

Introduction to C programming

Introduction

Writing, compiling and executing a program

Program structure

Constant and Variable Types

- * Data types
- * Constants
- * Variables
- * Scope
- * Arrays

Expressions and Operators

- * Assignment statement
- * Arithmetic operators
- * Comparison
- * Logical expressions

Control Statements

- * Branches
- * Loops

Functions in C

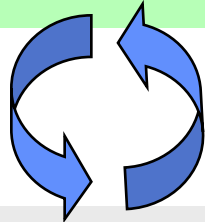
- * Function arguments
- * Function scope

Pointers in C

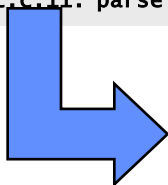
- * Arrays and pointers

Compiling and execution

```
#include <stdio.h>
/* The simplest C Program */
int main (int argc, char **argv)
{
    printf ("Hello World\n");
    return 0;
}
```



```
$ gcc -Wall -g my_program.c -o my_program
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and
separately
tt.c:8: `x' undeclared (first use in this
function)
tt.c:8: (Each undeclared identifier is
reported only once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-
void function
tt.c: At top level:
tt.c:11: parse error before `return'
```



my_program

1. Write text of program (source code) using an editor such as emacs, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:

```
$ gcc -Wall -g my_program.c -o my_program
```

-Wall -g ?

3-N. Compiler gives errors and warnings; edit source file, fix it, and re-compile

N. Run it and see if it works 😊

```
$ ./my_program
```

```
Hello World
```

```
$
```

./?

What if it doesn't work?

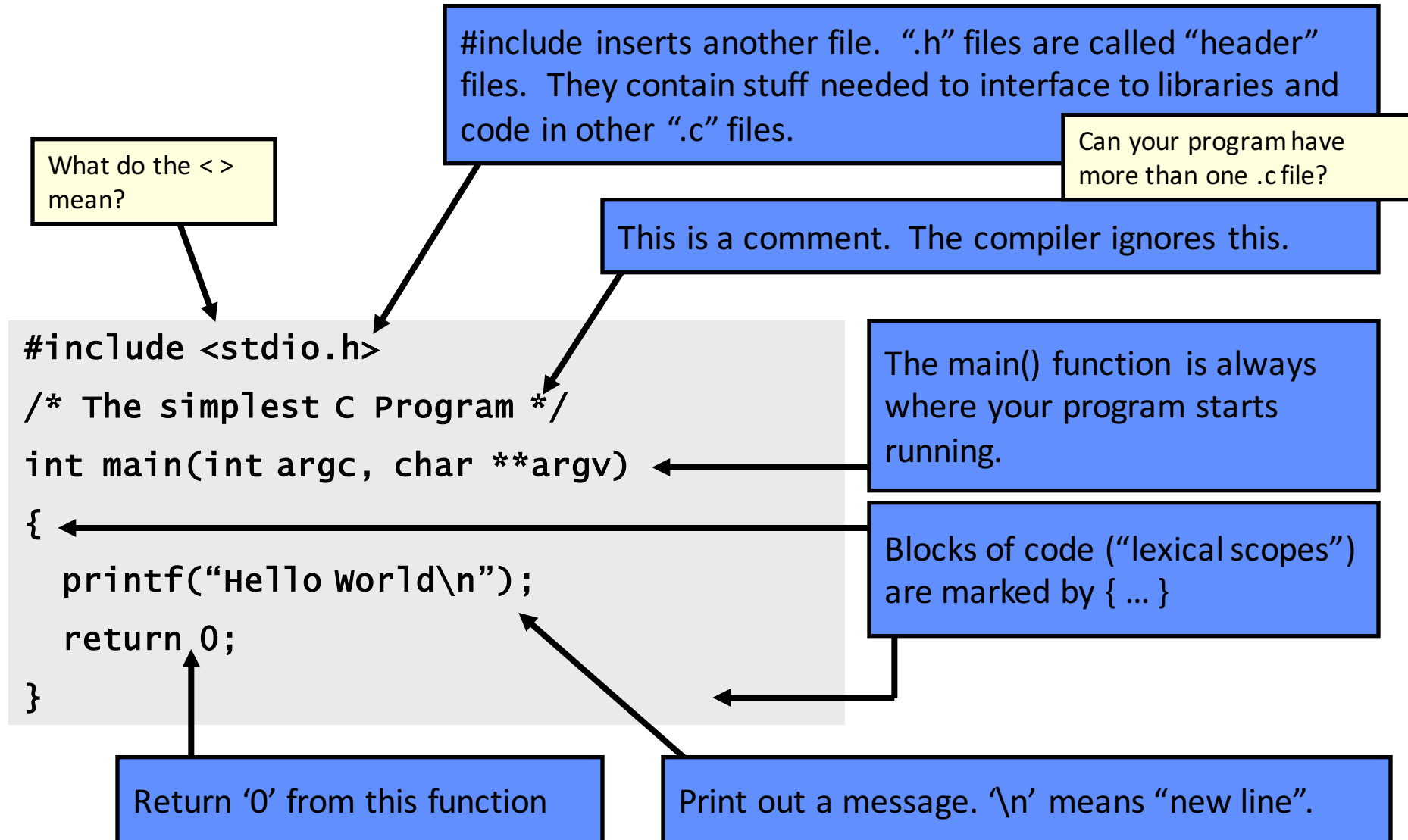
Structure of a C program

- A **C** program is composed by one or more functions, and one of them is called **main**.
- The program always starts its execution from function **main**. From this function it is possible to call other functions.
- C functions have
 - Header.
It includes the function name and the parameters (if any).
 - Variables declaration.
 - The sequence of sentences to be executed.

