The *default locale in a C program*, which starts as "C".
- We have to say setlocale(LC_ALL, ""); to change the l*ocale in the C program to the OS current locale*.

You do not normally have to do things to set a default.

We can then use another locale, if it is available:

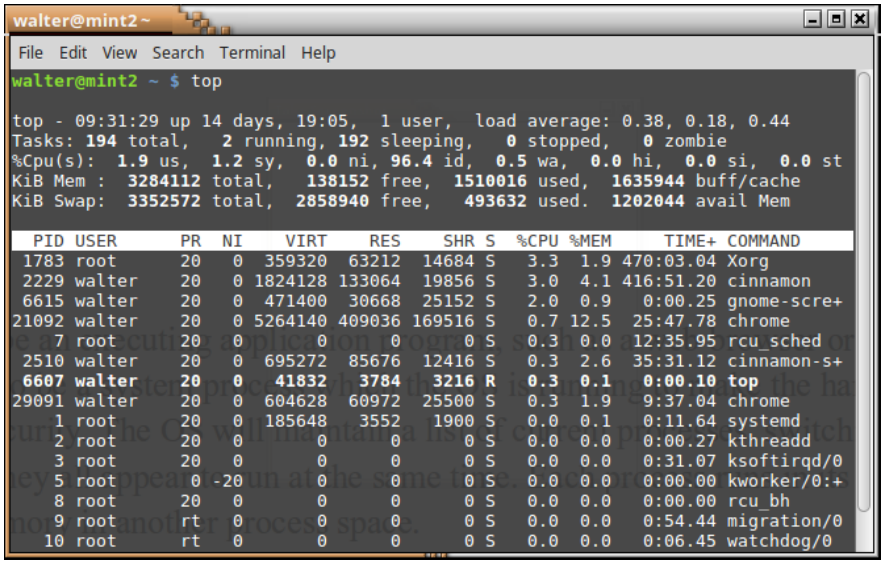
```
    printf("Locale is: %s\n", setlocale(LC_ALL, "en_IN.utf8")); // Locale is: en_IN.utf8
    lc = localeconv();
    printf("%s\n", lc→currency_symbol); // ₹
    printf("%s\n", lc→decimal_point); // .
    printf("%s\n", lc→thousands_sep); // ,
```

setjump

setjump is bad. Do not use setjump

signal

A *process* is an OS idea. A process might be an executing application program, such as a web browser or spreadsheet or word processor. It might also be a system process which the OS is running to make the hardware usable, or enable networking or enforce security. The OS will maintain a list of current processes, switching execution between them at high speed so they all appear to run at the same time. Each process runs in its own address space and cannot read or write memory in another process space.



top shows current processes

On Linux, the command top gives information on the current processes, as shown.

A *signal* is a message sent to a process – from the OS or another process. POSIX has a standard list of messages, each with an integer and symbolic name, so POSIX-compliant systems 'understand' these. That includes Linux and excludes Windows.

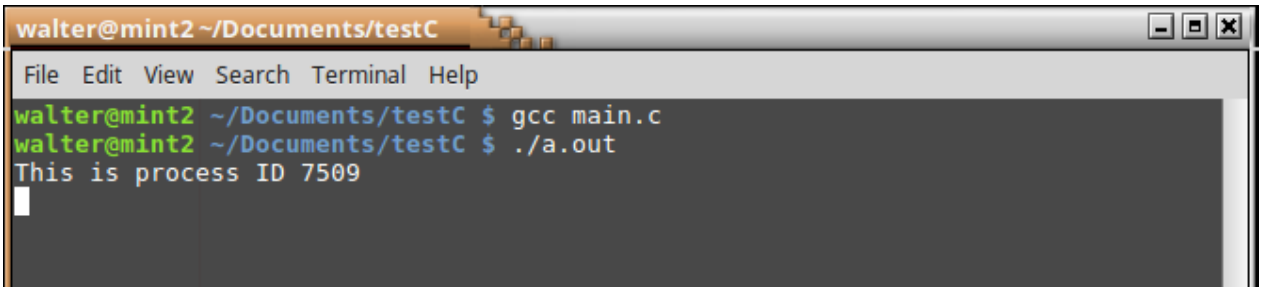
Each process has a pid – a process ID. In Linux we can use the kill command to send a signal to a process with some pid.

In signal.h two key functions are declared. The *raise* function sends a signal, and the *signal* function sets up a handler - in other words, says what function should be called when a signal is received.

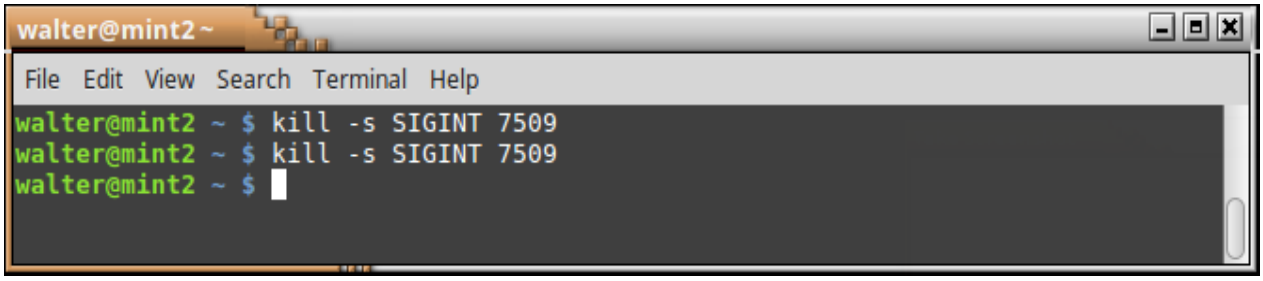
For example- a program in C:

```
#include <stdio.h>
#include <unistd.h> // for getpid
#include <stdlib.h>
#include <signal.h>

void handler(int signal) {
    static int count=0; // static remembers value
    puts("Interactive attention signal caught."); // confirm SIGINT received
```

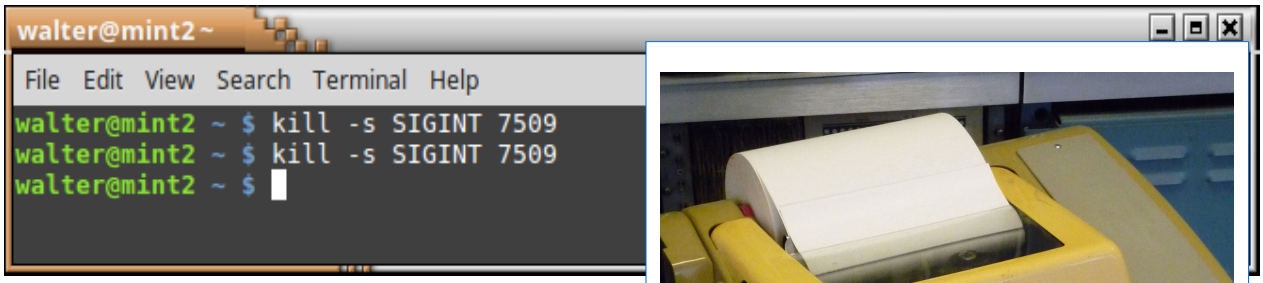
```
    count++;
    if (count==2) // on the second time..
        exit(1);  // end
}

int main() {
   // when we receive the SIGINT signal, execute handler function
   if (signal(SIGINT, handler) == SIG_ERR) {
        fputs("An error occurred while setting a signal handler.\n", stderr);
        return EXIT_FAILURE;
    }
    int p= getpid(); // output our pid
    printf("This is process ID %d\n", p);

    while (1); // go into endless loop

    return 0;
}
```



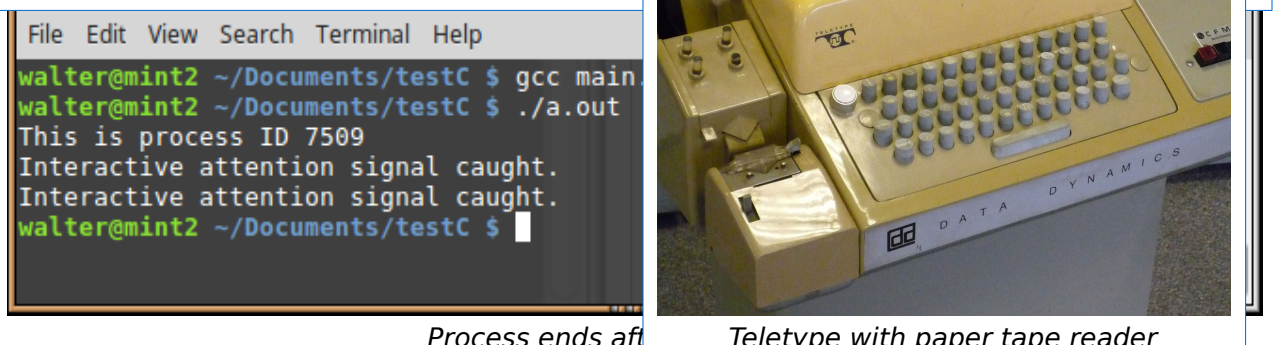
Compile and start program



Send SIGINT twice from another terminal



Send SIGINT twice fro



Process ends aft

Teletype with paper tape reader

The signal SIGHUP is like an India Jones piece of archaeology, showing the history of Unix. Around 1970 a typical hardware setup would have been a single processor (mini-computer) used by several users on Teletypes, connected through a serial interface to the processor, possibly through a modem. Users would start and stop processes on the shared processor through their terminals.

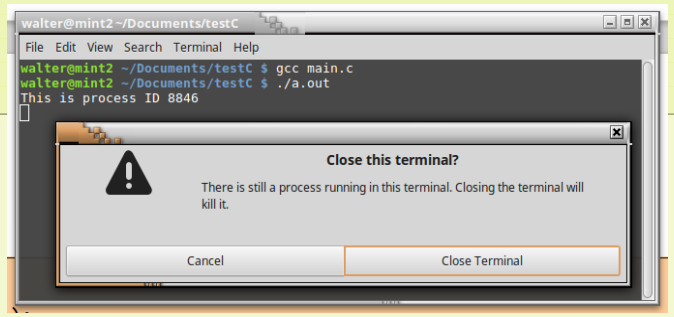
Now suppose the user's connection was lost – perhaps because they had hung up. What would happen? A SIGHUP – signal hung up – would be sent to the process.

Modern hardware would have the processor being a desktop or laptop and a single user. The equivalent of the Teletype is a 'terminal' and the equivalent of hanging up is closing the terminal window.

