

Introduction to Computer Programming in C

Level 4 course: 2014

Professor A. Carrington

Correlated Electron Systems Group

Room G.23, a.carrington@bristol.ac.uk

Introduction : Structure of computing in physics course

Level 4: **Computational Physics**: Introduction to basics of programming in C

Level 5: **Computational Physics**: Numerical problem solving of Physics problems

Level 6: (Msci only) Advanced numerical techniques + project work

Level 6: (BSc) / Level 7 (MSci) Projects

Introduction : Structure of course

Instruction

4 x 1 hour lectures

OPTIONAL 2 hour help sessions in room 1.14 (in physics) so that students can DEBUG their own code.

Demonstrators will be available during these help sessions

Please direct questions to the demonstrators during these help sessions.

Introduction : Structure of course

Assessment

There are 3 assessed exercises (Exercise 1 consists of multiple small parts).

The exercises will be worth 20% of the mark for the level 5/6 course.

Marks will be given for functionality but also programming style. Is the code efficient and well set out ? (use comments). As exercise 1 consists of short pieces of code, marks will be awarded solely on the basis of whether a good attempt has been made or not. For the other two exercises marks will be given for functionality but also programming style. Is the code efficient, easy to understand and logically laid out?

**** Note that plagiarism will be checked for electronically. Detection will normally result in, a) award of mark of zero, b) repeat exercises, c) noted on record**

**** Don't do it....you will be caught*****

Introduction : Structure of course

Assessment : Submitting your code

Submit via Blackboard

Submit ONLY the c code (for example, main.c) not the executable code. **Make sure you submit the correct files. If you don't you may get zero marks for your efforts.**

Only use standard ANSI c (as documented in these notes) – use of non-standard functions may prevent us running your code and hence result in a lower mark. If you're not sure check the code runs with the mingw (gcc) compiler using codeblocks on the 1.14 computers.

The deadline for the first two exercises is :

Friday 13th June, last day of this term.

The deadline for the 3rd exercise is

Monday 29th September 2014, Monday of Week 1 of NEXT academic year

Standard faculty penalties for late work will be applied.

You will receive feedback for exercise 1 and 2 during the summer (by email) and for the 3rd exercise in approximately week 3 of next term.

Introduction : Learning Objectives

To become proficient at writing simple computer code using the C programming language. Specifically the course will cover:

Simple variables and arrays

Operators

Branching

Looping

Input and output: console/keyboard and also via files

String manipulation

Functions

Dynamics arrays and pointers

Introduction : Why learn C?

Uses:

Solving problems numerically (virtually all real world physics problems are solved numerically)
Controlling equipment / processing data

C and C++ combined are the most popular programming languages. This is especially true for scientific / engineering computing.

Low level code vs high level code or packages

Low level code: Very versatile, fast, can do anything

Packages: Optimised for specific tasks (e.g., plotting graphs, fitting data), usually slower than low level code.

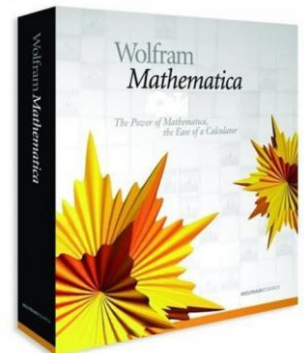
MATLAB
The Language of Technical Computing



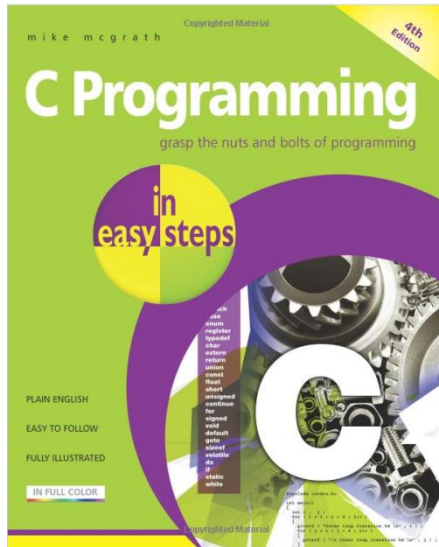
OriginLab



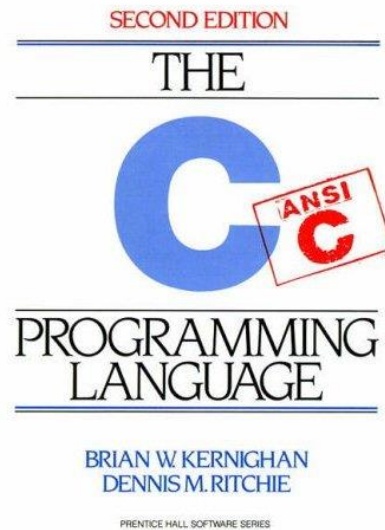
Maplesoft
Mathematics • Modeling • Simulation
Maple 13
The Essential Tool for Mathematics and Modeling



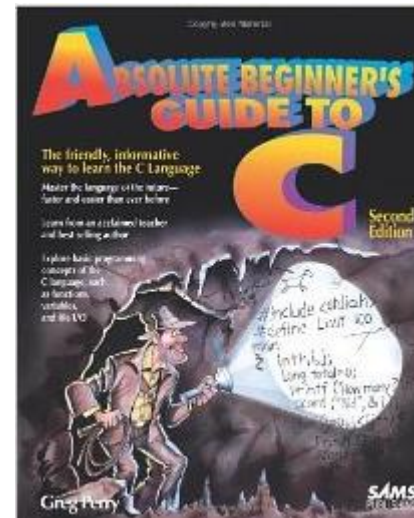
Introduction : Books



Cost <£10 (Amazon)



Cost £34 (Amazon)



Cost £14 (Amazon)

Free Online stuff

http://en.wikibooks.org/wiki/C_Programming

The C Book

by Mike Banahan, Declan Brady and Mark Doran

http://publications.gbdirect.co.uk/c_book/



Introduction : Hardware / Software

Course will focus (mainly) on standard C: so can be run on any computer with a C compiler (Windows PC, Mac, Linux)

Computers in 1.14 use Windows 7 OS

Integrated Development Environment (IDE)

Strictly not needed. Could just use a text editor (notepad) + compiler (e.g., gcc)

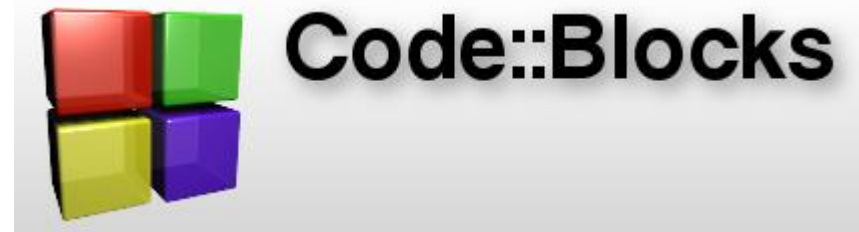
BUT

Does make life a lot easier.