C program example

```
global variables.  
/* This is a comment */  
function1( )  
{  
    local variables  
    code  
}  
/* Main program*/  
main( )  
{  
    local variables  
    code  
}
```

Compiling and execution



What is “Memory”?

Memory is like a big table of numbered slots where bytes can be stored.

The number of a slot is its **Address**.
One byte **Value** can be stored in each slot.

Some “logical” data values span more than one slot,
like the character string “Hello\n”

A **Type** names a logical meaning to a span of
memory. Some simple types are:

`char`
`char [10]`
`int`
`float`
`int64_t`

a single character (1 slot)
an array of 10 characters
signed 4 byte integer
4 byte floating point
signed 8 byte integer

not always...

Addr	Value
0	
1	
2	
3	
4	'H' (72)
5	'e' (101)
6	'l' (108)
7	'l' (108)
8	'o' (111)
9	'\n' (10)
10	'\0' (0)
11	
12	

72?

Variables in C

<type> <*name*> ;

- Variable type indicates the amount of memory required to store that variable
- Variable names can include letters, numbers and _
- C is case sensitive

[Qualifier] <type> <*name1*>, <*name2*>, <*name3*>=<*value*>, <*name4*> ;

What is a Variable?

A **Variable** names a place in memory where you store a **Value** of a certain **Type**.

You first **Define** a variable by giving it a name and specifying the type, and optionally an initial value

```
char x;  
char y='e';
```

Initial value of x is undefined

Initial value

Name

Type is single character (char)

The compiler puts them somewhere in memory.

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

C types

Type	Size	Range
-		TRUE or FALSE
char	1 byte	-128 a 127
int	4 bytes	-2147483648 a 2147483647
float	4 bytes	-2147483648 a 2147483647
double	8 bytes	1'7 E-308 a 1'7 E+308

Declaration of variables

- Simple declaration:
 - `char c;`
 - `unsigned int i;`
- Multiple declaration:
 - `char c,d;`
 - `unsigned int i,j,k;`
- Declaration and initialization:
 - `char c='A' ;`
 - `unsigned int i=133,j=1229;`

Type qualifiers

- **Signed (default)**

- **Unsigned**

It indicates that the variable is an absolute value.

unsigned char	1 byte	0 to 255
unsigned int	2 bytes	0 to 65535

- **Short (default)**

Value range in short format (limited).

- **Long**

Value range is long format (extended).

long int	4 bytes	-2.147.483.648 to 2.147.483.647
long double	10 bytes	-3'36 E-4932 to 1'18 E+4932

What is a Constant?

It allows you to declare a variable as non-changing

the values of variables declared as const remain constant throughout the life of the program.

Examples:

```
#define pi 3.141516
```

```
#define A 5
```

```
#define FileName "c:\\Temp\\Test.txt"
```

```
const int index = 5;
```

```
const char president[16] = "Abraham Lincoln";
```

```
const int factor = 35;
```

```
const float twoThirds = 2./3.;
```

Variables scope

- Variables can be: Global or Local
- **Global** variables can be used from any point and must be declared before main function (**main()**).
- **Local** variables can only be used inside the function they have been declared. They should be declared after the { symbol

Examples

```
/* Variables declaration */
#include <stdio.h>      /* I/O library */

/* global variables */
    int a=10;
unsigned    int b=20;

main() /* Shows two values */
{
    int b=4;
    printf("b is local and its value is %d\n",b); /* b is 4 */
    printf("a is global and its value is %d",a); /* a es 10 */
}

void function()
{
    printf("b and a globals: %d, %d\n",a,b);
    /*a=10,b=20*/
}
```

Arithmetic operators

- Operators with 2 operands

+ Addition

- Substraction

*** Multiplication**

/ Division

% Module (rest).

Sintax:

<variable1><operator><variable2>

- Operators with 1 operand

++ Increment (add 1)

-- Decrement (sub 1)

- change sign

Sintaxis:

<variable><operador>

<operador><variable>

Examples

```
int a=1, b=2,c=3,r;
```

```
r = a + b;
```

```
r = r * 2;
```

```
r = r % 4;
```

```
r = r / 3
```

```
c++;
```

```
++c;
```

```
r = c++ * 3;
```

```
c= 5;
```

```
r = ++c * 3;
```

Examples

```
int a=1, b=2,c=3,r;
```

<pre>r = a + b;</pre>	<pre>/* r equal 3 */</pre>
<pre>r = r * 2;</pre>	<pre>/* r equal 6 */</pre>
<pre>r = r % 4;</pre>	<pre>/* r equal 2 */</pre>
<pre>r = r / 3</pre>	<pre>/* r equal 0 */</pre>
<pre>c++;</pre>	<pre>/* c equal 4 */</pre>
<pre>++c;</pre>	<pre>/* c equal 5 */</pre>
<pre>r = c++ * 3;</pre>	<pre>/* r equal 15 and c equal 6 */</pre>
<pre>c= 5;</pre>	<pre>/* c equal 5 */</pre>
<pre>r = ++c * 3;</pre>	<pre>/* r equal 18 y c equal 6 */</pre>

Assignment operators

= Assignment

-= Substraction

/= Division

+= Addition

***= Multiplication**

%= Module (rest)

**n=n+3 can be
k=k*(x-2) can be**

**n+=3
k*=x-2**

Conditional operators

> greater than

< smaller than

>= greater or equal than

<= smaller or equal than

== equal

!= Different

- These operands return **1** if condition is true and **0** if condition is false.