For example:

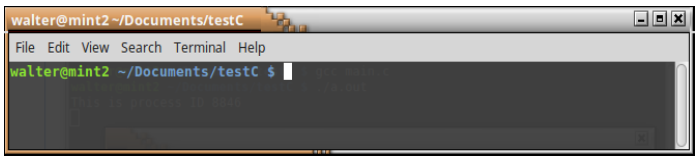
```
#include <stdio.h>
#include <unistd.h> // for getpid
#include <stdlib.h>
#include <signal.h>

void handler(int signal) {
    system("gnome-terminal"); // open a new terminal
}

int main() {
   // set up handler
   if (signal(SIGHUP, handler) == SIG_ERR) {
        fputs("An error occurred while setting a signal handler.\n", stderr);
        return EXIT_FAILURE;
    }
    int p= getpid(); // output our pid
    printf("This is process ID %d\n", p);

    while (1); // go into endless loop
}
```

Compile and start it, then close the window:



Close the window

and get a new terminal window:



New terminal

stdarg

This provides a way of passing a variable number of arguments to a function – sometimes called a *variadic* function. For example

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

int adder(int argCount, ...) {
    // add up arguments - all ints
    // argCount is how many we have
    va_list ap; // need a local of type va_list
    va_start(ap, argCount); // and call va_start on last named argument
    int sum=0;
    for (int count=0; count<argCount; count++)
    {
        sum+=va_arg(ap, int); // gets next arg each time
    }
    va_end(ap);
    return sum;
}

int main() {
    int total=adder(4, 1,2,3,4); // 10
    printf("%d\n", total);

    return 0;
}
```

There are two issues. One is how many arguments there are. In this example we pass that as the first argument. Another solution might be to use a sentinel value, say -1 as the last argument.

The other is type safety. The programmer must ensure the arguments passed are the type the function is expecting.

stddef

Contains a few macros, including NULL, the pointer to nowhere, which is probably zero. A very dull header.

stdlib

This is the 'standard library' and contains a set of signficant functions:

atof, atoi, atol	Convert a string to a double, int and long
strtod, strtol, strul	String to double, long, unsigned long
calloc, malloc, realloc, free	Request and release memory at runtime
abort	End program abnormally
atexit	Set function to call at exit
exit	End normally
getenv	Get environment variable

system	Pass command to system to execute
bsearch	Binary search
qsort	Quicksort
abs, labs, div, ldiv	Absolute value and divide
rand srand	Random number, seed generator
mbtowc and others	Multi-byte and wide char conversions

For example:

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

void sayBye()
{
    puts("Bye");
}

int main() {
    int i = atoi("23");
    double d=atof("34.56");
    atexit(sayBye);
    exit(1);

    return 0;
}
```

outputs

```
Bye

RUN FINISHED; exit value 1; real time: 0ms; user: 0ms; system: 0ms
```

calloc and so on are discussed elsewhere.

Binary search and quicksort are standard algorithms.

Environment variables

The OS maintains a set of environment variables, which are sets of name-value pairs. The names depend on the OS. We can list all available like this:

```
 int main(int argc, char **argv, char **envp) {
   int i = 1;
   char * s;
   for (s = *envp; s; i++) {
       printf("%s\n", s);
       s = *(envp + i);
   }

   return 0;
}
```

which outputs:

```
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
USER=walter
```

```
SNAP_COMMON=/var/snap/netbeans/common
SNAP_INSTANCE_KEY=
XDG_SEAT=seat0
SSH_AGENT_PID=2023
SHLVL=1
SNAP_LIBRARY_PATH=/var/lib/snapd/lib/gl:/var/lib/snapd/lib/gl32:/var/lib/snapd/v
oid
HOME=/home/walter
MDM_LANG=en_GB.UTF-8
..
```

We can use getenv to find one value:

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

int main(int argc, char **argv) {
    char * s = getenv("HOME");
    puts(s); // /home/walter
    return 0;
}
```

Random numbers

Devices with conventional hardware cannot produce true random numbers. They can only produce numbers in a sequence which 'looks mixed up', but which will eventually repeat. They are in fact *pseudo-random numbers*.

rand is a function in stdlib which produces pseudo-random ints, across the full range of possible int values.

If we take that % n, in other words the remainder when divided by n, it must be in the range 0 to n-1.

So rand()%10 + 1 would be in the range 1 to 10, inclusive, for example.

rand follows a sequence, and it would be the same sequence on every program run. srand is a function which 'seeds' that sequence – in other words re-starts it at some value.

It is common to call srand using a value which depends on the current time. This way we will get a different sequence on every program run:

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


int main(int argc, char *argv[]) {
    // initialize the random number generator depending on the current time
    time_t t;
    srand((unsigned) time(&t));
    for (int i=0; i<10; i++)
    {
    int number = rand() % 10 + 1; // random number 1 to 10
    printf("%d\n", number);
    }
    return 0;
}
```

math

math.h contains standard maths functions. For example

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


int main(int argc, char *argv[]) {
    // some constants
    printf("%lf\n", M_E); // e
    printf("%lf\n", M_PI); // pi
    printf("%lf\n", M_SQRT2); // root 2
    // exponential and log
    printf("%lf\n", exp(1)); // e
    printf("%lf\n", log(M_E)); // log
    // power
    printf("%lf\n", sqrt(4)); // square root
    printf("%lf\n", pow(3,2)); // power
    // trig
    printf("%lf\n", sin(M_PI/2.0)); // sine - uses radians
    // nearest int
    printf("%lf\n", floor(4.78)); // floor

    return 0;
}
```

Test

1. rand is a linear congruential generator. Google to find out what that is.

2. Write and test your own pseudo-random number generator.

Strings

A string is a sequence of characters.

A character is represented on a digital system by its *codepoint* - an integer, different for each character. Each *character set* has a different set of characters and codepoints. C implementations commonly (*but not always*) use the ASCII character set, which we use here. For the 'wide char' type, see later.

For example

```
 #include <stdio.h>

int main(void) {
    char * str = "ABC123";
    for (int offset=0; offset<8; offset++)
    {
        printf("%p %d %c\n",str+offset, *(str+offset), *(str+offset));
    }

    return 0;
}
```

Here "ABC123" is a string constant, with 6 characters, in “double” quotes. When the compiler compiles this, it constructs a string in the executable file, and when loaded, this produces a string constant in memory.

```
    char * str = "ABC123";
```

means str is a pointer to a character, and points to the start of the string.

```
        printf("%p %d %c\n",str+offset, *(str+offset), *(str+offset));
```

outputs data as the (%p) address, contents as a decimal integer (%d), and as a character (%c). The output is:

```
0x55de12956744 65 A
0x55de12956745 66 B
0x55de12956746 67 C
0x55de12956747 49 1
0x55de12956748 50 2
0x55de12956749 51 3
0x55de1295674a 0
0x55de1295674b 37 %
```

The address advance by just 1, because each character is stored in 1 byte. The contents go 65, 66 and 67 (ASCII codepoints for A B and C) then 49 50 and 51 (1,2,3). Then *we have a zero*. Then we have a 37, which is whatever comes next in memory - not part of our string.

The important thing is the zero - this is how strings are represented in C. They are *null-terminated strings* - in other words, they have a 0 at the end.

Why? So we can have code like this:

```
    char * str = "ABC123";
    char * ptr=str;
    char c;
    while (c=*ptr)
    {
        printf("%c",c); // ABC123
        ptr++;
    }
```

This

```
while (c=*ptr)
```

gets the character pointed to by ptr. But at the end of the string, this is 0 - and this counts as false, so the loop ends.

This is often written as

```
    char * str = "ABC123";
    char * ptr=str;
    char c;
    while (c=*ptr++)
    {
        printf("%c",c); // ABC123
    }
```

using post-increment - to get the most efficient machine code.

String processing

Suppose, for example, we want to count the words in a sentence. We assume each word is separated by one space:

```
    char * str = "This is a test.";
    char * ptr = str;
    char c;
    int wordCount = 1;
    while (c = *ptr++) {
        if (c == ' ') // is c a space character?
            wordCount++;
    }
    printf("%d", wordCount); // 4
```

The string standard library

If we say

```
#include <string.h>
```

we have a choice of 22 functions relating to strings. For example:

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


int main(void) {
    char * space=(char *)malloc(7*sizeof(char)); // get 7 bytes
    memset(space, 0, 7); // set them to zero
    strcpy(space,"cat"); // copy "cat" into that space
    printf("%s\n", space); //cat
    strcat(space, "dog"); // concatenate "dog" to that string
    printf("%s\n", space); // catdog
    printf("%s\n", strchr(space,'t')); // tdog find the first 't
    printf("%ld\n", strlen(space)); // 6 length of string

    return 0;
}
```

These need to be used with care, to avoid a *buffer over-run*. For example:

```
    char * space=(char *)malloc(7*sizeof(char)); // get 7 bytes
    memset(space, 0, 7); // set them to zero
    strcpy(space,"12345678901234567890123456789012345678901234567890"); // copy 40 chars
    printf("%s\n", space); // 12345678901234567890123456789012345678
```

Here space is enough room for 7 bytes. The memory after those 7 bytes will probably be used by something else. The strcpy has copied 40 bytes there. The first 7 are OK, but after that we are writing into memory used for other data, so it goes wrong.

So there are safer versions, which limit the amount copied:

```
    char * space=(char *)malloc(7*sizeof(char)); // get 7 bytes
    memset(space, 0, 7); // set them to zero
    strncpy(space,"12345678901234567890123456789012345678901234567890", 6); // copy max 6 chars
    printf("%s\n", space); // 123456
```

Why 6, when we have 7 bytes? Because we need one for the final 0.

See documentation for details of all the string functions.

DMA techniques

Some languages, such as Java, manage memory for us – and so restricts what the programmer can do directly.

C enables us to manipulate memory directly, which gives great power. However this must be done very carefully, maintaining a clear idea of how memory will be processed.

As an example, suppose we want a function to do trim(). This removes leading and trailing spaces from a string. So that “ 123 “ is changed to “123”. How to do that?

One approach would be:

1. Find the first non-space character

2. Find the last non-space character
3. Work out how many characters there are between these
4. Get that amount of new space, plus 1
5. Copy those characters into the new space
6. Put a zero at the end
7. Return the new space.

Like this:

```
char *  trim(char * string)
{
    char * start=string;
    char * end = string+(strlen(string)-1); // point to last char
    // move start forward over spaces until reach non-space, or
    // hit end (string is all spaces)
    while (*start==' ' && start!=end) start++;
    // move end back to first non-space
    while (*end==' ' && end!=start) end--;
    // reserve enough memory for this, including final 0
    char * newSpace=(char *)malloc(end-start+2);
    // copy non-space chars to new space
    strncpy(newSpace, start, end-start+1);
    // put terminating 0
    *(newSpace+(end-start+2))=0;
   return newSpace;
}
```

Test

Write and test a function

char * reverse(char * string)

which returns a pointer to a string which is the reverse of the string passed to it.

Level Two

Operators and expressions

Ternary operator

This is different in that it has two symbols, ? and :, and 3 parts, as <first> ? <second> : <third>. The idea in effect is:

- Is <first> true?
- If so the answer is <second>
- If not, <third>

For example:

```
    int x = 5;
    int y;
    y = (x == 5) ? 2 : 3; //makes y 2
    x = 11;
    y = (x > 12) ? 5 : 6; // makes y 6
```

The last is equivalent to

```
if (x>12)
 y=5;
else
y=6;
```

but is probably more efficient machine code.

The bitwise operators

Logical OR is ||. This takes two operands, treated as true or false, and produces true if and only if both are true - taking false as 0 and everything else as true. For example

```
    int x=2;
    int y=3;
    int z = x||y;
    printf("%d\n", z);  // 1 = true
```

The bitwise operators treat the operands as bit patterns, and operate on corresponding bits. For example:

```
    int x=8;
    int y=5;
    int z = x|y;
    printf("%d\n", z);  // 13
```

Why? Because 8 is 1000 in binary, and 5 is 0101. If we OR those, we get 1101, which is 13. In fact these have 32 bits not 4, but this saves space.

| is bitwise OR, & is bitwise AND, ~ is NOT.

The bit shift operators move the bits in an operand left or right. For example:

```
    int x=8;
    int z = x>>2;
    printf("%d\n", z);  // 2
```

8 is 1000. 8>>1 is 0100 or 4. 8>>2 is 0010 or 2

See elsewhere (like here) for more examples of bitwise processes.

= as an operator

In C, an expression followed by a semi-colon is a valid statement. For example

```
int main(int argc, char *argv[]) {
    2 + 3;
    return 0;
}
```

The expression 2+3 is evaluated, and the result discarded.

= is an operator, like + and *. Its value is the left hand side. So

```
x=2+3
```

evaluates 2+3, and this is the value. It has the *side-effect* of making x to be 5.