Introduction : Choice of IDE

I will use

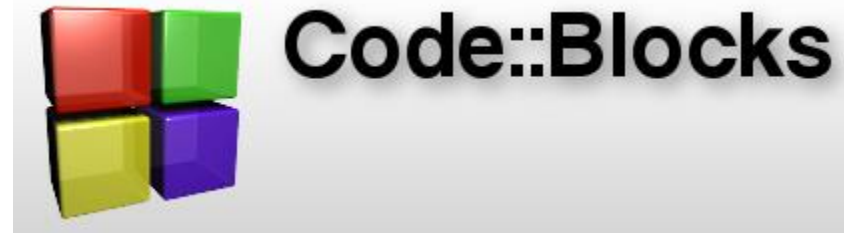
Code::Blocks (cross platform)



http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

IDE	License	Windows	Linux	Mac OS X	Other platforms	Debugger	GUI builder	Integrated toolchain	Profiler	Code coverage	Autocomplete	Static code analysis	GUI-based design	Class browser	Latest stable release	C compiler	C++ compiler
Anjuta	GPL	No	Yes	No	FreeBSD	Yes	Yes	Yes	Yes	Unknown	Yes	Unknown	Yes	Yes	2010 09		
C++Builder	Proprietary	Yes	No	No		Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	2009 08		
Code::Blocks	GPL	Yes	Yes	Yes	FreeBSD	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes ^[4]	Yes	2010 05 ^[6]	Yes (MinGW + custom)	Yes (MinGW + custom)
CodeLite	GPL	Yes	Yes	Yes	FreeBSD, Mac OS	Yes	Yes	Yes	No	No	Yes	Yes	Yes, Qt	Yes	2010 12	Yes (GCC + Custom)	Yes (GCC + Custom)
Dev-C++	GPL	Yes	Yes ^[8]	No		Yes	No	Unknown	Yes	Unknown	Yes	Unknown	Yes	Yes	2005 02	Yes	Yes
Eclipse CDT	EPL	Yes	Yes	Yes	JVM	Yes	Yes ^[2]	No	Unknown	Unknown	Yes	Yes	No	Yes	2009 06		
Geany	GPL	Yes	Yes	Yes	FreeBSD, OpenBSD	Yes	No	No	Unknown	Unknown	Yes	Unknown	Unknown	Unknown	2010 06		
GIAT Programming Studio	GPL	Yes	Yes	Yes	Solaris	Yes	Unknown	Yes	Yes	Yes	Yes	Yes	Unknown	Yes	2009 06		
GNUstep ProjectCenter	GPL	Yes	Yes	Yes		Yes	Yes	Yes	No	No	No		Yes	Yes	0.5.0		
KDevelop	GPL	No ^[7]	Yes	Yes	FreeBSD, Solaris	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	2011 01	External	External
LccWin32	Freeware / Proprietary	Yes	Yes (obsolete)	No		Yes	Yes (unstable)	Yes	Yes	Unknown	Yes	Yes	Yes	Unknown	Unknown		
MonoDevelop	LGPL	Yes	Yes	Yes	FreeBSD	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes	2011 01	Yes (GCC + Custom)	Yes (GCC + Custom)
NetBeans C/C++ pack	CDDL	Yes	Yes	Yes	Solaris	Yes ^[8]	Yes ^[8]	Yes ^[9]	Yes ^[8]	Yes	Yes	Yes	Yes	Yes	2010 06		
OpenWatcom	OSI	Yes (32-bit only)	partial	No	MS-DOS, OS/2, FreeBSD	Yes GUI remote	Yes	Yes	Yes	No	No	No	Yes	Yes	2009 02		
PellesC	Freeware	Yes	No	No	Windows CE	Yes	Yes	Yes	Unknown	Unknown	Yes	Unknown	Unknown	Unknown	2009 08		
Qt Creator	GPL / LGPL /	Yes	Yes	Yes	FreeBSD, Symbian, Maemo	Yes	Yes	Unknown	No	No	Yes	No	Yes	Yes	2011 03		

Installing Code::Blocks + C-compiler



Download Code::Blocks from

<http://www.codeblocks.org/downloads/binaries>

for PC, you need to download the version which has the **mingw** compiler with it
codeblocks-13.12mingw-setup.exe

For Mac, you will need to install the xcode software

<https://developer.apple.com/xcode/>



Download it from the app store. In order to complete the exercises you will need to install the Command Line Tools. These are not installed by default (last time I checked).

Almost all Linux packages have a c-compiler already installed.

Starting out: Using Code::Blocks

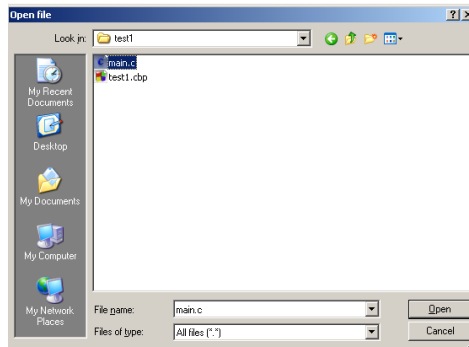
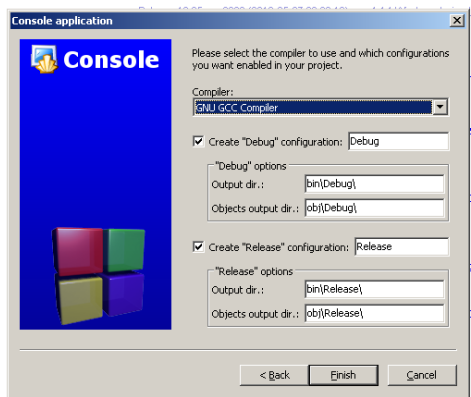
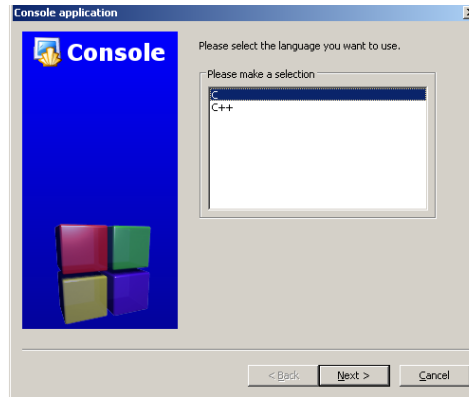
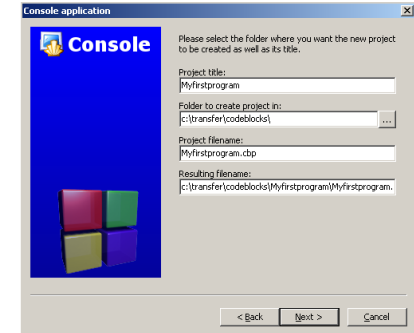
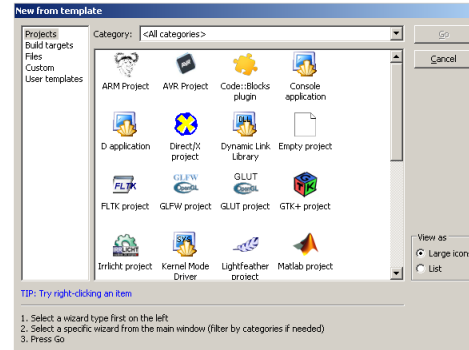
Create a new project