Conditional sentences (if)

- **if...else**

Syntax:

if (condition) sentence;

The sentence only will execute if condition is true: Otherwise it will continue without executing the sentence.

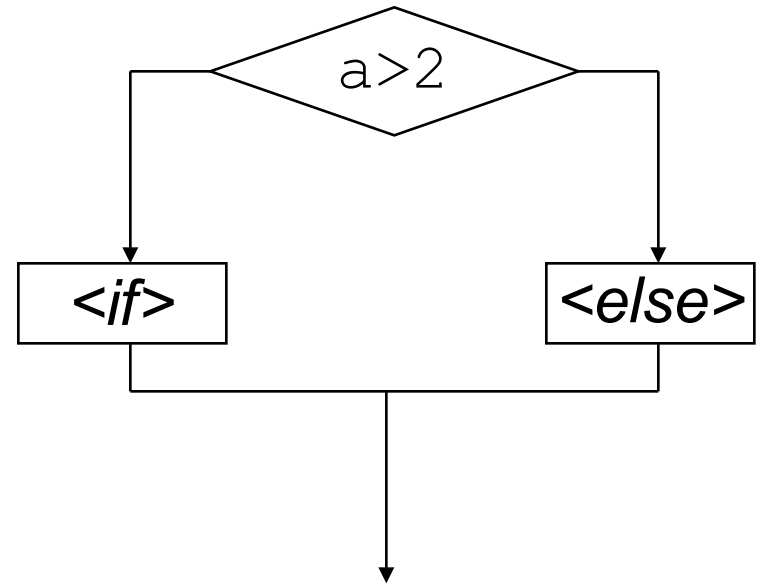
if (condition) sentence1; else sentence 2;

If condition is true then it executes sentence 1. If condition is false then it executes sentence 2. In both cases it will continue after sentence 2.

if ... else

```
int main()
{
    int a=3, b;

    if (a>2)
    {
        b=100+a;
        printf("part if");
    }
    else
        printf("part else");
}
```



Logic operators

- There are three basic logic operators:

&& AND **||** OR

! NOT

Example:

```
if ( (age >= 25) && (age < 65) ||  
    (salary < 1250€) && !Fixed )  
    Wantsalaryincrease=1;
```

Conditional sentences (if/switch)

```
if (condition) sentence1;  
else if (condition) sentence2;  
    else if (condition) sentence3;  
        else sentence4;
```

```
switch (variable){  
    case value1:  
        sentence; break;  
    case value2:  
        sentence; break;  
    default:  
        sentence;  
}
```


Switch example

```
#include <stdio.h>
main() /* Write week day */
{
    int day;
    scanf("Introduce day: %d",&day);
    switch(day)
    {
        case 1:    printf("Monday"); break;
        case 2:    printf("Tuesday"); break;
        case 3:    printf("Wednesday"); break;
        case 4:    printf("Thursday"); break;
        case 5:    printf("Friday"); break;
        case 6:    printf("Saturday"); break;
        case 7:
        {
            printf("Sunday");
            printf("Holiday!!!\n");
            break;
        }
        default:    printf("Not a day");
    }
}
```

Iteratives sentences (loops)

- **WHILE**

Syntax:

while (condition) sentence;

or

while (condition)

{

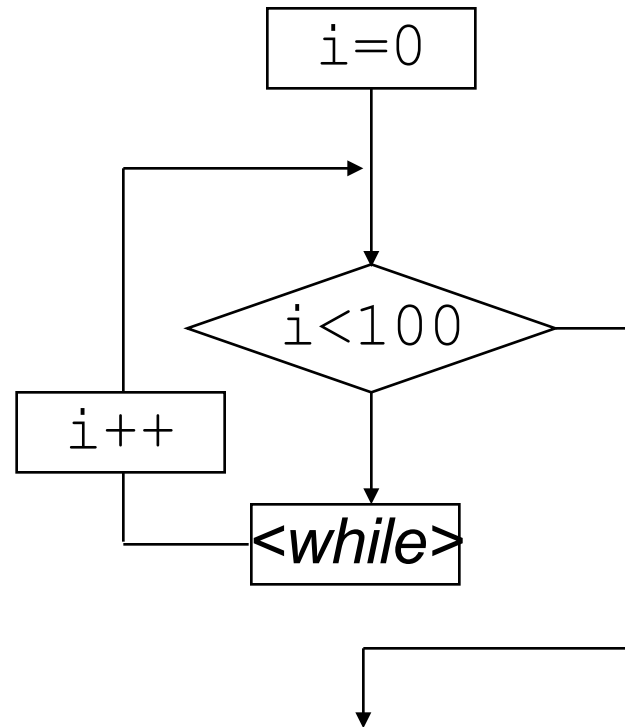
sentence;