So we can say

```
    y=x=2+3;
```

1. this works out 5
2. assigns to x - giving the result 5
3. assigns 5 to y

Comma separated lists

We can have a list of expressions, separated by commas, as a statement. These are evaluated left to right:

```
    x=4,y=2,x++;
    printf("%d",x); //5
```

Test

1. Why does this output 2?

```
#include <stdio.h>


int main(int argc, char** argv) {
    const int LEAST4BITS = 0xF;
    int x=258;
    printf("%x\n", x & LEAST4BITS);  // 2
    return (0);
}
```

2. Explain the output of this:

```
#include <stdio.h>

int main(int argc, char** argv) {
    unsigned int MASK = 0xFF000000;
```

```
    unsigned int x=0x12345678;
    printf("%x\n", (x & MASK) >> 24);
    MASK=MASK>>8;
    printf("%x\n", (x & MASK) >> 16);
    MASK=MASK>>8;
    printf("%x\n", (x & MASK) >> 8);
    MASK=MASK>>8;
    printf("%x\n", (x & MASK));
    return (0);
```

3. Write a function void showBits(int n) which outputs an integer n in binary.

Command line arguments and exit status

Command line arguments

One way to provide runtime input to a program is to supply values as command line arguments.

For example

./add 1 2

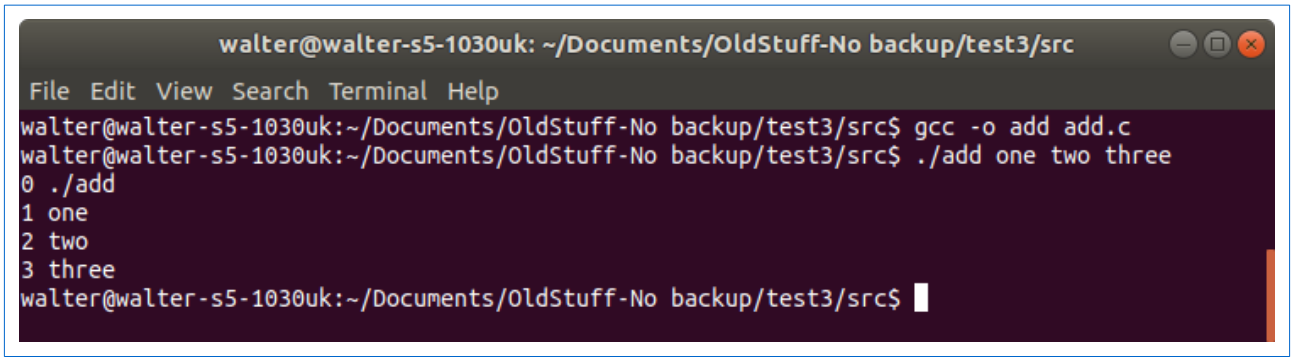
would run the program add, with input data 1 and 2, and we would expect output 3.

To do this, main can receive two parameters. These are an array of strings, which are the input values, and a count of how many values there are.

For example:

```
int main(int argc, char *argv[])
{
    for (int i=0; i<argc; i++)
    printf("%d %s\n",i,argv[i]);

    return 0;
}
```

Compiling and running this:



Exit status

It is traditional for a program to return a small integer showing if execution was successful, or some error code if there was a problem. The standard for success was 0, and so main usually ends

```
return 0;
```

and we can return something else if there was a problem.

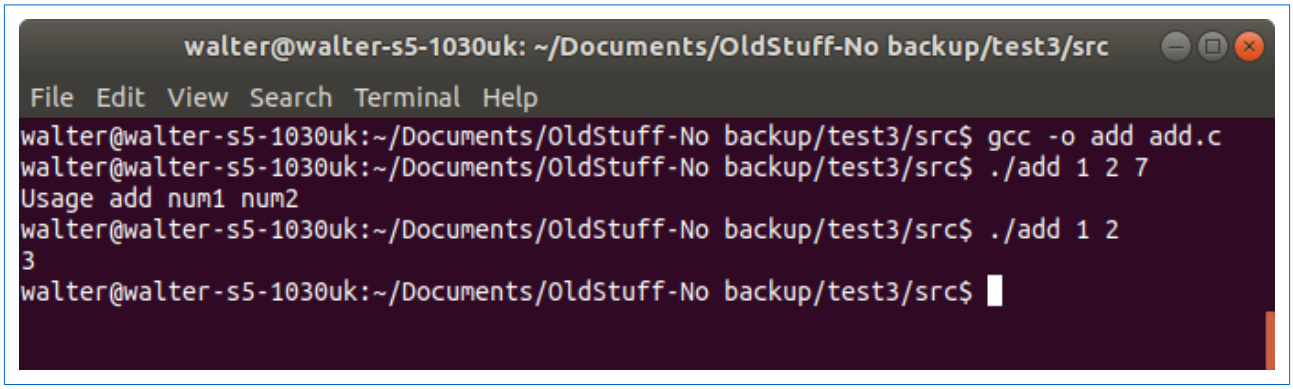
The exit status can be accessed by the OS, and used in a *batch script*. But this depends on the OS *shell*, not C.

Command line example

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

int main(int argc, char *argv[])
{
    if (argc!=3) // should be 3 elements in argv
    {
        printf("Usage add num1 num2\n");
        return 1; // error code
    }
    int num1=atoi(argv[1]); // convert string to int
    int num2=atoi(argv[2]);
    int sum=num1+num2;
    printf("%d\n",sum);

    return 0; // exit success
}
```

The string array is often named argv, for argument values, and argc is the argument count. The first argument is the program name itself.

Compiling and running this:



IDEs usually have a feature to provide command line arguments for testing.

Test

Write a program calc which takes 3 command line arguments – two ints and an operand, + - or *. So that for example

`calc 3 * 4`
outputs 12, while

`calc 3 – 4`
outputs -1.

The pre-processor

The pre-processor is software which processes source code text, before it is compiled. It can only process the text - it does not produce binary code.

Instructions to the pre-processor are called *pre-processor directives*, and they start with a #. So in fact

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

is a pre-processor directive.

#define

This sets up a *macro*. For example

```
#define PI 3.1416
```

This means wherever you find PI in source code replace it by 3.1416. for example

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

#define PI 3.1416

int main(void) {

    printf("%f", sin(PI/2)); // 1.0000

    return 0;
}
```

This makes code a lot more readable (makes more sense) and saves typing.

Check - no final semi-colon, so not

```
#define PI 3.1416;
```

This type of macro is useful for what are in effect constants:

```
#define BUFFER_SIZE 80
..
char * buffer= (char *)malloc(BUFFER_SIZE);
```

This is readable, and we can easily alter the buffer size.

Constants are usually UPPER CASE to make them recognisable.

Function-like macros

A macro can take an argument, and this means they can mimic a function. For example:

```
 #include <stdio.h>

#define square(X) X*X

int main(void) {
    int x=square(3);
    printf("%d",x); //9
    return 0;
}
```

because when the macro is expanded it becomes

```
int x=3*3;
```

But these can go wrong - for example:

```
int main(void) {
    int x=square(3+1);
    printf("%d",x); // 7
    return 0;
}
```

We would have expected x to be 16, but we get 7. This is because the expanded macro said:

```
    int x=3+1*3+1;
```

A second problem is when we compare it with function calls. Every time we have a macro, the corresponding code is repeated. But for a function we just repeat the call, not the function code.

Avoid function-like macros if possible.

Test

Write a macro print(X) which outputs the int X.

Conditional compilation

As well as using #define to give a value to a word, we can just use it as a flag. Then we can do something like:

```
#include <stdio.h>

#define VERBOSE

int main(void) {
#ifdef VERBOSE
    printf("Starting..\n");
#endif
    int x=3;
    printf("%d\n",x); // 3
   #ifdef VERBOSE
    printf("Ending..\n");
#endif

    return 0;
}
```

The

```
#ifdef VERBOSE
```

tells the pre-processor that if the VERBOSE macro is defined, keep this section in - upto the #endif. Or miss it out if its not. This is *conditional compilation*. This means we can have a single source code version, and switch parts of code in or out with a single change (like commenting out #define VERBOSE)

We will see the standard use of conditional compilation when discussing header files.

#include

This does a textual include of source code at this point. In other words:

```
#include myfile.c
```

in effect does this:

1. gets myfile.c
2. selects and copies it
3. pastes it, in place of the #include

Why? Usually for header files, explained later.

Test

Google and look at The International Obfuscated C Code Contest

Type

Pointer to function type

If we define a function f, then f(); calls the function. But f by itself is the start address of the function - a pointer to a function.

For example

```
#include <stdio.h>

int inner() // define a function
{
    return 99;
}

int main(int argc, char *argv[]) {
    int (*func99)(void) = inner;       // func99 is a pointer to a function, returning an int, taking no
                                       // arguments. We assign inner to it

    int x=func99();                    // call inner through a pointer to it
    printf("%d",x);
    return 0;
}
```

Why not call inner directly? We could have a function which takes a function pointer as argument, and calls it. Then what that function does depends on what argument we pass to it:

```
#include <stdio.h>

int inner()
{
    return 99;;
}
int another()
{
    return 100;
}

void exec(int (*funcPtr)(void)) // we pass a function pointer to this
{
    printf("%d", funcPtr()); // and call it
}

int main(int argc, char *argv[]) {
    int (*func99)(void) = inner;
    int (*func100)(void) = another;
```

```
    exec(func99);
    exec(func100);
    return 0;
}
```

Then function exec serves as a despatcher, executing whatever function we pass to it.

Unsigned

How are ints represented in C? In other words, how are values stored as binary patterns, byte by byte?

The C standards do not say. In other words, different C representations may do it in different ways.

For ints, there are common methods - sign and magnitude, one's complement and two's complement. Sign and magnitude is the simplest so we explain that.

Values are stored as 1 bit for the sign (usually the leftmost bit, 0 for + and 1 for -) and the rest as the base two value. So in 8 bits, 1000 0011 would be -3, while +3 would be 0000 0011. Another example - 0000 1100 is +12, and 1000 1100 is -12

Then the biggest we can have is 0111 1111 = +127, and the smallest is 1111 1111 = -127.

But suppose the value we are representing cannot be negative. For example it might be the number of cars queuing in a traffic flow simulation. We cannot have a negative number of cars, so half the range is wasted.

Instead we could just have no sign bit, and use 1111 1111 to be 255, down to 0000 0000 for 0. Storing it as an unsigned value gives us a bigger maximum value, in the same memory.

So unsigned can be added to many types, giving us examples like signed short, unsigned short, signed int, unsigned int, and so on.

The biggest whole number type is a long long.

Each type has a minimum size, but the actual size depends on the implementation, and we must use sizeof if we need to know.

Boolean type

To start with (C89 ANSI C), C treated 0 as false, and any other value as true.

C99 did the following:

- Added a new type, _Bool
- In header stdbool.h, added a macro to define bool as _Bool
- Added macros for true as 1 and false as 0

so you could say the more readable:

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

int main(int argc, char *argv[]) {
    bool x;
    x=true;
    if (x)
        printf("x is true\n");
    bool y;
    y=3==4; // so y=false;
    if (!y)
        printf("3 and 4 are different");
    return 0;
}
```

Wide characters

What do we mean?

A character set is a set of characters, each with a unique codepoint, which is an integer. Common ones are ASCII and Unicode

A character-encoding is a way of representing those codepoints in binary. Common ones are UTF-8 and UTF-16. We outline UTF-8

UTF-8

works using 1, 2, 3 or 4 bytes per character. If the codepoint is less than 128, it just goes in the single byte. The leading bit is 0, so the system knows there are no further points.

In the range hex 0080 to 07FF it goes into 2 bytes. The first byte starts 110 and the second 11 so the system knows it is a 2 byte character.

In the range 8000 to FFFF it goes into 3 bytes, starting 1110, 10 and 10

10000 to 10FFFF goes in 4 bytes, starting 11110, 10 10 and 10.

This therefore can handle Unicode codepoints. But it does not mean that characters in UTF-8 must be Unicode. They might use ASCII – in which case they are all 1 byte characters.

Recall:

- The char type stores characters as codepoint integers in 1 byte
- Can use any character set, but most commonly, ASCII

We now need to handle a large number of different scripts worldwide, not just A to Z, which means we need far more characters than the 256 which will fit into 1 byte.

The current response to this seems muddled. There is

<wchar.h> with the wchar_t wide character type

char16_t for chars 16 bits wide

char32_t for chars 32 bits wide

<uchar.h> for Unicode support

The width of wchar_t is implementation-defined – it might be just 1 byte. It therefore may not handle Unicode.

The 16 bit and 32 bit have implicatiosn for what the encoding would be (eg 16 bit cannot handle UTF8 4 byte codepoints)

and uchar.h assumes a character set, going against C's philosophy of not assuming a character set.

We explore the wchar type:

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

int main(int argc, char *argv[]) {
    fwide(stdout, 1);
    wchar_t c = L'δ';
    wprintf(L"Size of c = %ld\n", sizeof (c)); // Size of c = 4
    wprintf(L"c=%x %c \n", c, c); // c=3b4

    return 0;
}
```

The complier standard needs to be after C89, or it will not understand wchar.h

Wide char character and string constants need to start L. so

```
    wchar_t c = L'δ';
```

declares c to be a wchar character, and assigns it to the Greek letter delta, to be stored as a wide character.