Console application

Give it a name (project title)

Select GNU GCC compiler

File: Open: (navigate to folder with name given above) then open main.c



Starting out: Hello world

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

Standard Libraries

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

All C code starts with the function main()

starts new line

Lines end with semicolon ;

all programmes (functions) end like this

Braces {} show limit of main function

To get it to work:

Build and run, F9

Build and Run

the **COMPILER** takes the text file (main.c) and converts this into machine code (instructions the processor can understand). This produces an object file: main.o

The **LINKER** then takes the object code and links it with other code in various libraries which perform common tasks (like printing to the screen). This produces an **EXECUTABLE** file: myprogramname.exe

BUILDING a programme compiles it then links it, producing the executable

The **EXECUTABLE** file is the complete instruction set for the processor. This is run by either typing its name (in a command window) or clicking on it.

Exercise: Locate the executable file created by the compiler and run it (double click or command line)

From a command line, (Windows/ Mac/ Linux) you can direct the output into a file
myprogramme > myoutput.txt

Critical error

A possible problem occurs if you have installed the wrong version of Code::Blocks. Code::Blocks is just a IDE (an integrated editor) it does not contain (by default) a compiler. If you get the message below (or some variant)

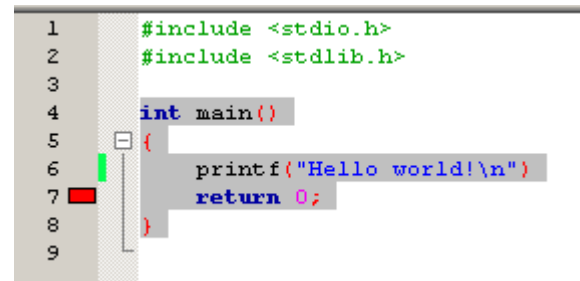
"Example_For - Debug" uses an invalid compiler. Probably the toolchain path within the compiler options is not setup correctly?! Skipping...
Nothing to be done.

This means that Code::Blocks cannot find the compiler. You need to install the version which has the mingw (gcc) compiler included. If you are working on a mac you need to install the x-code software.

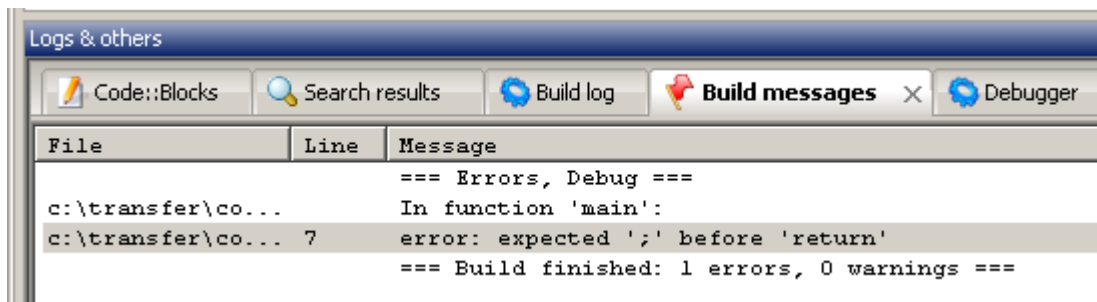
Compile time errors

Missing semicolon from the end of the printf line. A very common error !

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



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n")
7      return 0;
8  }
9
```



Variables

variables represent computer memory spaces which store data

```
int main()
{
    int i,j,k;
    float x, part1;
    double part2;

    i=1212;
    j=3;
    k=i*j;
    x=34523.34535;
    part2=34.3434;

    return 0;
}
```

Variables

The basic DATA TYPES

int - data type

int is used to define integer numbers.

```
{ int count; count = 5; }
```

float - data type

float is used to define floating point numbers.

```
{ float miles; miles = 5.6; }
```

double - data type

double is used to define high precision and large floating point numbers.

It reserves twice the storage for the number

```
{ double weight; weight = 2500.234564; }
```

char - data type

char defines characters

```
{ char letter; letter = 'x'; }
```

Variables

Modifiers

Some of the data types can have the following modifiers

short	E.g.,
long	
signed	short int
unsigned	long int
	long long int

The **const** keyword is used to create a read only variable. Once initialised, the value of the variable cannot be changed but can be used just like any other variable.

```
const double Pi=3.14159265358979;
```

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

```
...
```

```
const double Pi=4.0*atan(1.0);
```

```
Or use M_PI (defined in math.h)
```

writing code

!! NOTE that C is case sensitive !!

So

`width`

and

`Width`

are two different variables

also

`Printf` is **not** a function in `<stdio.h>`

but

`printf` is

Comments

Appropriate use of comments is very important element of good programming style

Comments are simple English statements which explain what a particular piece of code does, or gives other information: For example – who wrote the code.

You can also use comments to temporarily take out bits of unused code (e.g., for debugging or other testing purposes).

There are two ways of making comments, these are illustrated below

```
#include <stdio.h>
#include <stdlib.h>
/*Comments longer than
one line can be added like this
*/
int main()
{
    int i,j,k;

    i=1212;
    j=3;
    k=i*j; // single line comments can be added like this

    return 0;
}
```

Variables: Limits

Depends on machine and compiler. Can find limits using the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>
int main()
{
    printf("int max = %d\n", INT_MAX);
    printf("long int max = %d\n", LONG_MAX);
    printf("short int max = %d\n", SHRT_MAX);
    printf("float max = %e\n", FLT_MAX);
    printf("float digits precision = %d\n", FLT_DIG);
    printf("double max = %e\n", DBL_MAX);
    printf("double digits precision = %d\n", DBL_DIG);
    getchar(); // used to pause programme (waits for keypress)
    return 0;
}
```

Output for typical 32 bit machine

```
int max = 2147483647
long int max = 2147483647
short int max = 32767
float max = 3.402823e+038
float digits precision = 6
double max = 1.797693e+308
double digits precision = 15
```

Writing to screen

Example:

```
printf ("The value is %f",45.6);
```

Format Specifier	Description	Example
%d	decimal integer	23
%f (or %lf)	floating point number	344.3432
%e, %E	Exponential format	3.2323E+023
%g, %G	Shorter of f or e	3.2323E+023
%c	single character	's'
%s	string of characters	"Hello World"

We can specify the number of digits displayed

%7d displays always seven places. The number is right justified and the other spaces are filled up with spaces