}

while sentence checks the condition before entering the loop. If condition is not true then the program does not enter the loop.

while

```
int main()  
{  
    int i=0, ac=0;  
  
    while (i<100)  
    {  
        printf ("%d", i*i) ;  
        ac+=i;  
        i++;  
    }  
}
```



Iteratives sentences (loops)

- **FOR**

Sintaxis:

for (initialization; condition; increment)

{

sentence1;

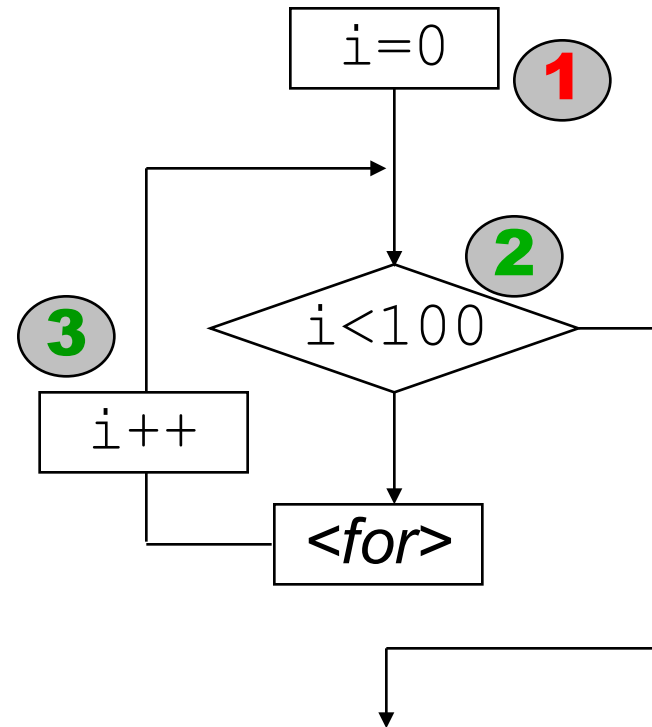
sentence2;

}

- The initialization indicates the control variable that guides the loop repetition.

for

```
int main()  
{  
    int i, ac=0;  
  
    for (i=0; i<100; i++)  
    {  
        printf ("%d", i*i);  
        ac+=i;  
    }  
}
```



Sintaxis:

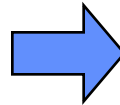
for(**initialization**, **condition**, **permanence**, **increment**)

for vs. while

The “for” loop is just shorthand for this “while” loop structure.

```
float pow(float x, uint exp)
{
    float result=1.0;
    int i;
    i=0;
    while (i < exp) {
        result = result * x;
        i++;
    }
    return result;
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```



```
float pow(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; i < exp; i++) {
        result = result * x;
    }
    return result;
}

int main(int argc, char **argv)
{
    float p;
    p = pow(10.0, 5);
    printf("p = %f\n", p);
    return 0;
}
```

Iteratives sentences (loops)

- **DO...WHILE**

Syntax:

```
do
{
    sentencia1;
    ...
    sentenciaN;
}
while (condición);
```

In this case the condition is verified at the end of the loop. The loop is executed at least once.

Example do-while loop

```
#include <stdio.h>
main() /* Write numbers from 1 to 10 */
{
    int number=1;
    do
    {
        printf("%d\n",number);
        number++;
    }
    while(number<=10)
}
```


Control sequence

- **BREAK**

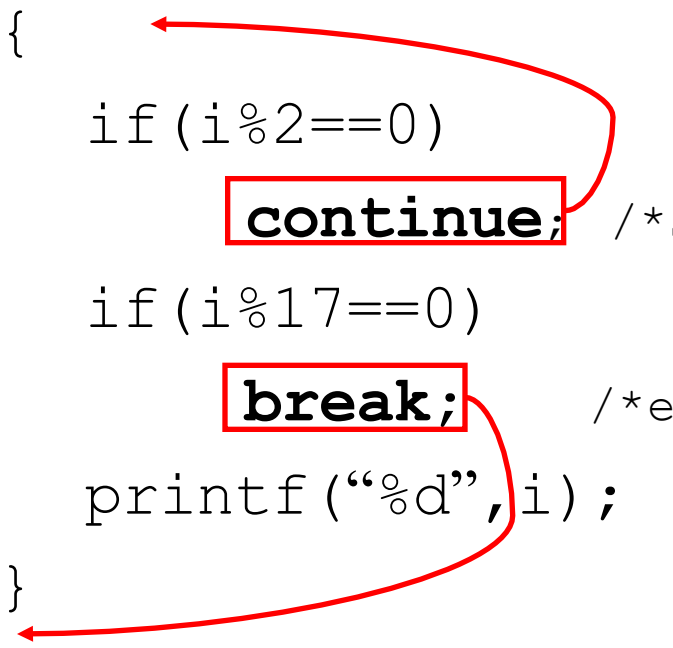
This sentence is used to finish the execution of a loop. Program continues after the loop.

- **CONTINUE**

It is used inside a loop to avoid the execution of the last sentences of some iterations.

break y continue

```
int main()
{
    int i;
    for (i=0; i<100; i++)
    {
        if (i%2==0)
            continue; /*Start the next iteration*/
        if (i%17==0)
            break; /*exit the loop*/
        printf ("%d", i);
    }
}
```



Exercise

- Make a C program that shows a Main menu with 4 options (Start, Open, Store and Quit). The user must select one option. The selected option must be shown on the screen. When the user selects Quit option. The program finishes (Use **loops** and **switch**).

Exercise (I)