You cannot use printf to output wide characters – use wprintf instead.

You cannot mix printf and wprintf. This is because stdout needs to be set to expect a stream of single byte, or wide, characters to write. So

```
    fwide(stdout, 1);
```

tells stdout to expect wide chars. In fact the first wprintf does this anyway. Once set this cannot be changed (by a further call to fwide for example). So you cannot mix printf and wprintf.

```
    wprintf(L"Size of c = %ld\n", sizeof (c)); // Size of c = 4
```

so in this implementation a wchar is 4 bytes wide – it may not always be.

```
wprintf(L"c=%x %c \n", c, c); // c=3b4
```

If we print out c as a hex integer we get 3b4. This is the codepoint of a δ in Unicode.

How precisely are strings stored?

```
    wchar_t * str = L"abγδ";
    wprintf(L"%ls\n", str);
    char * ptr = (char *) str;
    int byteCount = 0;
    while (byteCount < 20) {
        wprintf(L"%d %p %hhx \n", byteCount, ptr, *ptr);
        ptr++;
        byteCount++;
    }
```

outputs:

```
ab??
0 0x400778 61
1 0x400779 0
2 0x40077a 0
3 0x40077b 0
4 0x40077c 62
5 0x40077d 0
6 0x40077e 0
7 0x40077f 0
8 0x400780 b3
9 0x400781 3
10 0x400782 0
11 0x400783 0
12 0x400784 b4
13 0x400785 3
14 0x400786 0
15 0x400787 0
16 0x400788 0
17 0x400789 0
18 0x40078a 0
19 0x40078b 0
```

We get ab?? because this goes to a console which cannot display Unicode.

Each character is in 4 bytes, so ‘a’ is

```
0 0x400778 61
1 0x400779 0
2 0x40077a 0
3 0x40077b 0
```

The γ is

```
8 0x400780 b3
9 0x400781 3
10 0x400782 0
11 0x400783 0
```

because 03b3 is the Unicode codepoint.

At the end we have

```
16 0x400788 0
17 0x400789 0
18 0x40078a 0
19 0x40078b 0
```

which is the final terminating zero – but in 4 bytes, not just one.

Compilation in more detail

We said before that the compiler translates source code to machine code. In fact it is not as simple as that. A project will usually have several .c files, and use standard library functions (and maybe other libraries). The following happens:

1. Run the preprocessor and alter source code text appropriately
2. Translate each source code file to an intermediate form - assembly language
3. Link the assembly language, together with library functions
4. Produce an executable file

We have looked at the preprocessor. We study the next three steps:

Intermediate assembly language

Assembly language is similar to machine code, except that it is textual not binary coded. Assembly language uses instruction like mov for move data, and add for add. We will actually see movl meaning move long - move four bytes. And addl add long, four bytes. It will refer to memory locations, and registers, which are very fast working storage inside the processor (not in memory). Each assembly language instruction matches one machine code instruction, so it is easy to turn assembler into machine code.

We can get gcc to stop work at this stage by the -S option. In other words

gcc -S main.c

will output assembly language code into the file main.s, and stop. If main.c is:

```
#include <stdio.h>


int main(void) {
    int x=3;
    int y=4;
    int z=x+y;
    printf("%d",z);

    return 0;
}
```

then main.s is:

```
        .file   "main.c"
        .text
        .section        .rodata
.LC0:
        .string "%d"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0: // program code starts here
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
```

```
        .cfi_def_cfa_register 6
        subq    $16, %rsp // up to here, setup
        movl    $3, -12(%rbp) // put 3 in memory (where x is)
        movl    $4, -8(%rbp)  // put 4 in memory (where y is)
        movl    -12(%rbp), %edx // get the 3 back into the dx register
        movl    -8(%rbp), %eax  // get the 4 into the ax register
        addl    %edx, %eax      //  add dx into ax, result in ax
        movl    %eax, -4(%rbp)  // store ax into memory (where z is)
        movl    -4(%rbp), %eax  // get z into ax
        movl    %eax, %esi      // then into register si
        leaq    .LC0(%rip), %rdi // prepare for function call
        movl    $0, %eax
        call    printf@PLT      // call printf
        movl    $0, %eax      // return status code 0
        leave                   // done
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

We have added comments to outline what is happening. Register names start with a %. rbp is a register containing the address of where data is being held in memory. It is using -12(%rbp) to hold x, -8(%rbp) for y, and -4(%rbp) for z.

The code stores 3 and 4 in memory, then fetches them into registers, adds them and stores the result back to memory (in z). Then it sets up to call printf, then ends.

The compiler may *optimize* the code produced – that is, change it to make it use less memory or go faster. As an example, for timing reasons we may have a 'do-nothing' loop to produce a delay:

```
for (int i=0;  i<1000000; i++);
```

The compiler may work out this has no net effect, and so simply remove the code. This is useless for timing purposes.

Test

1. Look at some assembler from C code and try to make sense of it. Keep it very simple.

2. Read this

Linking

The assembler has instructions like

```
        call    printf@PLT
```

The call part will just turn into the machine code equivalent. But the printf@PLT is a *symbolic address*, which needs to be replaced by the actual *numeric address* of where the printf function starts - in the library file stdio.

This is what the linker does - works out the actual address of where things are, and repaces the symbols with the numeric addresses. You sometimes see a linker error, when it cannot find some symbol.

Executable file production

An executable file is not simply pure machine code. The OS will load the executable file into memory, and it has to be in a format which it understands. For example it needs to be told how much memory to reserve for data for the process, and what initial data values to set up in it. For example if the source code says

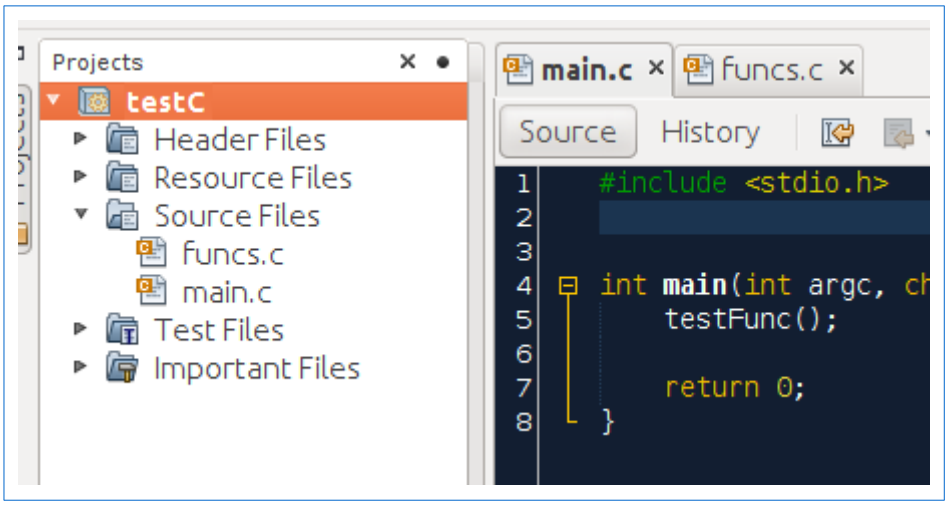
```
printf("Hello world");
```

then the "Hello world" will be setup up as a string constant in that memory.

The format of the executable depends on the OS. For example on Windows it is .exe file format. On Unix type systems it is ELF - Executable Linkable File.

Multi-file projects and headers

We usually have projects which contain several .c code files, not just 1. Like this, in NetBeans:



Why?

1. So that files are not too long and complex
2. To organise code into logical units which are related
3. To allow teams to work on separate files
4. To re-use code in other projects.

Standards call files *compilation units*. In a build, the *make utility* will only compile those files that have altered since the last compile (then link them).

Accessing code across files

In our example we have main in file main.c, and put other functions in funcs.c.

Suppose we want to call a function coded in funcs.c from main.c.

func.c is:

```
#include <stdio.h>


void testFunc()
{
    printf("In testFunc\n");
}
```

and main.c is

```
int main(int argc, char *argv[]) {
    testFunc();
    return 0;
}
```

What happens? When we compile, we get a warning:

```
main.c:4:5: warning: implicit declaration of function ‘testFunc’ [-Wimplicit-
function-declaration]
      testFunc();
      ^~~~~~~~
```

This is because in main.c the compiler has not seen a declaration (prototype) of testFunc.

How to fix it?

One solution is to put the declaration in main.c

```
void testFunc(void);

int main(int argc, char *argv[]) {
    testFunc();
    return 0;
}
```

This works, but if we have a lot of source code files with many functions it would become unmanageable. If we altered code in one file we might need to alter declarations in all other files which use it. Better to use a header file.

Header files

We solve the problem by:

1. Adding a new file to the project - a header file funcs.h
2. In funcs.h we write function declarations
3. In main.c we #include funcs.h

So funcs.h is

```
// This is funcs.h

void testFunc(void);
```

and main.c is

```
#include "funcs.h"

int main(int argc, char *argv[]) {
    testFunc();
    return 0;
}
```

To include standard library files we use angle brackets, like <stdio.h>. For our own headers we use quotes, like “funcs.h”

Sharing and hiding data

Suppose we want to have data from one file accessible in another?

1. Declare it to be *extern* in the header
2. Define it in one file
3. Use it in the other

For example, in the header:

```
// This is func.h

extern int funcX;

void testFunc(void);
```

In func.c:

```
#include <stdio.h>

int funcX=9;

void testFunc()
{

    printf("In testFunc\n");
}
```

In main.c:

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

int main(int argc, char *argv[]) {
    testFunc();
    printf("%d", funcX); // 9
    return 0;
}
```

Or, maybe we want it to be only usable in one file, and not elsewhere? This is done by declaring it to be *static*:

```
#include <stdio.h>

static int funcX=9;

void testFunc()
{
    printf("In testFunc\n");
}
```

In this case funcX in func.c is local to that file. If the variable is also defined in another file, it is unrelated.

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

int funcX=10;

int main(int argc, char *argv[]) {
    testFunc();
    printf("%d", funcX); // 10
    return 0;
}
```

(static has two meanings. For global data, it means limit scope to the one file. For local variables, it means the value is kept between function calls.)

Header guards

We might well use code from one file in several others. Each one might include the corresponding header file. But this might mean the same header is compiled several times. If the header defines a variable or function, this will be an error. The fix is to use a conditional compilation header guard like this:

```
// This is func.h

#ifndef FUNC_H  // this is the first time - if not defined

void testFunc(void);

#define FUNC_H // won't do it again


#endif
```

The first time, the macro FUNC_H is not defined, so this is compiled, including the #define FUNC_H.

But that means if the compiler meets this again, the second time the macro is defined, and the code is ignored.

Declarations, definitions, initialisations and assignments

What do these mean, in C?

A *declaration* tells the compiler the type of an identifier

A *definition* allocates storage for an identifier

An *initialisation* gives an initial value in a new storage location

An *assignment* changes the value in a storage location

```
int c=9;
```

probably does three of these. It declares c to be an int, it allocates storage for it, and it initialises it to 9. If we had missed out the =9, it would have been initialised to 0.