%10.8f displays 10 places with 7 places after the decimal point

\n starts a newline

**** displays a \

Input from keyboard

Use the scanf function

Like the printf function there are field specifiers which determine how the input is read

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

int main()
{
    int i;

    printf("Please enter a number \n");
    scanf("%d",&i); //Note the & before the i
    printf("i=%d i^2=%d",i,i*i);
}
```


Example 2

You need the %lf specifier to read a double. To see this try the following:

```
int main()
{
    double x;
    printf("Please enter a number \n");
    scanf("%f",&x);
    printf("x=%f x^2=%f",x,x*x);
}
```

This produces an error. The problem is that we've tried to place a normal precision (float) into a double precision variable (double) by using the %f specifier.

Change the %f to %lf and all will be well.

Doing Arithmetic

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	decrement

`i++;` // add 1 to i, equivalent to `i=i+1;`

`i--;` // decreases i by 1;

`i%j ;`// divides i by j and returns the remainder (e.g., `5%3 = 2`)

More shorthand

Operator	Example	Equivalent
=	a=b	a=b
+=	a+=b	a=(a+b)
-=	a-=b	a=(a-b)
=	a=b	a=(a*b)
/=	a/=b	a=(a/b)
%=	a%=b	a=(a%b)

Comparison and logic

Operator	Function
==	Equality
!=	Not equals (inequality)
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Operator	Function
&&	AND
	OR
!	NOT

<math.h> : most important functions

Function	Purpose
<code>sin(x)</code> , <code>cos(x)</code> , <code>tan(x)</code>	trig functions
<code>asin(x)</code> , <code>acos(x)</code> , <code>atan(x)</code>	inverse trig functions
<code>sinh(x)</code> , <code>cosh(x)</code> , <code>tanh(x)</code>	hyperbolic trig functions
<code>exp(x)</code> , <code>log(x)</code> , <code>log10(x)</code>	e^x , $\ln(x)$, $\log_{10}(x)$
<code>pow(x,y)</code> , <code>sqrt(x)</code>	x^y , $x^{1/2}$

Note that x^y does not raise x to the power of y in C. You need to use the `pow` function instead

<complex.h> : Complex numbers

Function	Purpose
<code>_Complex_I</code>	$i = \sqrt{-1}$, a constant
<code>csin(z)</code> , <code>ccos(z)</code> , <code>ctan(z)</code>	trig functions
<code>casin(z)</code> , <code>cacos(z)</code> , <code>catan(z)</code>	inverse trig functions
<code>csinh(z)</code> , <code>ccosh(z)</code> , <code>ctanh(z)</code> , <code>casinh(z)</code> , <code>cacosh(z)</code> , <code>catanh(z)</code>	hyperbolic trig functions (and inverse)
<code>cexp(z)</code> , <code>clog(z)</code>	e^z , $\ln(z)$
<code>conj(z)</code>	conjugate of Z
<code>cpow(z,y)</code> , <code>csqrt(z)</code>	z^y , $z^{1/2}$
<code>creal(z)</code> , <code>cimag(z)</code>	Real / imaginary parts
<code>cabs(z)</code> , <code>carg(z)</code>	Magnitude / phase

All except the last two rows return double complex type, the last two are just double.

<complex.h> : Complex numbers: example

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
int main()
{
    double complex z1,i;

    i=_Complex_I;//simplify the syntax
    z1=2.323+0.1223*i;//define a constant

    printf("Z    = %6.4f+%6.4fI\n",z1);
    printf("1/Z = %6.4f+%6.4fI\n",1/z1);
    printf("Z'   = %6.4f+%6.4fI\n",conj(z1));
    return 0;
}
```

<stdlib.h> : Random numbers

`int rand(void)`

returns a pseudo-random number between 0 and a maximum (RAND_MAX)

`void srand (unsigned int seed)`

sets the seed for the random number generator

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

```
int main()
{
    double x;
    int i;

    for (i=0;i<5;i++)
    {
        x=rand() / (double)RAND_MAX; //note the use of a 'cast' here
        printf("%f\n", x);
    }
}
```

OUTPUT

```
0.001251
0.563585
0.193304
0.808741
0.585009
```


Casts

Consider the following

```
int i,j;  
double x;  
  
i=10;  
j=3;  
x=i/j;  
printf("x=%f",x);
```

output

```
x=3.000000
```

why? the compiler sees that i and j are both integers so performs integer division throwing away the decimal remainder, BEFORE it places the result in the double x

replace the line

```
x=i/j
```

with

```
x=i/(double)j;
```

output

```
x=3.333333
```

The (double) operator tells the compiler to treat j as a double. Hence, it then performs the division using floating point operators. This is called a TYPE CAST.

Bytes, Bits and other storage

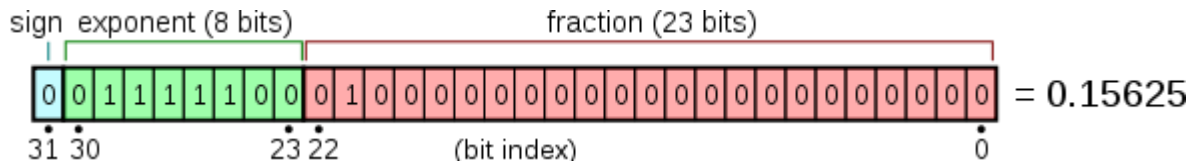
Bit = 1 or 0 , this is the most basic element of number storage

Byte = 8 bits, i.e, 00000000 to 11111111 in binary = 0..255 in decimal or 0..FF in Hexadecimal

All numbers are stored and manipulated as bits in computers. Different number types are stored differently: even if they are the same number ! Hence if you mix types things can go wrong.

Small positive integers can be stored as a simple byte

Floating point numbers: e.g., 4 byte float are much more complex



2's complement encoding (you can ignore this!)

2's complement representation of a N bit number

positive number = simple binary representation

negative number = binary rep of number then flip all bits and add 1

All negative numbers have the most significant bit (far left) = 1

e.g.,

$$00000101_2 = 5_{10}$$

$$11111011_2 = -5_{10}$$

Addition

$$\begin{array}{r} 00000101 \\ + 11111011 \\ \hline 100000000 \end{array}$$

so $5 + -5 = 0$



ignore this because result is restricted to 8 bits

Loops: for(){}

General format

```
for (initialiser ; test expression ; incrementer)
{
    Statements
}
```

EXAMPLE

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

int main()
{
    int i;
    for (i=0; i<10; i++)
    {
        printf("i=%d, i^2=%d\n", i, i*i);
    }
    return 0;
}
```

OUTPUT

```
i=0, i^2=0
i=1, i^2=1
i=2, i^2=4
i=3, i^2=9
i=4, i^2=16
i=5, i^2=25
i=6, i^2=36
i=7, i^2=49
i=8, i^2=64
i=9, i^2=81
```

Loops: for

EXAMPLE 2