```
#include <stdio.h>
main() /* Menu exercise */
{
    char selection;
    do{
        printf("1.- Start\n");
        printf("2.- Open\n");
        printf("3.- Store\n");
        printf("4.- Quit\n");
        printf("Select one option: ");
```

Exercise (II)

```
selection=getchar();
switch(selection){
    case '1':
        printf("Start");
        break;
    case '2':
        printf("Open");
        break;
    case '3':
        printf("Store");
    }
}while(selection!='4');
}
```

Example

```
/* Use of FOR. */  
#include <stdio.h>  
  
main() /* Write multiplication table*/  
{  
    int num,x,result;  
    printf("Introduce number: ");  
    scanf("%d",&num);  
    for (x=0;x<=10;x++){  
        result=num*x;  
        printf("\n%d mul %d = %d\n",num,x,result);  
    }  
}
```

Exercise

- Make a program that calculates the first 100 prime numbers.

Prime numbers

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

#define LAST 100
main() /* Example: calculate prime numbers */
{
    unsigned int num,divisor;
    int isPrime; /* boolean */
    num=1;
    do{
        isPrime=1;
        for (divisor=1;((divisor<num) && (isPrime!=0));divisor++)
        {
            /* It can be divided <> 1 */
            if ((divisor!=1) && (num % divisor == 0)) isPrime=0;
        }
        if (isPrime==1) printf("%d ",num);
        ++num;
    }
    while (num<=LAST);
}
```


Arrays in C

- An **array** is an identifier of a set of data, all of them of the same type.
- Each component of the array is identified by an **index**. The index must be an integer and positive value.
- In **C** the first component of an array is identified by index value 0.

Vectors

- A vector is an *unidimensional array*.
- Declaration:

type name [size];

- Access to vectors:

name[index]

/ Access an especific element*/*

name

/ Access the whole vector*/*

Vectors

first element $\rightarrow 0$, last element $\rightarrow n-1$

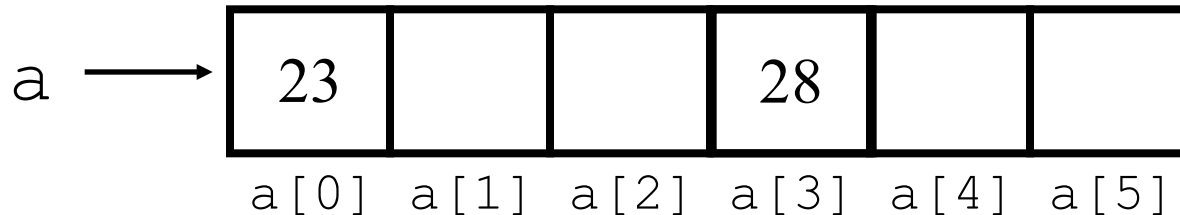
```
int a[6];
```

```
a[0]=23;
```

```
a[3]=a[0]+5;
```

```
for(i=0;i<6;i++)
```

```
    printf("%d", a[i]);
```



Vectors example

```
main() /*Multiplication table */
{
    int vector[10],i,num;

    scanf ("Introduce one number: %d\n", &num);

    for (i=0; i<10; i++)
        vector[i]=i*num;

    for (i=0; i<10; i++)
        printf("%d mul %d = %d\n", i, num, vector[i]);
}
```

Matrix

- A matrix is a multidimensional array.
- Matrixes require an index for each dimension.
- Sintax:

type name [size1][size2]...;

- Access:

name[ind1][Ind2]

Matrix

```
/* Multiplication of a matrix  
and a vector */
```

```
#define SIZE 3
```

```
void main()
```

```
{
```

```
    int vector[SIZE] = {3,3,1};
```

```
    int matrix[SIZE][SIZE] = { {1,1,1}, {2,1,2}, {2,2,2}};
```

```
    int result[SIZE];
```

```
    int i,j;
```

```
    /* initialization of result vector */
```

```
    for (i = 0; i < SIZE; i++)
```

```
        result[i] = 0;
```

```
    /* calculate the result */
```

```
    for (i = 0; i < SIZE; i++)
```

```
        for (j = 0; j < SIZE; j++)
```

```
            result[i] += matrix[i][j] * vector [j];
```

```
    /* print the result */
```

```
    for (i = 0; i < SIZE; i++)
```

```
        printf(" row %d -> %d \n", i, result[i]);
```

```
}
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 2 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 11 \\ 14 \end{bmatrix}$$

Matrix example

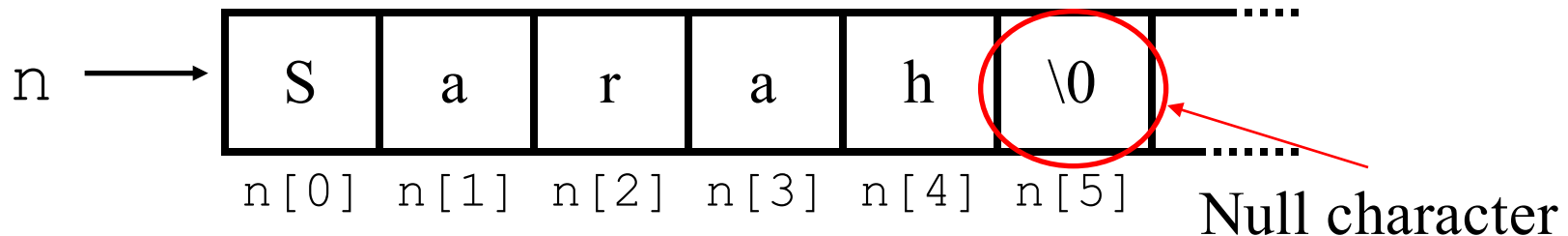
```
main()
{
    int x,y,matrixA[3][3],matrixB[3][3],Res[3][3];
    /* Introduce matrix A*/
    for (x=0; x<3; x++)
        for (y=0; y<3; y++)
            scanf("%d",& matrixA[x][y]);
    /* Introduce matrix b*/
    for (x=0; x<3; x++)
        for (y=0; y<3; y++)
            scanf("%d",& matrixB[x][y]);
    /* Add matrix A and B */
    for (x=0; x<3; x++)
        for (y=0; y<3; y++)
            Res[x][y]=MatrixA[x][y]+MatrixB[x][y];
}
```

Arrays - strings

String is an array of characters of one dimension