```
    extern double d;
```

declares d to be a double type – but does not define it (no storage allocated) So

```
#include <stdio.h>

    extern double d;

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

    printf("%lf",d);
    return 0;
}
```

will not compile – we get “undefined reference to `d'”

This is also OK:

```
#include <stdio.h>

    extern double d;
    double d=8.8;


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

    printf("%lf",d);
    return 0;
}
```

but we would not usually do this, because if d is in this file, the extern declaration is pointless.

We would probably say:

```
#include <stdio.h>

    extern double d;

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

    printf("%lf",d);
    return 0;
}
```

and in another file

```
double d=3.1;
```

We can even declare it twice:

```
#include <stdio.h>

    extern double d;
    double d; // OK

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

    printf("%lf",d);
    return 0;
}
```

but we cannot initialise it twice:

```
#include <stdio.h>

    extern double d;
    double d=6.6; // no


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

    printf("%lf",d);
    return 0;
}
```

which makes sense.

We can declare it twice:

```
#include <stdio.h>

    extern double d;
    extern double d;
int main(int argc, char *argv[]) {

    printf("%lf",d);
    return 0;
```

or define it twice

```
#include <stdio.h>

double d;
double d;

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

    printf("%lf",d);
    return 0;
}
```

but not to different values.

And we cannot declare and define it locally twice

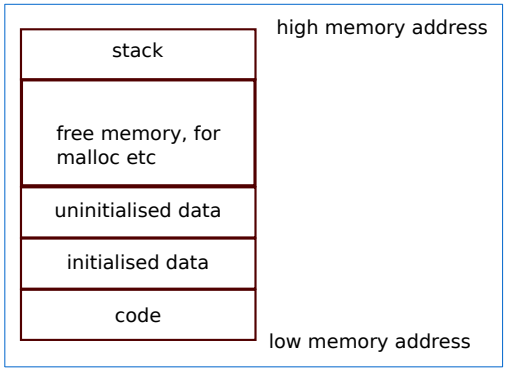
```
#include <stdio.h>

int main(int argc, char *argv[]) {
    double d;
    double d; // no
    printf("%lf", d);
    return 0;
}
```

Memory layout of a C program

How is memory laid out as a C program executes?

Trick question. C programs do not execute – they are simply source code. What executes is native code – machine code – after compilation.

There are two issues – how an executable *file* is laid out, and how is *memory* laid out when the file is loaded into memory and execution starts.



Neither have anything to do with C. Both are controlled by the OS. On Windows, exe files have a standard format and layout once in memory. On Linux, executable files have an ELF format and layout in memory.

Memory will be something like this, after the executable file has been loaded.

The stack holds data local to a function. If we have

```
void fun()
```

```
{
int x=9;
..
}
```

then when fun starts, a stack frame is created on the stack, holding the 9 for x. When the function ends, this is taken off the stack. The stack grows down into memory.

Free memory is available for use by malloc and so on during execution.

People sometimes talk about the *heap* and the *stack*. Both terms have two meanings, so this is confusing.

A stack is a classical data structure – a last in first out list. So is a heap – a type of tree where each node is the largest in the sub-tree of which it is the root.

But in terms of memory usage, 'heap' means just a bunch of data, used globally by the program. For compiled C, this is global data. The stack is used for functions – for parameter passing and return adddresses, and local variables with lifetime limited.

This diagram cannot be taken literally – because

- It has nothing to do with C
- The details depend on the OS
- This is just for one process. The OS runs many *processes* at the same time – application programs, system software like the file system, the OS itself. Each process runs in its own memory space. The diagram just represents the memory space of one process.
- Most OS use *virtual memory* – processes can be swapped in and out of physical RAM onto a disc swap file if memory is low. At any one time, a process might be swapped out.
- When swapped back in, a process may be at a different place in physical RAM than it used to be – it is *relocatable code*.

Make files

Suppose we have a 'project' containing several .c files and several header files. We need to

1. Compile each .c file
2. Link the object code files to produce an executable.

If we alter a .c file we need to re-compile it.

If we alter a header file, we need to re-compile any .c files that include it.

So we need to track what we do. But we want to avoid re-compiling code which we have already compiled and not changed.

This is what a *make utility* is for. We give it a *makefile*, saying what is in our project and what depends on what. Then the make utility tracks the datestamps on files to decide which need to be re-compiled.

If we use an IDE, this seems to happen by magic. In fact the IDE uses some make utility, together with an editor and compiler and linker and the rest of its tool chain.

This is not part of C. Each make utility has its own rules. These are for the GCC make. Details are here.

For example suppose we have main.c, utils.c and utils.h:

main.c is

```
#include <stdio.h>

#include "utils.h"

int z = 9;

int main(int argc, char *argv[]) {
    add(3);
    printf("%d\n", z);

    return 0;
}
```

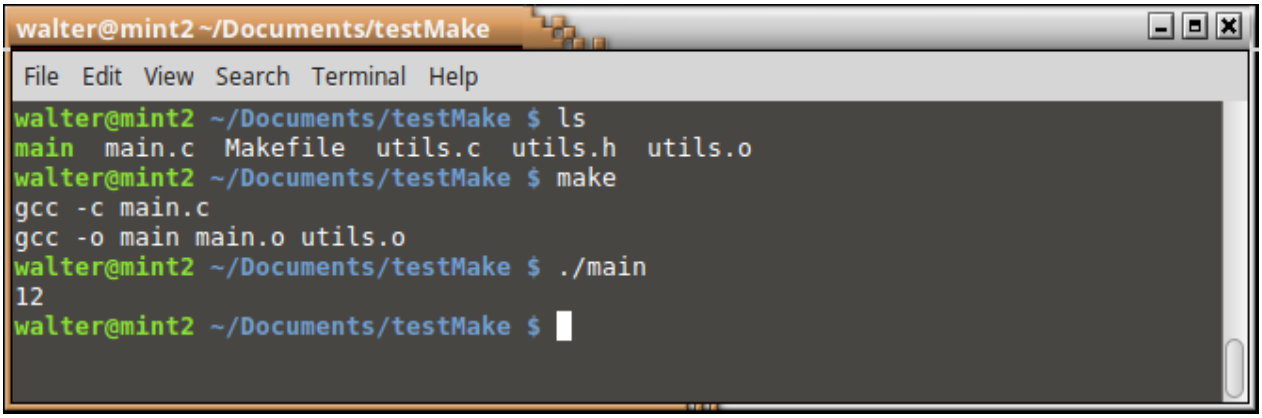
```
// utils.c

#include "utils.h"

extern int z;

void add(int x)
{
    z+=x;
}
```

```
// utils.h

void add(int);
```

The make file (in a file named Makefile) is:

```
# makefile

main : main.o utils.o
	gcc -o main main.o utils.o

main.o : main.c utils.h
	gcc -c main.c

utils.o : utils.c utils.h
	gcc -c utils.c
```

We can use it by:



Using make

The make file is a series of rules. So

```
main : main.o utils.o
	gcc -o main main.o utils.o
```

means that main depends on main.o utils.o (so if either changes, we need to re-compile it, and 'gcc -o main main.o utils.o' is how to make it.

Before gcc is *one tab* – not spaces.

Standards

A language is a set of rules

The rules are about

syntax, or grammar - exactly how you say things, and

semantics, meaning, what the code will do.

Who decides the rules?

Some languages are *proprietary*, which means they are invented and owned by a company - such as Apple owning Swift. The rules for proprietary languages are chosen, and can be altered, by their owner.

Languages such as C have *standards* to set out the rules. The standards are agreed by panels of experts in standards committees in organisations like the ISO.

Standards documents use English in a very odd way. For example C99 says:

> *In this International Standard, ''shall'' is to be interpreted as a requirement on an implementation or on a program; conversely, ''shall not'' is to be interpreted as a prohibition.*

This is useful if you understand what ‘requirement’ or ‘prohibition’ mean, but not ‘shall’ or ‘shall not’. Otherwise its not much use.

The history of C standards is outlined here:

Usual name	Formal name	Notes	Features
K&R C		By Kernighan and Ritchie in ‘The C Programming Language’ first edition 1978	
ANSI C = C89	ANSI X3.159-1989 "Programming Language C"	First committee standard. As in K&R second edition	Function prototypes Void pointers International character sets
ISO C = C90	ISO/IEC 9899:1990	Same as C89, but approved by ISO not ANSI. C95 amended version	
C99	ISO/IEC 9899:1999		Inline functions long long complex variable length arrays // one-line comments no prototype not int by default
C11	ISO/IEC 9899:2011		Multi-threading Unicode gets removed Generic macros Alignment
C18	ISO/IEC 9899:2018		Corrections to C11 No new features

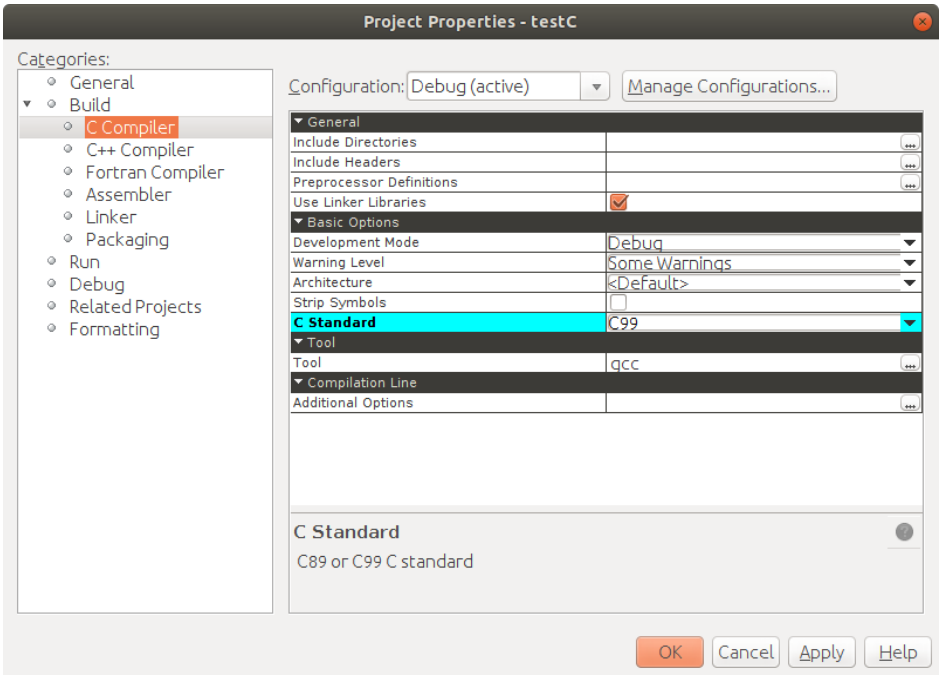
The earliest standard for C was called C89, because it was published in 1989. That was also known as ANSI C. The latest version is C18.

The official versions of the standards cost money, but various drafts and revisions are on the web. C18 is here.

What is the purpose of a standard? What is it for?

For compiler-writers. It sets out for them what the compiler should accept, and reject, and what the code it produces should do. Without a standard, a C program might be compiled by one compiler but rejected by another.

For programmers. So they know what rules their code needs to obey, and what it will do – no matter which compiler it compiles on, or what platform it executes on.



Compiler using C99 standard

Many compilers do not currently meet C11 or C18.

It is important that the code you write matches *some* standard. This is mostly under the control of the compiler. A good compiler will have command line options to control which standard it uses. For example