```
for (i=0; i<30; i=i+3)
{
    printf("i=%d, i^2=%d\n", i, i*i);
}
```

OUTPUT

```
i=0, i^2=0
i=3, i^2=9
i=6, i^2=36
i=9, i^2=81
i=12, i^2=144
i=15, i^2=225
i=18, i^2=324
i=21, i^2=441
i=24, i^2=576
i=27, i^2=729
```

Loops: while (true){}

General format

```
initialiser
while (test expression)
{Statements;incrementer;}
```

EXAMPLE

```
int main()
{
    int i;
    i=0; //initialiser
    while (i<10)
    {
        printf("i=%d,i^2=%d\n",i,i*i);
        i++; // incrementer
    }
    return 0;
}
```

OUTPUT

```
i=0, i^2=0
i=1, i^2=1
i=2, i^2=4
i=3, i^2=9
i=4, i^2=16
i=5, i^2=25
i=6, i^2=36
i=7, i^2=49
i=8, i^2=64
i=9, i^2=81
```

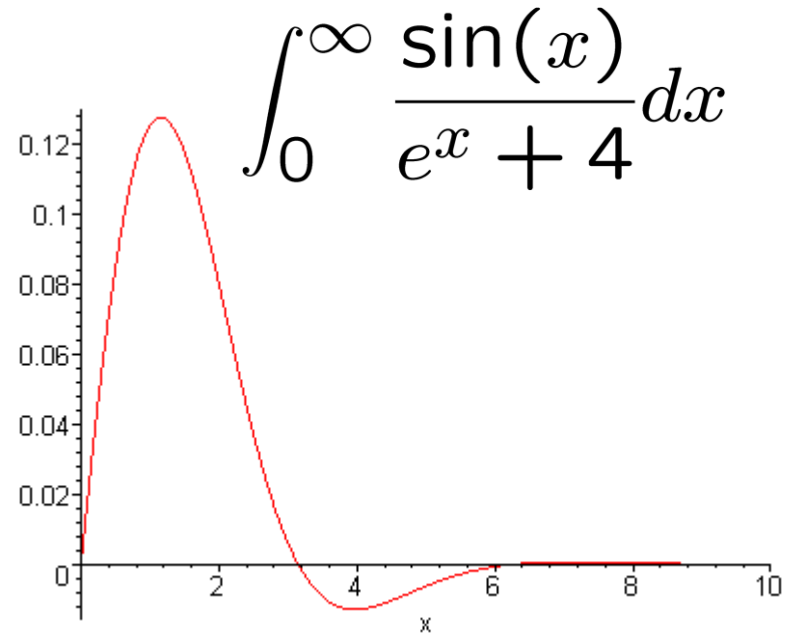
Example 2: while

Poor man's integration

```
int main()
{
    double x,xstep,sum;

    sum=0;
    x=0;
    xstep=1e-4;

    while (x<100)
    {
        x=x+xstep;
        sum=sum+sin(x)/(exp(x)+4)*xstep;
        if (fmod(x,10)<xstep) printf("%5.0f %12.9f\n",x,sum);
    }
    printf("%12.9f",sum);
    return(0);
}
```



This is not a particularly good method

fully converged answer = 0.2090574164

Arrays : single dimension

```
int main()
{
    float data[5]; //First array element is 0, so data[5] i.e., the 6th
                  // element is not defined here

    int i;

    data[0]=3.1;
    data[1]=12.2;
    data[2]=74.4;
    data[3]=67.33;
    data[4]=343.2;
    printf("Element number 3 = %6.2f\n",data[3]);
    //now print all of them
    printf("\nAll the elements\n");
    for (i=0; i<5; i++)
    {
        printf("Element %d %6.2f\n",i,data[i]);
    }
    return 0;
}
```

What happens if we try to use array element data[5] which is not defined ?

Arrays : single dimension (2nd example)

```
int main()
{
    int i,myarray[30];
    //initialise array elements all to zero
    for (i=0; i<30; i++)
    {
        myarray[i]=0;
    }
    //now fill the array with values we want to use later
    for (i=1;i<30;i++)
    {
        myarray[i]=i*i;
    }
    printf("Array element 5 is %d\n",myarray[5]);
    return 0;
}
```

OUTPUT

Array element 5 is 25

Arrays : multiple dimension

```
int main()
{
    int i,j,myarray[5][2];
    //initialise array elements all to zero
    for (j=0; j<2; j++)
    {
        for (i=0; i<5; i++)
        {
            myarray[i][j]=0;
        }
    }
    //now fill the array with values we want to use later
    myarray[3][1]=4;
    myarray[3][2]=3;
    printf("Array element 3,2 is %d\n",myarray[3][2]);
    return 0;
}
```

declares and array with 5 rows and 2 columns

Nested loops

HELP my code does not work: Debugging

Use the **debugger** built into CODEBLOCKS

With this you can step through you code line by line checking the contents of the variables, step by step.



To use this you **MUST** save you program as a PROJECT
Set a breakpoint
Start debugging

The screenshot shows the CodeBlocks IDE with a C program in a file named `main.c`. The code is as follows:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int i,j,myarray[5][2];
7      //initialise array elements all to zero
8      for (j=0; j<=1; j++)
9      {
10         for (i=0; i<=4; i++)
11         {
12             myarray[i][j]=0;
13         }
14     }
15     //now fill the array with values we want to use later
16     myarray[3][1]=4;
17     myarray[3][2]=3;
18     printf("Array element 3,2 is %d\n",myarray[3][2]);
19     return 0;
20 }
21
```

The Watches window is open, showing the following variables and their values:

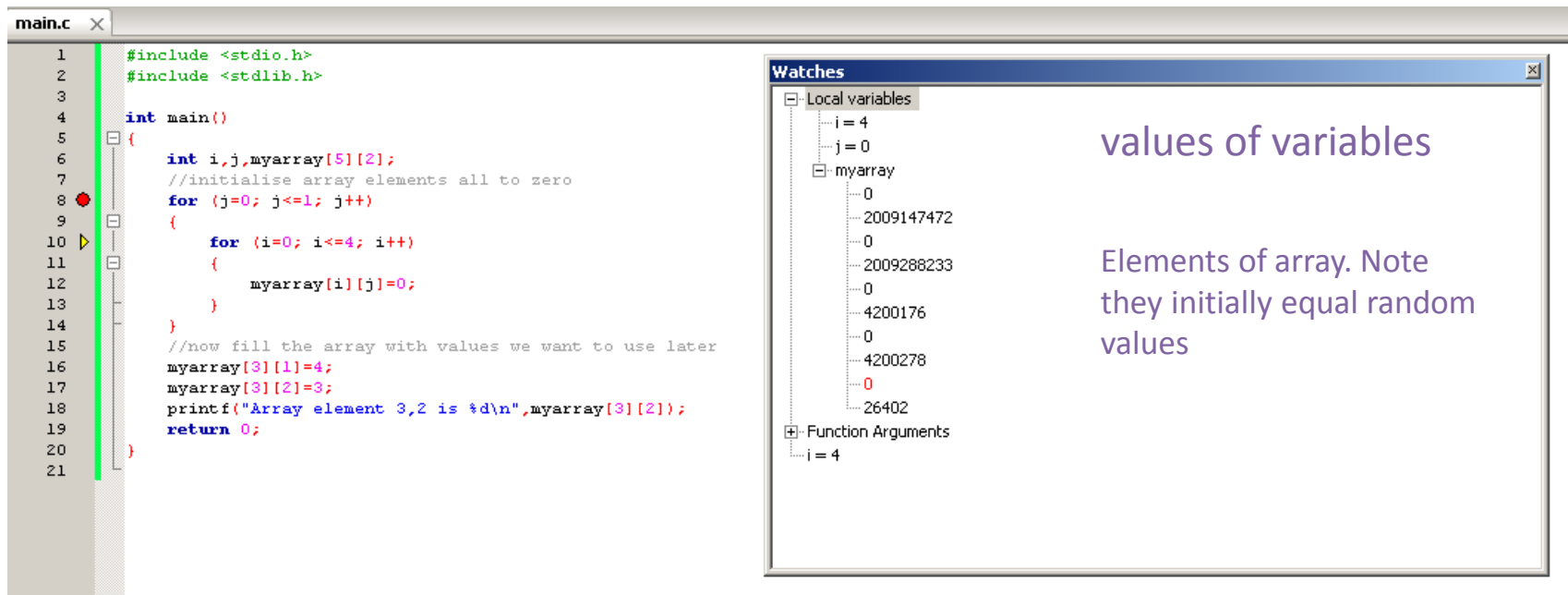
- Local variables
 - `i = 4`
 - `j = 0`
 - `myarray`
 - 0
 - 2009147472
 - 0
 - 2009288233
 - 0
 - 4200176
 - 0
 - 4200278
 - 0
 - 26402
- Function Arguments
 - `i = 4`

Debugging

(this obviously will vary depending on the IDE / platform you are using)

- 1) Set breakpoint
- 2) Start debugger
- 3) Step through code line by line looking at variables in the watch list

breakpoint



The screenshot shows a debugger window with a C program named `main.c`. The code is as follows:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int i,j,myarray[5][2];
7     //initialise array elements all to zero
8     for (j=0; j<=1; j++)
9     {
10         for (i=0; i<=4; i++)
11         {
12             myarray[i][j]=0;
13         }
14     }
15     //now fill the array with values we want to use later
16     myarray[3][1]=4;
17     myarray[3][2]=3;
18     printf("Array element 3,2 is %d\n",myarray[3][2]);
19     return 0;
20 }
21
```

A red dot indicates a breakpoint is set at line 8. A yellow arrow points to line 10, indicating the current execution point. To the right, the 'Watches' window displays the following variables and their values:

- Local variables
 - `i = 4`
 - `j = 0`
 - `myarray`
 - 0
 - 2009147472
 - 0
 - 2009288233
 - 0
 - 4200176
 - 0
 - 4200278
 - 0
 - 26402
- Function Arguments
 - `i = 4`

values of variables

Elements of array. Note they initially equal random values

Current line

Decision making and Branching

METHOD 1a

```
if (expression is true)
{do this};
```

Example

```
if (i%2==1)
{
    j=1
};
```

or

Compact version

```
if (i%2==1) j=1;
```

METHOD 1b

```
if (expression is true)
{do this}
else
{do this};
```

```
if (i%2==1)
{
    j=1;
}
else
{
    j=0;
}
```

METHOD 1c

```
if (expression is true)
{do this}
else if
{do this}
else
{do this}
```

Decision making and Branching

METHOD 2 (shorthand version)

(if expression is true)? do this : else do this;

Example

```
myarray[i] = (i%2==1) ? 1 : 0;
```

tests if $i \bmod 2 = 1$

if it is (i.e., i is odd) then `myarray[i]` is set to 1 otherwise it is set to 0

if statement: Example 1

Example

```
int main()
{
    int j;

    printf("Enter a number:");
    scanf("%d",&j);
    if (j==3)
    {
        printf("you inputted 3");
    }
    else
    {
        printf("you did not input 3");
    }
}
```

if statement: Example2

Example

```
int main()
{
    int j;

    printf("Enter a number:");
    scanf("%d",&j);
    if (j==3)
    {
        printf("you inputted 3");
    }
    else if (j==5)
    {
        printf("you inputted 5");
    }
    else
    {
        printf("you did not input 3 or 5");
    }
}
```


if statement: tests

You can use the OR operator (||) and the AND operator (&&) to combine tests

See table of **Comparison and logic** operators

```
int j;
printf("Enter a number:");
scanf("%d",&j);
if ((j==3) || (j==5))
{
    printf("you inputted either 3 or 5");
}
else
{
    printf("you did not input 3 or 5");
}
```

If statements more examples

`if ((j>3) && (j<6))` True if j is greater than 3 but less than 6

`if (j!=3)` True if j is not equal to 3

`if ((j>3) && (j!=6)) || (j==1)`
True if j is greater than 3 and not equal to 6, OR j equals 1



Equality tests on floats

Consider the following

```
float x;
printf("Enter a number:");
scanf("%f",&x);
if (x==5.0)
{
    printf("you inputted 5");
}
```

You might think that if you enter 5 at the prompt then the if statement would be true.

This might be correct but it depends on the compiler and the system precision. For example, depending on the number of significant digits 5 could be represented internally as 5.00000000000001, and then the if statement would be false. Hence

AVOID EQUALITY TESTS ON FLOATING POINT NUMBERS

instead use

```
#include <math.h>

int main()
{
    float x;
    const float eps=1e-9;
    printf("Enter a number:");
    scanf("%f",&x);
    if (fabs(x-5.0)<eps)
```

***fabs()** takes the absolute value of a floating point number*

***abs()** does the same thing for an int*

Switch...case

An alternative to the `if() {} else {}` statement is the `switch..case` statement

```
printf("Enter a number:");
scanf("%d",&i);

switch (i)
{
case 1:
    printf("You entered 1");
    break;
case 2:
    printf("You entered 2");
    break;
case 3:
    printf("You entered 3");
    break;
default:
    printf("You entered something else");
}
```

Note the use of the `break` keyword

Pointers

Variables are stored in memory. Pointers work with the memory location of a variable rather than directly with the variable itself. The reason you might want to do this will become clearer later.

Normal variable

```
int j;  
j=10;  
printf ("%d",j);
```

First line declares the variable (reserves a space in computer memory for it)

Second line, put the value 10 into that space in memory

Third line retrieves the value in the memory location and print the ascii character it represents on the screen

Pointers

```
int i; //declare integer i
int *p; //declare a pointer to an integer

i=23; //assign 23 to i
p=&i; // assign the pointer p to the address of i
*p=10; // assign the memory location pointed to by p to the value 10
      // the operation *p is called dereferencing p
printf("%d",i); // print i (i now equals 10)
```

Pointers