```
char n[50]="Sarah";
```

```
printf ("The string is: %s\n", n);
```



Arrays - strings

String assignment:

using operator “=” -> only in declaration

```
char n[50]="Sarah";    /* Correct */  
char x[50];  
x="Another name"; /* Error: this is not  
                    declaration */  
strcpy (x, "Another name");
```

Arrays - strings

String comparison:

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

Compares two strings *s1* and *s2*. It returns an integer

- less than 0 if *s1* is less than *s2*
- equal to 0 if *s1* matches *s2*
- greater than zero if *s1* is greater than *s2*.

Arrays - strings

Example

```
#include <string.h>

...

char a[10] = "Class 1";
char b[10] = "Class 2";
if (strcmp (a, b) == 0)
    printf ("Strings are equal")
else
    printf ("Strings are NOT equal")

...
```

Array initialization

- Syntax:

type name [][]...={ value1, value2...}

Examples:

```
int vector[]={1,2,3,4,5,6,7,8};
```

```
int numbers[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

```
char vector[]="program";
```

```
char days[5][]={"monday","tuesday","wednesday","thursday","friday"};
```

```
char vector[]={ 'p','r','o','g','r','a','m','a','\0' };
```

Functions

A function is a block of code that performs a calculation and returns a value.

A function allows complicated programs to be parceled up into small blocks, each of which is easier to write, read, and maintain.

```
return_type name (argument1, argument2 ...)  
{  
    ...  
    function body  
    ...  
}
```

By default functions return integers (int).

If no value is returned, void must be specified.

Functions

A function must be declared before it can be used.

- Declaration of the function: Function header or prototype
- Definition of the function: Function code

```
int power(int n);
```

```
int power(int n)  
{  
    int ret = 1;  
    ret = n*n;  
    return (ret);  
}
```

Function example

```
int a, b, c;  
int addition (void);  
  
int addition(void)  
{  
    int r;  
    r = a + b;  
    return (r);  
}  
void main(void)  
{  
    a = 2;  
    b = 3;  
    c = addition ();  
    printf("%d + %d = %d",a, b, c);  
}
```

Parameters

- **Value**

Any change in the parameter, inside the function, does not affect the original value of the parameter.

Parameters

```
int a, b, c ;  
int addition (int, int, int);
```

```
int addition(int x, int y, int z)  
{  
    z = x + y;  
    printf("%d + %d = %d", x, y, z);  
    return ();  
}  
void main(void)  
{  
    a = 2;  
    b = 3;  
    c = 25;  
    addition (a, b, c);  
    printf("%d + %d = %d",a, b, c);  
}
```

Parameters

- **Reference**

What is passed to the function is not just the value, but the memory address. The modification inside the function is maintained after exiting the function:

- To pass the address it is necessary to indicate that the parameter is an address:

void foo (int *x); /*Declaration*/

Value=foo(&a); /*Call*/

Parameters

```
int a, b, c ;  
int addition (int, int, int);
```

```
int addition(int x, int y, int z)  
{  
    z = x + y;  
    printf("%d + %d = %d", x, y, z);  
    return ();  
}  
void main(void)  
{  
    a = 2;  
    b = 3;  
    c = 25;  
    addition (a, b, c);  
    printf("%d + %d = %d",a, b, c);  
}
```

What should be changed?