```
-std=c99 -pedantic
```

means it will use the C99 standard – and therefore throw warnings or errors for code which does not meet the standard.

In Netbeans we can go to the project, properties, build, C compiler, and see this

Here we are using the C99 standard.

Note we are also using a Warning Level of Some Warnings. A warning means something which may be wrong. Take note of warnings - if you get one, fix it.

Cross platform C

A *platform* is the combination of processor hardware and OS.

Ideally we want to be able to write just one version of a C program, which will work correctly on any platform.

Because we compile to machine code, which is different for different processors, the source code needs to be compiled for different processors

How many bytes is used by an int? We can find out:

```
int main(void) {
    printf("%lu\n", sizeof(int));  // 4
    return 0;
}
```

We can use *sizeof* to find out how many bytes a type uses.

So why not use 4 in code? Why is sizeof better?

Because the size of these types is chosen for highest efficiency on a given platform. This was on a 64bit system, where a long is 64 bits and an int is 32 bits. That means the data bus is 64 bits wide, and this is how much data we can move in one step.

But on other platforms an int may not be 4.

If we say sizeof, not 4, and re-compile the code for another platform - then the same code works on any platform.

Files

The OS provides functions to deal with files - create them, read and write them, delete them. They do this through *file handles* or *file channels*. When a file is opened, a file handle is created for it, input and output works with reference to that handle, and when the file is closed, the handle deleted.

The details depend on the OS.

From C, we use functions to call those OS functions. But we always need the sandwich:

1. open file
2. use it
3. close file

The idea of the file handle in C is called a file pointer.

Basic file functions

The standard file functions (in stdio.h) are

fopen	open a file. Different attributes for create new file, read-only file, append file (write data at the end) and so on
fscanf or fgetc	read data from file

fprintf or fputs	write to a file
fclose	close a file

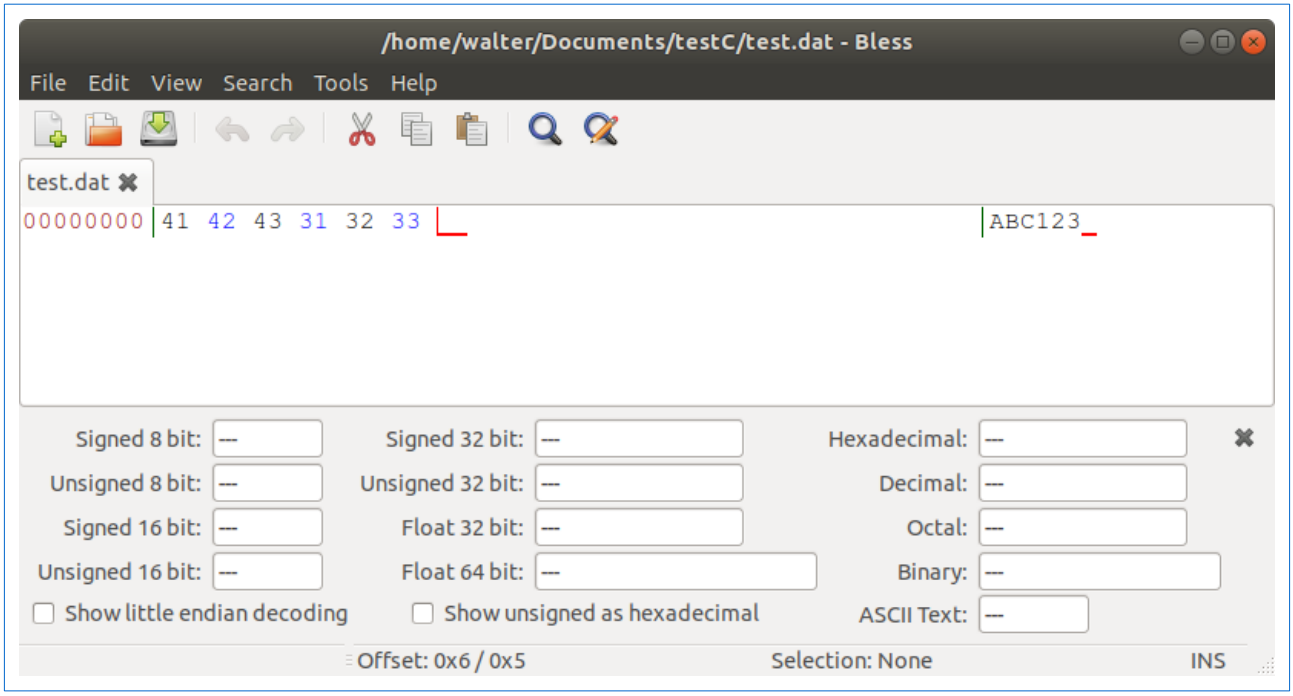
These treat files as if they were on tape, with *serial access* only - we start at the start and work forwards.

Two other functions fseek and rewind - let us do *random access*, to move anywhere within the file.

Write a text file

```
#include <stdio.h>

int main(void) {
    FILE * fptr; // declare file pointer
    fptr=fopen("test.dat", "w");       // open file. w means write to file, delete if already
                                       // exists
    fputs("ABC123", fptr);             // output string to file
    fclose(fptr);                      // close file

    return 0;
}
```

If we look at the contents of test.dat in a hex editor, we see:



so it contains six bytes, hex 41 42 43 31 32 33, ASCII codes of ABC123

Read text file

We know our file test.dat is six bytes long. But we code it as if we did not know how long it would be:

```
#include <stdio.h>

int main(void) {
```

```
    FILE * fptr;
    fptr=fopen("test.dat", "r"); // open file to read
    char c;
    while (1) // loop indefinitely
    {
        c=fgetc(fptr); // get a character
        if (feof(fptr)) break; // end of file, end loop
        printf("%c",c); // ABC123
    }
    fclose(fptr);

    return 0;
}
```

Writing a binary file

A binary file contains data which is not text. Image files are an example.

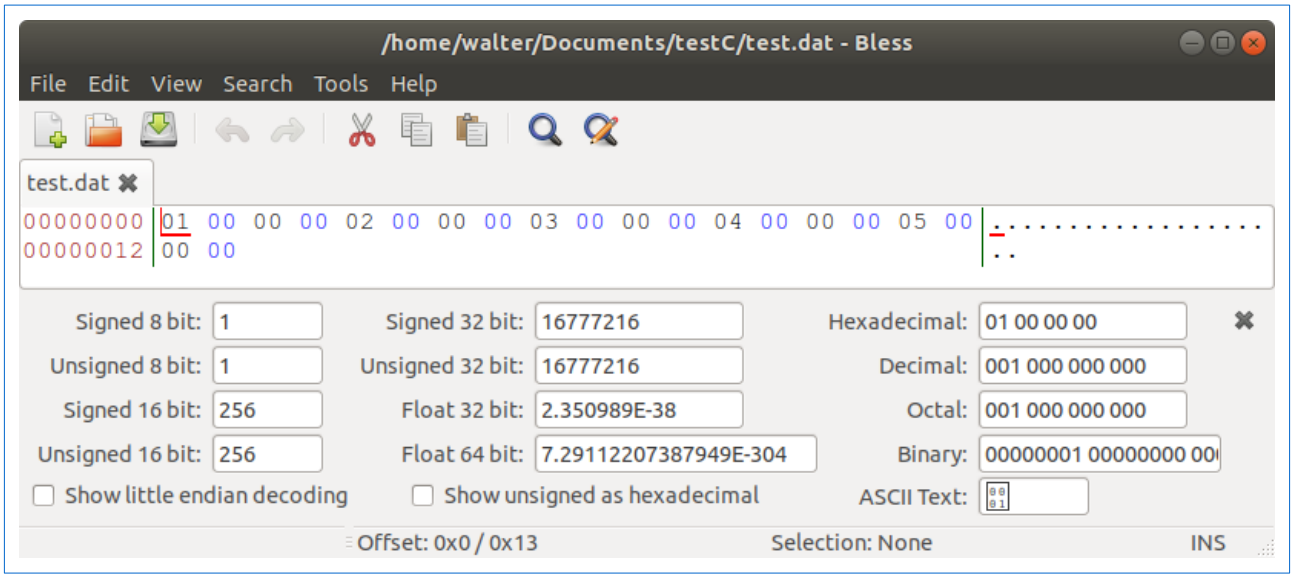
```
#include <stdio.h>

int main(void) {
    FILE * fptr;
    fptr=fopen("test.dat", "wb");  // wb = write binary
    int data[] = {1,2,3,4,5};
    fwrite(data, sizeof(int), sizeof(data)/sizeof(int), fptr );
    fclose(fptr);

    return 0;
}
```

This writes 5 integers into the file. The arguments to fwrite are:

- pointer to the data,
- size of each element to write, in bytes
- number of elements
- file pointer to write to.

We know there are 5 elements, but we say sizeof(data)/sizeof(int) so it will work for any size.

If we look at test.dat, we see:



This is 20 bytes long, or 5 times 4 bytes for each int. This is *not* ASCII codes. The first four bytes are 1 0 0 0, which is how 1 is represented in binary as an int (least significant byte first).

Read a binary file

A binary file is just a stream of bytes, so there is no way to tell by looking at them (for sure) how they are formatted, unless we know

- the file format (eg gif, jpeg, tiff or whatever), and
- how such a file is formatted

In our example, we know the format is a sequence of ints. We must use that knowledge to read it back:

```
#include <stdio.h>

int main(void) {
    FILE * fptr;
    fptr = fopen("test.dat", "rb"); // read binary
    int buffer; // a space to hold each int read
    while (1) { // forever
        fread(&buffer, sizeof (int), 1, fptr); // read an int
        if (feof(fptr)) break; / stop on end of file
        printf("%d", buffer); // 12345
    }
    fclose(fptr);

    return 0;
}
```

The arguments to fread are

- the address of a space to read data into
- the size of each data item
- how many data items to read
- file pointer to read from

This code does not assume how many ints are in the file. We read them one at a time until we get end of file.

Efficiency of file I/O and buffer size

A file read or write is usually pretty slow, because the underlying hardware (such as a disc file or a network connection) is slower than memory and processor. It is usually faster to have a larger buffer and fewer reads. In our last example it would have been faster to read 5 ints in a single operation, not 1 int in a loop 5 times. Either would have been very fast, but if the file had been megabytes long that would not have been true.

However such considerations are complicated by what the OS does (because the OS is what actually does the I/O). So file writes, for example, may go into a buffer of the OS. In other words when we execute

```
fwrite(data, sizeof(int), sizeof(data)/sizeof(int), fptr );
```

the OS may not actually carry out the file write. Instead it may write it to a buffer, then write it out to disc (or wherever) when it has time or when it has to, like something else reading the file.

It may also maintain that buffer as a cache. So

```
        fread(&buffer, sizeof (int), 1, fptr);
```

may actually mean the OS reads data from the cache which is faster than a disc read.

This makes the process unpredictable, and if timing is critical, the best approach is to code it, then actually measure how long it takes, like this:

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

int main(void) {
    clock_t start, end;
    start = clock();
    FILE * fptr;
    fptr = fopen("test.dat", "wb");
    int data = 1; // time writing a million ints
    for (int count = 0; count < 1000000; count++) {
        fwrite(&data, sizeof (int), 1, fptr);
    }
    fclose(fptr);
    end = clock();
    printf("Time taken = %ld clock ticks", (end - start));
    return 0;
}
```

Abstract treatment of files

Since Unix, files have been treated as abstracts. This means that how you read a file on magnetic disc, or on a solid state drive, or on a network, or on a CD, will be different. But the difference is in the device drivers. As far as the OS, and C, is concerned, these are all files.

The idea of a file is abstracted. We can read, write, create and delete files, and we do not care how it is done. We use the abstract idea of a file - what it can *do*, not *how*.

stdin, stdout and stderr

There are 3 special file channels, stdin, stdout and stderr. These are files are opened by the system at startup. stdin connects usually to the keyboard for input, stdout usually to the command line. stderr is where error messages go - probably also to the console. We say probably because there may be no keyboard, or console.