```
#include <stdio.h>

int main()
{
    int i,j; //declare two integers i and j
    int *p; //declare a pointer to an integer

    i=23; //assign 23 to i
    p=&i; // assign the pointer p to the address of i
    j=*p; // assign j to the value p points to (*p dereferences p)
    printf("%d",j); // print j on the screen using the decimal integer %d format specifier
    return 0; //end the programme returning 0 to indicate it has terminated normally
}
```

Functions

Use functions to simplify code structure and also to maximise reuse of code

Example

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

float volume (float height, float width, float length );
/*this is called a function prototype.  You place it before the main{} function.  It
declares the function and its arguments but does not actually contain the code.  The
code is specified later after the main{} function.  The header files #include <math.h>
etc contain function prototypes.  The actual code is linked in later with the linker*/

int main()
{
    float dim1,dim2,dim3;

    dim1=23.5;
    dim2=789.232;
    dim3=1.099;
    printf("The volume is %f",volume(dim1,dim2,dim3));
}

float volume (float height, float width, float length )
//this is the actual function
{
    return height*width*length;
}
```


Functions

If you don't want the function to return anything then use the `void` data type

```
void printdoublex(float x);
```

```
int main()
{
    float height;
    height=3.54;
    printdoublex(height);
}
```

```
void printdoublex(float x)
{
    printf("%f",2*x);
}
```

Functions II: Passing information both ways

Need to use pointers

```
float volumechangeheight (float *pheight, float width, float length );

int main()
{
    float dim1,dim2,dim3;

    dim1=23.5;
    dim2=789.232;
    dim3=1.099;
    printf("The variable height was %f\n",dim1);
    printf("The volume is %f\n",volumechangeheight(&dim1,dim2,dim3));
    printf("The variable height is now %f",dim1);
}

float volumechangeheight (float *pheight, float width, float length )
{
    *pheight=*pheight+2.0;/**pheight is the variable pointed to by pheight
    return (*pheight)*width*length;
}
```

Strings

Generally, a string is a collection of characters: e.g., “Hello”
They are usually stored in memory as their ASCII character codes

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	_	127	7F	DEL

So “Hello” is stored as: 72,101,108,108,111 in decimal

Strings

In C there is no special type for strings. They are represented as an array of type `char` terminated by a special null character `\0`

`Char` represents a single character. It is a number between 0 and 255 (00 to FF in hex)

Assignment

```
char string1[100]="Hello";
```



This is OK

```
char string1[100];  
string1="Hello";
```



This doesn't work, you can't assign an array to an array in this way. Instead copy it

```
char string1[100];  
strcpy(string1, "Hello");
```



This is OK

Strings: converting numbers to strings

Use the `sprintf` function. It works very similarly to `printf`

```
double x;  
char s[256];  
  
x=2323.2323;  
  
sprintf(s, "%lf", x);
```

s now equals "2323.232300"

Strings: Functions in <string.h>

Function	what it does
<code>strlen(s)</code>	returns the length of the string
<code>strcpy(s1,s2)</code>	copies s2 to s1
<code>strncpy(s1,s2,n)</code>	copies n characters of s2 into s1
<code>strcmp(s1,s2)</code>	returns 0 if s1 is exactly the same as s2
<code>strcat(s1,s2)</code>	Concatenates strings. Adds s2 to the end of s1
<code>strncat(s1,s2,n)</code>	add n characters of s2 to s1
<code>strchr(s,c)</code>	locate character c in string s
<code>strstr(s1,s2)</code>	locate substring s2 in string s1

Note: When using `strcat` or `strcpy` you must make sure there is sufficient room in the destination string for the copy / concatenated strings.

Strings: Function in <ctype.h>

The Standard C library has functions you can use for manipulating and testing character values. The first 7 functions return a non-zero value if the test is TRUE, otherwise they return 0 (FALSE). The last 2 are for character conversion.

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

```
int islower(ch);  
int isupper(ch);  
int isalpha(ch);  
int isdigit(ch);  
int isalnum(ch);  
int ispunct(ch);  
int isspace(ch);  
char tolower(ch);  
char toupper(ch);
```

Strings: function in <stdlib.h>

atof (s)	converts s to a double
atoi (s)	converts s to an int

Strings: Example 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
int main ()
{
    char string1[100],string2[12];
    strcpy(string1,"Hello");//contains 5 characters (bytes)
    strcpy(string2," or Goodbye");//contains 11 characters (bytes)
    strcat(string1,string2); //need to have enough space in string 1 to fit string1+string2
    printf("%s\n",string1);

    return 0;
}
```

Strings: Example use of functions

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
int main ()
{
    char string1[100];
    int stringlength,i;
    strcpy(string1,"Hello or Goodbye");
    stringlength=strlen(string1);
    for (i=0; i<stringlength; i++)
    {
        if (islower(string1[i]))
            printf("Char %2i is \"%c\" and is LOWERCASE\n",i,string1[i]);
        if (isupper(string1[i]))
            printf("Char %2i is \"%c\" and is UPPERCASE\n",i,string1[i]);
        if (!isalpha(string1[i]))
            printf("Char %2i is \"%c\" and is not Alpha\n",i,string1[i]);
    }
    //now convert the whole string to uppercase
    for (i=0; i<stringlength; i++)
    {
        string1[i]=toupper(string1[i]);
    }
    printf("The string has now been converted to upper case\n");
    printf("%s\n",string1);

    return 0;
}
```

Strings: Example use of functions:Output

```
Char 0 is "H" and is UPPERCASE
Char 1 is "e" and is LOWERCASE
Char 2 is "l" and is LOWERCASE
Char 3 is "l" and is LOWERCASE
Char 4 is "o" and is LOWERCASE
Char 5 is " " and is not Alpha
Char 6 is "o" and is LOWERCASE
Char 7 is "r" and is LOWERCASE
Char 8 is " " and is not Alpha
Char 9 is "G" and is UPPERCASE
Char 10 is "o" and is LOWERCASE
Char 11 is "o" and is LOWERCASE
Char 12 is "d" and is LOWERCASE
Char 13 is "b" and is LOWERCASE
Char 14 is "y" and is LOWERCASE
Char 15 is "e" and is LOWERCASE
The string has now been converted to upper case
HELLO OR GOODBYE
```

Strings: Another example

Could be use when reading data from a file where numbers and text are mixed

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
int main ()
{
    char string1[100],numbit[100];
    float width;

    strcpy(string1,"Width=34.5565");//load values into string1
    strcpy(numbit,strchr(string1,'=')+1);//find the position of the = sign
                                   //then copy the substring into numbit
    width=atof(numbit);//do the conversion, if numbit is not numeric then the conversion will
                                   // fail and width=0
    printf("The numbers was %f",width);//display the result of the conversion
    return 0;
}
```

Strings: yet another example