Parameters

```
int a, b, c ;  
int addition (int, int, *int);
```

```
int addition(int x, int y, int *z)  
{  
    *z = x + y;  
    printf("%d + %d = %d", x, y, *z);  
    return ();  
}  
void main(void)  
{  
    a = 2;  
    b = 3;  
    c = 25;  
    addition (a, b, &c);  
    printf("%d + %d = %d",a, b, c);  
}
```

Example (I)

```
#include "stdio.h"
```

```
/* Prototypes */
```

```
int CalculateMax (int n1,int n2,int n3);
```

```
int CalculateMin (int n1,int n2,int n3);
```

```
float CalculateAvg (int n1,int n2,int n3);
```

```
int CalculateMax (int n1,int n2,int n3)
```

```
{
```

```
    if (n1>=n2 && n1>=n3)
```

```
        return (n1);
```

```
    if (n1>=n2 && n1>=n3)
```

```
        return (n2);
```

```
    if (n3>=n2 && n3>=n1)
```

```
        return (n3);
```

```
}
```

Example (II)

```
int CalculateMin (int n1,int n2,int n3)
{
    if (n1<=n2 && n1<=n3)
        return (n1);
    else
        if (n1<=n2 && n1<=n3)
            return (n2);
        else
            return (n3);
}
```

```
float CalculateAvg(int n1,int n2,int n3)
{
    return ((n1+n2+n3)/3);
}
```

Program parameters

```
$ gcc prog.c -o prog
```

```
$ prog one two three four
```

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

```
int main(int argc, char **argv)
```

```
{
```

```
    int i;
```

```
    printf("argc is: %d\n", argc);
```

```
    for(i=0; i<argc; i++)
```

```
        printf("Parameter %d is: %s\n", i, argv[i]);
```

```
}
```

Number of arguments

List of arguments

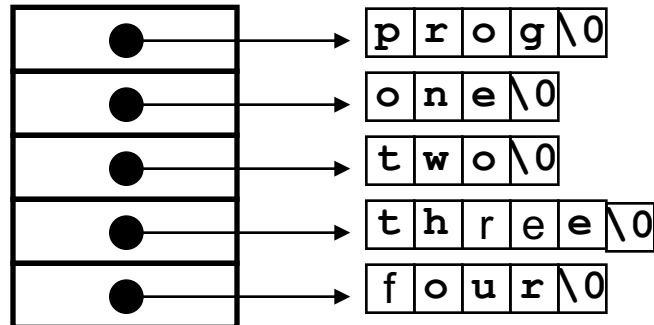
argv[0]

argv[1]

argv[2]

argv[3]

argv[4]



Program parameters

```
$ gcc prog.c -o prog
```

```
$ prog one two three four
```

```
#include <stdio.h>

void printParams (int argc, char **argv)
{
    int i;
    printf("argc is: %d\n", argc);

    for(i=0; i<argc; i++)
        printf("Parameter %d is: %s\n", i, argv[i]);
}

int main(int argc, char **argv)
{
    printParams (argc, argv);
}
```


Calling functions with arrays

- Arrays (and matrices) can only be passed by reference.
- So, it is necessary to pass the address of the first element of the array.

Example

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

```
Void VisualizeArray (int [], int); /* prototype */
```

```
main()
```

```
{
```

```
    int array[5]={1,2,4,6,10}
```

```
    VisualizeArray(&array[0],5);
```

```
}
```

```
void VisualizeArray (int vec[], int len)
```

```
{
```

```
    int i;
```

```
    for (i=0;i<len;i++) printf("%d", vec[i]);
```

```
}
```

Matrix multiplication (I)

File Matrix.h:

```
#define DDimensionX 4
#define DDimensionY 4

typedef float[][] TMatrix;

/* Prototypes */
extern void IntroMatrix(TMatrix M);
extern void ShowMatrix(TMatrix M);
extern void MultiplyMatrix(TMatrix A, TMatrix B, Tmatrix Res);
```

File Matrix.c:

```
void IntroMatrix(TMatrix M)
{
    for (y=0; y<DDimensionY; y++)
        for (x=0; x<DDimensionX; x++)
            fscanf("Introduce data %d,%d:
                    %f.\n", x,y,&M[y][x]);
}
```

Matrix multiplication (II)

File Matrix.c (cont):

```
void ShowMatrix(TMatrix M)
```

```
{  
    for (y=0; y<DDimensionY; y++)  
    {  
        for (x=0; x<DDimensionX; x++)  
            fprintf("\t%f ",M[x][y]); fprintf("\n");  
    }  
}
```

```
void MultiplyMatrix(TMatrix A, TMatrix B, TMatrix Res)
```

```
{  
    for (y=0; y<DDimensionY; y++)  
        for (x=0; x<DDimensionX; x++)  
        {  
            Res[y][x]=0;  
            for (z=0; z<DDimensionZ; z++)  
                Res[y][x]+=A[y][z]*B[z][x];  
        }  
}
```

Matrix multiplication (III)

File MultMatrices.c :

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

```
#include "matrix.h"
```

```
main()
```

```
{
```

```
    printf("Introduce matrix A:\n")
```

```
    IntroMatrix(MatrixA);
```

```
    printf("Introduce matrix B:\n")
```

```
    IntroMatrix(MatrixB);
```

```
    MultiplyMatrix(MatrixA,MatrixB,MatrixRes);
```

```
    printf("Result A * B:\n")
```

```
    ShowMatrix(MatrixRes);
```

```
}
```

Structures in C

struct: a way to compose existing types into a structure

struct complex data structure that contains a set of fields with different types

```
struct [label]  
{  
    type field1;  
    type field2;  
    ...  
};
```

structs define a layout of typed fields

structs can contain other structs

Structures in C

```
struct person
```

```
{
```

```
    char    name[20];
```

```
    int     age;
```

```
    float   weight;
```

```
} you;
```

```
struct persona he={"John Smith",31,80};
```

```
struct persona *she, all[20];
```

Structures in C

```
struct person he, *she, all[20];  
  
printf("His name is %s\n", he.name);  
all[2].age=20;  
she=&all[2];  
printf("Her age is %d\n", she->age);
```

Fields are accessed using '.' notation or -> in case of pointers to the structure.