As a result there are pairs of functions:

printf goes to stdout

fprintf is the same, but goes to a file

scanf comes from the keyboard (stdin)

fscanf comes from a file

Structs and typedefs

typedef

A typedef lets you set up your own name for a type. Like:

```
typedef int wholeNumber ;

int main(void) {
    wholeNumber x,y,z;
    x=3;
    return 0;
}
```

Then you can write wholeNumber in place of int.

This is little use, other than to confuse someone looking at your code, who will ask 'what is this wholeNumber?'

See later

struct

We often want to have data values which have parts. For example in a banking application we might want to deal with bank cheques. A cheque has a number, an amount, an account number and other parts. How can we represent that as an int or double? We cannot - we need a struct:

```
struct BankCheque
{
    int number;
    int account;
    double amount;

};

int main(void)
{
    struct BankCheque cheq1;
    cheq1.number=23;
    cheq1.account=243566;
    cheq1.amount=9.55;

    return 0;
}
```

So now a BankCheque is in effect a new type. It has 3 fields - a number, an account and an amount. We can declare a variable of this type by

```
    struct BankCheque cheq1;
```

and refer to a field as

```
    cheq1.number=23;
```

We can write functions handling the struct:

```
#include <stdio.h>
struct BankCheque
{
    int number;
    int account;
    double amount;

};

void display(struct BankCheque cq)
{
    printf("Bank cheque: Number 5d\n", cq.number);
    printf("Account: %d Amount %f\n",cq.account, cq.amount);
}

int main(void)
{
    struct BankCheque cheq1;
    cheq1.number=23;
    cheq1.account=243566;
    cheq1.amount=9.55;
    display(cheq1);
    return 0;
}
```

and so on.

Typedef struct

The only problem is that we have to keep on typing struct, like

```
    struct BankCheque cheq1;
```

The fix is to have our struct, then typedef it to something short and meaningful. Like this:

```
#include <stdio.h>
typedef struct BankCheque
{
    int number;
    int account;
    double amount;

} Cheque;

void display(Cheque cq)
{
    printf("Bank cheque: Number %d\n", cq.number);
    printf("Account: %d Amount %f\n",cq.account, cq.amount);
}

int main(void)
{
    Cheque cheq1;
    cheq1.number=23;
    cheq1.account=243566;
    cheq1.amount=9.55;
    display(cheq1);
    return 0;
}
```

→ notation

Suppose we have a pointer to a struct. How can we access a field of the struct? If the pointer is ptr, then *ptr is the struct, and (*ptr).f is the field. But ptr→f is a better way to say it. For example

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

```
#include <stdlib.h>

typedef struct S
{
    int f1;
    int f2;
} MyStruct;

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

    MyStruct * ptr;
    ptr=(MyStruct *)malloc(sizeof(MyStruct));  // make a struct
    ptr→f1=27; // write 27 to field f1
    printf("%d", ptr->f1); //27

    return 0;
}
```

Typecast struct

It is usually impossible to typecast a struct to another type, either implicitly of explicitly – because it makes no sense. For example

```
typedef struct {
    int x;
    int y;
} Pair;

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

    Pair pair;
    pair.x = 3;
    pair.y = 5;
    long d = (long) pair;

    return 0;
}
```

We get a syntax error:

```
error: aggregate value used where an integer was expected
```

We have given no instructions as to how the two ints in a Pair should be converted to a long.

We might have said

```
    Pair pair;
    pair.x = 3;
    pair.y = 5;
    long d = ((long)pair.x)<<32 | pair.y;
    printf("%lx",d); // 300000005
```

What is happening here?

(long)pair.x

explicitly casts pair.x, a 4 byte int, to a 64bit long. Then

```
((long)pair.x)<<32
```

shifts the bits 32 places right, then

```
((long)pair.x)<<32 | pair.y;
```

ORs those bits with the 32 bits of pair.y.

So we have put together the 2 lots of 32 bits from a Pair into a single 64 bit long, with the x field in the top 32 bits and the y field in the lower 32 bits. In hex we see

```
300000005
```

Bitfields

A bit field is a set of fields with each one packed into the smallest possible number of bits it might need (as decided by the programmer).

For example in a chess game we might have :

```
typedef struct {
    int row;
    int column;
} Square;
```

as a type for a chess board square. With 2 ints in it, that might take up 8 bytes. But we only have 8 rows or columns, so the only possible values are 1 to 8, and we can fit that into 4 bits (0001 to 1000). So we could do that as a bitfield:

```
typedef struct {
    char row:4;
    char column:4;
} Square;
```

We have switched to char from int, and said we only need 4 bits for each. Then we can say something like:

```
    Square s1;
    s1.column=2;
    s1.row=3;
```

If we say:

```
    s1.row=17;
```

that would not fit into 4 bits, and we might get a warning:

```
warning: overflow in implicit constant conversion [-Woverflow]
```

How big is a Square?

```
    printf("%d\n", sizeof(Square)); // 1
```

compared with 8 bytes for a 2 int struct.

The exact way a bitfield is set out will depend on the implementation. It will at least be packed out with spaces to make it a number of bytes, probably a power of 2:

```
 typedef struct {
    char row:4;
    char column:4;
    char color:1; // so at least 9 bits
} Square;
..
printf("%d\n", sizeof(Square)); // 2
```

As another example – on Linux a file has a set of ‘permissions’. These are whether the file can be read, written to, or executed, either by the owner of the file, the group of the owner, or by others. We can represent that as a bitfield:

```
#include <stdbool.h>

typedef struct {
    bool OwnerRead : 1;
    bool OwnerWrite : 1;
    bool OwnerEx : 1;
    bool GroupRead : 1;
    bool GroupWrite : 1;
    bool GroupEx : 1;
    bool OtherRead : 1;
    bool OtherWrite : 1;
    bool OtherEx : 1;

} PermStruct;
```

then say

```
    PermStruct someFilePerm;
    someFilePerm.GroupEx=true;
    someFilePerm.GroupWrite=false;
    printf("%ld\n", sizeof(PermStruct)); // 2
```

Enums

An enumeration is a set of named integer constant values.

For example, for our file permissions, we could have a code: 0 means none, 1 means read, 2 means write, 3 means read and write, 4 means read and execute, and so on. But we might struggle to remember the code scheme. Instead we should have an enum, and use it in a bitfield struct:

```
enum Perm { N, R, W, RW, RX, WX, RWX };

typedef struct
{
    enum Perm owner:3;
    enum Perm group:3;
    enum Perm other:3;
} PS;
```

then say

```
    PS p2;
    p2.group=RW;
    p2.owner=RWX;
```

In fact N, R and so on are just integer codes:

```
    p2.owner = RWX;
    printf("%d\n", p2.owner); // 6
    p2.owner = 1;
    printf("%d\n", p2.owner); // 1
    p2.owner = 9; // warning: large integer implicitly truncated to unsigned type [-Woverflow]
    printf("%d\n", p2.owner); // 1 : 1001 truncated to 0001
```

but enums are a lot more readable.

Union

A struct is a set of fields stored one after the other. A union is a set of fields stored ‘in the same place’. In other words the bytes in a union can be used to hold different types at different times. The idea is we share memory for different purposes.

For example:

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

```
// define the MyUnion union type..
union MyUnion {
    char c;     // can hold a char
    int i;      // or an int
    double d;   // or a double - in same place
};

int main(int argc, char** argv) {

    union MyUnion u1; // make a union
    u1.c = 'a';             // store a char there
    printf("%c\n", u1.c);
    u1.i = 4;               // or an int
    printf("%d\n", u1.i);
    u1.d = 3.5;             // or a double
    printf("%lf\n", u1.d);
    union MyUnion * uPtr;   // pointer to a union
    // make one
    uPtr = (union MyUnion *)malloc (sizeof(union MyUnion));
    uPtr->c='b';            // use arrow notation
    printf("%c\n", uPtr->c);

    return (EXIT_SUCCESS);
}
```

As for a struct, typing union MyUnion every time is tedious, and we can typedef it. We can also look at what is actually held in memory:

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

// define the type..
typedef union MyUnion {
    char c;     // can hold a char
    int i;      // or an int
    double d;   // or a double - in same place
} UnionType;

int main(int argc, char** argv) {

    UnionType u1;           // make a union
    u1.c = 'a';             // store a char there

    // what is actually in memory now?
    void * ptr = &u1;
    for (int offset=0; offset<8; offset++)
    {
        printf("%d %x\n", offset, *(char *)(ptr+offset));
    }
    u1.i=0x1011;
    for (int offset=0; offset<8; offset++)
    {
        printf("%d %x\n", offset, *(char *)(ptr+offset));
    }
    return (EXIT_SUCCESS);
}
```

The first loop outputs:

```
0 61  // codepoint of ‘A’ in hex
1 4   // garbage
2 40
3 0
4 0
5 0
6 0
7 0
```

The second loop outputs:

```
0 11 // the int hex value 1011 would be stored ..001011
1 10 // with the least significant byte first
```

```
2 0
3 0
4 0
5 0
6 0
7 0
```

This is on a system with 4 byte ints and 8 byte doubles.

We might also put structs as the things in a union:

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

typedef struct
{
    short s1;
    short s2;
} TwoShorts;

typedef struct{
    char c1;
    char c2;
    char c3;
    char c4;
} FourChars;

// define the type..
typedef union MyUnion { // space for 2 shorts OR 4 chars
    TwoShorts twoShorts;
    FourChars fourChars;
} UnionType;

int main(int argc, char** argv) {
    UnionType un1;
    un1.fourChars.c1='a';
    un1.fourChars.c1='b';
    un1.fourChars.c1='c';
    un1.fourChars.c1='d';
    un1.twoShorts.s1=5;
    un1.twoShorts.s1=9;
    printf("%hd\n", un1.twoShorts.s1);

    return (EXIT_SUCCESS);
}
```

The expected purpose of a union is to share memory for different uses at different times. It is therefore most widely used in embedded systems.

Test

Write and test code which:

1. Creates 10 Cheque strcuts as in the example above

2. Writes them out into a binary file

3. Reads them back and displays them.

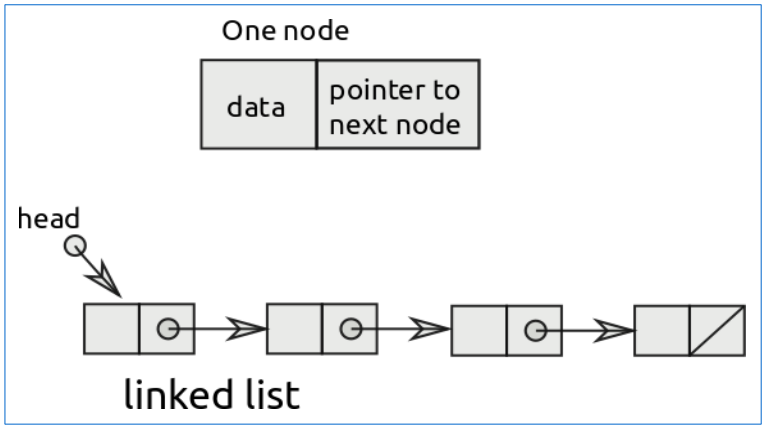
Linked list example

Linked list data structure

An array has a set of elements 'in a line', so is a type of list. An array is usually stored as a single block of memory. Elements can be accessed directly by index. However it is not possible to add or remove elements from an array, since it uses that fixed block of memory.

A linked list is a set of nodes, each node having a pointer to the next node. The nodes are usually not next to each other in memory - they can be anywhere. This means it is easy to add and remove nodes at runtime:

Each node contains two fields. These are some data (usually a key and values) and a pointer field, which points to the next node.