```
void numify(char *s)
//this function removes any characters from s which are not numbers (0..9) or decimal point
{
    int i,j;
    char *sout=NULL; //define an array but don't specify its size yet (it will be dynamic)

    sout=(char *) malloc(sizeof(char)*(strlen(s))); //reserve enough space for the answer
                                                    //this is a dynamic array (see later)
    j=0;
    for (i=0;i<strlen(s);i++)
    {
        if (isdigit(s[i])||((s[i]=='.')))
        {
            sout[j]=s[i];
            j++;
        }
    }
    sout[j]='\0'; // add a null termination to the string
    strcpy(s,sout); //copy the numify-ed string back to s
    free(sout); //free the memory which the dynamic array occupied
}
```

Strings: yet another example

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

void numify(char *s);

int main ()
{
    char string1[100], numbit[100];
    double width;

    strcpy(string1, "Width=34.5565");
    numify(string1); //this is the new bit !
    printf("%s\n", string1);
    width=atof(string1);
    printf("The numbers was %f", width);
    return 0;
}
```

Defining your own data structures : struct

```
typedef struct
{
    float height,weight;
    int hairsUpNose;
} mydatatype;
//declares the datatype

int main()
{
    mydatatype mydata;//declares a variable of mydatatype type

    mydata.height=175.2; //assigns a value to the field of the data type
    mydata.weight=79.23;
    mydata.hairsUpNose=23;
}
```

Using user defined data types generally improves the readability of the code and also makes passing parameters to functions more efficient (you can pass one `struct` instead of 10 variables)

Files: Output

Writing out data in tab delimited file

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ()
{
    FILE *myfile; //declare a variable to point to the file
    float x,y;
    int i,imax=100; //imax is declared and assigned at the same time

    myfile=fopen("myoutput.txt","w"); //opens the file for output
    if (myfile!=NULL) //check to see if the file was actually opened
    {
        for (i=0; i<imax; i++)
        {
            x=i/(float)imax *2*M_PI; //float keyword used to cast integer variable
            y=sin(x); //need the math library for this and M_PI
            fprintf(myfile,"%f%c%f\n",x,9,y); //character 9 is the tab character
        }
        fclose(myfile); //close the file
    }
    printf("All done"); //display something on the screen
}
```

File “myoutput.txt” now contains

```
0.000000      0.000000
0.062832      0.062791
0.125664      0.125333
0.188496      0.187381
0.251327      0.248690
0.314159      0.309017
```


Files

File mode	Operation
r	open an existing file to read
w	open a file to write. If the file exists already it contents will be overwritten. If the file does not already exist it will be created.
a	Appends a text file. Opens an existing file for writing and appends the newly written information to the end of the file.

Escape sequences in printf, fprintf etc

Directing output to specific folders in Windows

need to use a double slash \\ in the fopen statement

```
myfile=fopen("c:\\data\\myoutput.txt","w");
```

Special Character	Escape Sequence
backslash	\\
backspace	\b
carriage return	\r
double quote	\"
formfeed	\f
horizontal tab	\t
newline	\n
null character	\0
single quote	\'
vertical tab	\v
question mark	\?

Files: Input

Reading data back in from tab delimited file

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main ()
{
    FILE *myfile;
    float x,y,myarray[1000][2];
    int i,imax=100;

    myfile=fopen("c:\\data\\myoutput.txt","r");
    if (myfile!=NULL) //check file actually opened ok
    {
        for (i=0; !feof(myfile); i++)
        {
            fscanf(myfile,"%f%f",&x,&y); //read the values in
            myarray[i][0]=x;
            myarray[i][1]=y;
        }
        // data is now in the array called myarray, and i equals the number of elements-1
        fclose(myfile);
        //write some stuff to the screen to prove it worked
        printf("Read in %d pairs of numbers\n",i);
        printf("line 54 read %f\t%f\n",myarray[54][0],myarray[54][1]);
    }
    else
    {
        printf("File not found");
    }

    printf("All done");
    return 0;
}
```

Dynamic Arrays

How to make an array where you do not know the size of space needed until the programme is running. For example: reading data from a file of unknown length.

1) declare pointer to array (of desired type)

```
int *myarray
```

2) Allocate memory to the array

```
myarray = (int*) malloc(10000*sizeof(int));
```

`(int*)` is a type cast, `sizeof(int)` return the size in bytes of the data type `int` so this statement declares an array of 10000 elements

3) When you're finished with the array free it from memory

```
free(myarray);
```

Dynamic Arrays

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    double *array;
    long int n;

    n=10000000; //10^7
    array = (double *)malloc(n*sizeof(double));
    array[23344]=56.2;
    printf("%lf\n",array[23344]);
    free(array);
    return 0;
}
```

Dynamic Arrays

Better code: checks if memory allocation worked before proceeding

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    double *array;
    long int n;

    n=100000000; //10^7
    array = (double *) malloc(n*sizeof(double));
    if (array!=NULL) //this line checks if the memory was allocated
    {
        array[23344]=56.2;
        printf("%lf\n",array[23344]);
        free(array);
    }
    return 0;
}
```

Dynamic Multidimensional Arrays

In principle you can use single dimensional arrays to perform the same function. You just need to keep track of where the elements are stored. For example, if you wanted to define a 10x10 grid, you could do the following

```
int main()
{
    int i,j;
    int data[100];
    int data2[10][10];

    //using 1 dimensional array
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            data[i*10+j]=i*10+j;
        }
    }

    printf("Array 1 Element [4,5]= %d\n",data[4*10+5]);

    //using multidimensional array
    for (i=0; i<10; i++)
    {
        for (j=0; j<10; j++)
        {
            data2[i][j]=i*10+j;
        }
    }

    printf("Array 2 Element [4,5]= %d\n",data2[4][5]);

    return 0;
}
```

Dynamic Multidimensional Arrays

If you have to have a multi-dimensional dynamic array to the following

```
double** ar;//name of array. The ** means it's a pointer to a pointer
int dim1,dim2,i;

//declare the size of the array you want to create dim1 x dim 2
dim1=1000;
dim2=100;

//allocate memory for the array. Technically this is an array of pointers
//Should have error checking here to make sure that the memory required was actually available
ar= (int**)malloc(dim1 * sizeof(double*));
for (i = 0; i < dim1; i++)
{
    ar[i] = (int*)malloc(dim2 * sizeof(double));
}
//looks complicated but after you've done this you can just use the array elements as normal
//now do something with the space....
//just a simple example, set a particular element then read it back

ar[345][56]=45.45464;
printf("Element=%8.5f",ar[345][56]);

//finally when were done with array we need to free up the memory
for (i = 0; i < dim1; i++)
{
    free(ar[i]);
}
free(ar);
```


Dynamic Arrays

Other functions useful for working with dynamic arrays:

`calloc(numberofelements, elementSize);`

works like `malloc` but initialises all the elements to zero. Note it has 2 arguments, whereas `malloc` only has 1.

`realloc(*pointertoblock, newsize);`

Make a block of memory larger or smaller, while preserving the contents if making larger