Exactly what the pointer is depends on the implementation. In C it would be a RAM address of the next node. In Java it would be a reference.

The diagram shows how linked lists can be thought of. There is a variable called 'head' (usually) which is a pointer, and it points to the first node of the list. In the last node the next pointer has a special 'null' value, meaning it points nowhere. It is usually drawn as shown.

Linked list in C

We can have the following files:

llist.c file defining the relevant functions

llist.h include file, declaring what is needed to use the linked list implementation, and

main.cto test and use the linked list implementation.

The implementation

Llist.c follows. We try to choose identifier names and insert comments to make the code obvious:

```
// library includes
#include <stdlib.h>
// local includes
#include "llist.h"

// function definitions
Node * newNode(int number) // create a new node
{
 Node * node = (Node *) malloc(sizeof(Node));
 node->data=number;
 node->next=NULL;
return node;
}

ListPtr newList() { // create a new (empty) list
 ListPtr llp = (ListPtr) malloc(sizeof (struct ListStruct));
 llp->head = NULL;
 return llp;
}
```

```
void put(ListPtr llp, Node * newNode) { // put a node at the head of a list
 if (llp->head==NULL)
 {
 llp->head=newNode;
 return;
 }
 newNode->next= llp->head ;
 llp->head=newNode;
}

void putInt(ListPtr list, int n) // put an int value at the head
{
    Node * n1 = newNode(n);
    put(list, n1);
}

void printNode (Node * whichNode) // output a node
{
 printf("Node at %p data=%d next node at %p\n", whichNode, whichNode->data, whichNode->next);
}

void traverse(ListPtr llp) { // traverse list head to end, outputting data
 if (llp->head==NULL) return;
 Node * where = llp->head;
 while (where != NULL) {
 printNode(where);
 where = where->next;
 }
 printf("End of list\n");
 return;
}
```

Some of these functions do not need to be available externally – for example, not newNode. We just need a public function to add an int to the list, and that would call newNode internally.

The interface

Llist.h is:

```
#ifndef LLIST_H // header guard
#define LLIST_H

// typedefs
typedef struct NodeStruct{
 int data;
 struct NodeStruct * next; // pointer to next block
} Node;

typedef struct ListStruct {
 Node * head; // pointer to first node
} * ListPtr;

// function prototypes
ListPtr newList();
void traverse(ListPtr llp);
void putInt(ListPtr list, int n);

#endif /* LLIST_H */
```

We only have 3 externally available functions – make a new list, add an int to a list, and traverse it to test.

Code to use the list implementation

main.c is

```
#include "llist.h"
```

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

 ListPtr list = newList();
 putInt(list, 7);
 putInt(list,4);
 putInt(list,3);


 traverse(list);

    return 0;
}
```

The output from this is:

```
Node at 0x1029070 data=3 next node at 0x1029050
Node at 0x1029050 data=4 next node at 0x1029030
Node at 0x1029030 data=7 next node at (nil)
End of list
```

Test

1. Copy this and check it works

2. Add the ability to

- test if a list contains a given int
- find the length of a list
- remove the first occurrence of an int from a list
- keep the list in increasing order of data – that is, largest int at the head. This is then a priority queue

Topics after K&R

Void pointers

A void pointer is a *pointer to anything*, like

```
    void * ptr;
```

There are various rules, logically required, about void pointers.

This is OK:

```
    void * ptr;
    double x=3.5;
    ptr=&x;
    printf("ptr is %p\n", ptr); // 0x7ffcc969ae08
```

but this will not compile:

```
    double y=*ptr;
```

We cannot dereference a void pointer. When we dereference a pointer, we pick up the bytes starting at the address, corresponding to the type – so for a double (probably) eight bytes. But ptr here points to anything, so it does not know how many bytes to pick up.

We can say this:

```
     double y =  *((double *) ptr);
```

Here we first say (double *) ptr to typecast ptr to a pointer to a double; then we can dereference it.

Calloc and malloc return a void *, since we might store anything in the memory claimed. We would typecast it to some named type before use.

Inline functions

In computing in general, an inline function has its code duplicated at each function use.

Suppose we have

```
int f(..)
{
..function code
return x;
}
.. calling code
f(..); // point 1
..
f(..); // point 2
```

At point 1, we get machine code which will first

- push actual parameters on the stack
- push the return address on the stack
- call the start address of f

At the start of f, code will

- pop the formal parameters off the stack
- execute the function body
- at the return, pop the return address off the stack, push the return value onto the stack
- jump to the return address

Then calling code will pop the return value off the stack and use it.

The same is done at point 2. So we just have 1 copy of the function, and jump and return at each call.

For an inline function, the function code is simply *copied* to points 1 and 2.

So using an inline function:

- Is slightly faster, not using the call/return stack mechanism

- Uses more memory, duplicating code for every function use

The gcc compiler appears to use rules slightly different from C99. An inline function needs to be declared static (this file only) or extern (this file and others):

```
#include <stdio.h>

static inline void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
    return;
}

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

    int a = 1;
    int b = 2;
    swap(&a, &b);
    printf("%d %d\n", a, b);
    swap(&a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

The standard says "Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined."

Complex type

This supports complex number types. If you do not know about complex numbers, ignore this section.

For example:

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

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

    double complex z1 = 1.0 + 2.0 * I;
    double complex z2 = 2.0 + 3.0I;
    double complex z3 = z1*z2;
    printf("%lf%+lfi", creal(z3), cimag(z3)); // -4.000000+7.000000i

    return 0;
}
```

creal returns the real part and cimag the imaginary part.

For each function in math.h, there is a version in complex.h preceded by c for complex, with no suffix expecting a complex double parameter and returning a complex double result, an f suffix for float, and l suffix for long double. For example:

```
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

carg gives the argument (phase angle) and cabs the norm, so that for example:

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

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


    double complex z1 = 0.0 + 1.0 * I;
    double a = carg(z1);
    printf("%lf", a); // 1.570796

    return 0;
}
```

For compilation in gcc, this requires the use of the -lm flag to cause linking with the maths library.

Variable length arrays

This means arrays with a length fixed at run-time, not compile-time.

It does not mean arrays which can get larger or smaller during execution. For example:

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

int main(int argc, char *argv[]) {
    // initialize the random number generator depending on the current time
    time_t t;
    srand((unsigned) time(&t));

    int size = rand() % 9 + 1; // random 1 to 10
    int data[size]; // make array that random size
    printf("Size = %d\n", size);
    for (int index = 0; index < size; index++)  // its full of garbage
        printf("index = %d %d\n", index, data[index]);

    return 0;
}
```

gets removed

gets is a function which inputs a string from the console. From C11 it was removed, because it is dangerous and should not be used. Why not?

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

int main(int argc, char *argv[]) {
    char * buffer=(char *)malloc(5);
    printf("Input up to 4 chars..");
    gets(buffer);
    printf("You entered:%s",buffer);
    return 0;
}
```

Then buffer is enough space to hold 4 input characters plus a final zero, and gets(buffer) inputs data and stores it there. But suppose we input more than that? For example:

```
Input up to 9 chars..12345xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
You entered:12345xxxxxxxxxxxxxxxxxxxxxxxxxxxxYou entered:12345xxxxxx
```

The input went into the buffer, filled it, then spilled over to what was next in memory. In this case this was the string assembled by the printf "You entered.." so we get crazy output.

In general we can get a buffer over-run and corrupt memory – so gets cannot be used.

What to use instead? fgets is an option:

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

int main(int argc, char *argv[]) {
    const int BUFFER_SIZE=6;
    char * buffer=(char *)malloc(BUFFER_SIZE);
    printf("Input up to 5 chars..");
    fgets(buffer, BUFFER_SIZE, stdin);
    printf("You entered:%s\n",buffer);

    return 0;
}
```

The manual says:

The fgets function reads at most one less than the number of characters specified by n (second parameter) from the stream pointed to by stream (first parameter) into the array pointed to by s (third parameter). No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

So it will read up to 5 chars, including a newline if possible, then put a zero at the end. For example

```
Input up to 4 chars..123456789
You entered:12345
```

Here it read in 12345, then put a 0 at the end

```
Input up to 4 chars..123
You entered:123
```

Here it read in 123<newline> and it put that in the buffer, again followed by a zero – so we get 2 newlines.

We cannot get a buffer over-run.

Generic macros

In computing in general, *generic code* works for *any data type*. The problem is to reconcile that with a language like C which is *strongly typed* – variables must have known fixed type.

From C11 the solution is a new keyword, _Generic. This is used in an expression which evaluates to something - for example

```
_Generic( (i), char: 1, int: 2, long: 3, default: 0);
```

then if i is an int, this is 2, but if its a long, its 3. So

```
    int i;
    int v = _Generic( (i), char: 1, int: 2, long: 3, default: 0);
    printf("%d\n", v); // 2
```

We can put that in a macro:

```
#define type_idx(T) _Generic( (T), char: 1, int: 2, long: 3, default: 0)
```

so that

```
    printf("%d\n",type_idx(1L)); // 3
```

As an example – suppose we would like a function print(x), which would work for any data type of x – only we need a different format specifier for each type. So..

```
#define fSpec(T) _Generic( (T), char: "%c\n", int: "%d\n", double: "%lf\n", default: "")

#define print(T) printf(fSpec(T), T)

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

    int i = 4;
    print(i); // 4
    double d = 3.142;
    print(d); // 3.142000

    return 0;
}
```

Alignment

We can imagine memory as a series of boxes, each with an address, going 0,1,2,3,4,5..., with one address for each byte.

But most types are 2 or 4 or 8 bytes long – so how do these values fit onto byte address boundaries? This is what *alignment* means.

For example

```
    int a,b,c;
    printf("%p %p %p\n",&a, &b, &c);
    // 0x7ffd036e1acc 0x7ffd036e1ad0 0x7ffd036e1ad4
```

So a is stored at address ..acc. Then b is at ..ad0, which is 4 bytes up from that. And c is at ..ad4, or 4 bytes up from that. Which makes sense, because on this system ints are 4 bytes long.

But C11 introduces new macros, alignas and alignof:

```
    int alignas(long) a,b,c;
    printf("%p %p %p\n",&a, &b, &c);
    // 0x7ffd9193cf50 0x7ffd9193cf58 0x7ffd9193cf60
```

Now the 4 byte ints are aligned on 8 bytes boundaries.

This also works:

```
    int alignas(8) a,b,c;
```

but

```
    int alignas(5) a,b,c;
```

produces

```
error: requested alignment is not a positive power of 2
```

Further projects

Microsoft Windows native code in C

The Windows OS has a GUI as a central part (unlike Linux type OS, for which various alternate GUIs are 'add-ons' to the OS). Windows does not use POSIX-style signals. In place it uses 'messages'. For example, if the mouse is moved across a window, the OS sends a stream on WM_MOUSEMOVE messages to the process which owns the window. The process can take action in response – or ignore the messages.

The standard text about this is the Petzold book.

Explore this if you like.

OpenGL

This is about video-game style 3D graphics.

OpenGL is a specification for what a graphics rendering system should be able to do. Those facilities were at expressed in terms of C functions (but have now been written in most languages). Graphics co-processor chips can now do most of these in hardware (and so are much faster than using the main processor, in software).

A good but rather old set of online tutorials are NeHe. A book is the SuperBible.

Non-Microsoft GUIs

Two options are GTK and QT.

Database

We can write data into files on non-volatile media, as described above.

But if we want very fast very large scale data storage, it is better to use a database server. This software is optimised to handle databases. There are different types of database – relational, network and hierarchical – but relational is the most common. Some well-known servers are Oracle, MySQL, Microsoft SqlServer andPostgres. Commands are sent to the server in a standard language known as SQL.

Database theory and practice is a major topic. One text is this.

Once you have become competent in topics such as normalised design, an option is to see how C programs can connect with a database server, issue SQL commands and process returned resultsets – as here.