Pointers

- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- A pointer “points” another variable.

*ptr - a pointer variable
var - a normal variable

// declare a pointer variable ptr of type int

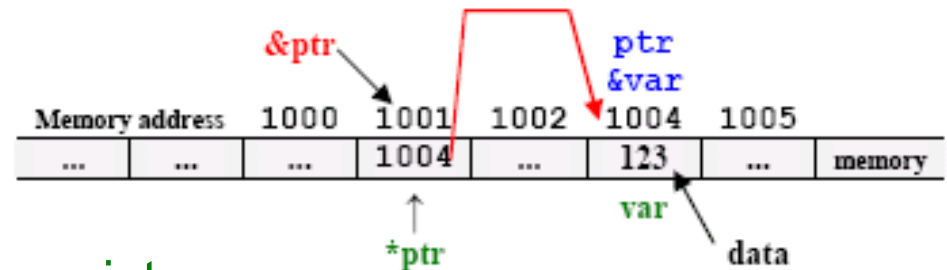
int *ptr;

// declare and initialize normal variable var of type int

int var = 123;

// assign the address of normal variable var to pointer ptr.

ptr = &var;



Pointers

Recall our model for variables stored in memory

What if we had a way to find out the address of a symbol, and a way to reference that memory location by address?

```
address_of(y) == 5  
memory_at[5] == 101
```

```
void f(address_of_char p)  
{  
    memory_at[p] = memory_at[p] - 32;  
}
```

```
char y = 101;    /* y is 101 */  
f(address_of(y)); /* i.e. f(5) */  
/* y is now 101-32 = 69 */
```

Symbol	Addr	Value
	0	
	1	
	2	
	3	
char x	4	'H' (72)
char y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

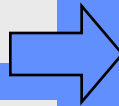
Pointers

This is exactly how “pointers” work.

“address of” or reference operator: &
“memory_at” or dereference operator: *

```
void f(address_of_char p)
{
    memory_at[p] = memory_at[p] - 32;
}
```

```
char y = 101;      /* y is 101 */
f(address_of(y));  /* i.e. f(5) */
/* y is now 101-32 = 69 */
```



A “pointer type”: pointer to char

```
void f(char * p)
{
    *p = *p - 32;
}
```

```
char y = 101;      /* y is 101 */
f(&y);             /* i.e. f(5) */
/* y is now 101-32 = 69 */
```

Pointers are used in C for many other purposes:

- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

Pointers

- ***, &, ->**
 - * returns the content of the variable.
 - & returns the address of the operand.
 - > access to a field of a structure.
- **Operations**
 - Assign (=)**
 - Compare (==, !=)**
 - Inicialize (NULL)**
 - Increment (++), Decrement (--)**

Example (I)

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

```
void main()
```

```
{
```

```
    int n1, n2;
```

```
    int *pn1, *pn2;
```

```
    n1=3;                /* init */
```

```
    n2=10;
```

```
    pn1=pn2=NULL;
```

```
    pn1=&n1;              /* 'pn1' points 'n1' */
```

```
    pn2=&n2;              /* 'pn2' points 'n2' */
```

Example(II)

```
if (pn1!=pn2)
{
    printf("`pn1' and `pn2' points different memory  
positions \n");
    printf("` The content of `pn1' is: %d \n", *pn1);
    printf("` The content of `pn2' is: %d \n", *pn2);
}

pn2=pn1;
if (pn1==pn2)
{
    printf("`pn1' and `pn2' points the same memory  
position \n");
    printf("`The content of `pn1' and `pn2' is: %d  
        \n",*pn1);
}
}
```

Example (III)

```
#include <stdio.h>
void main()
{
    int x, *p;

    x=10;
    *p=x;
}
```

Is it a correct code?

```
#include <stdio.h>
void main()
{
    int x, *p;

    x=10;
    p=x;
    printf("%d", *p);
}
```

What is the value of *p?

Input/Output functions

- In C there is a standard package for performing input and output to files or the terminal.
- To use these facilities, your program must include these definitions by adding the line `#include <stdio.h>` near the start of the program file. If you do not do this, the compiler may complain about undefined functions or datatypes.
- For example:
 - Input: `scanf()`
 - Output: `printf()`

printf()

- The format of the `printf()` function call is:

`printf(format, exp1, exp2, exp3, ..., expn);`

where:

- *format* : string of the data output format
- *exp_i* : Expression to include inside the format

printf()

Example:

```
int a=3;
```

```
float x=23.0;
```

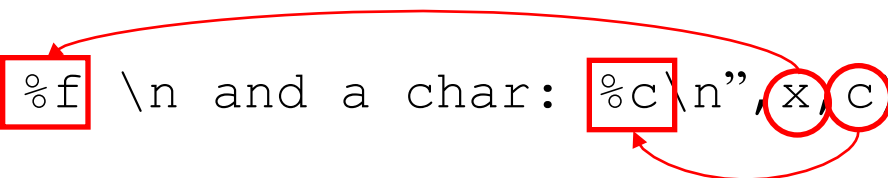
```
char c='A';
```

```
printf("Hello World!!\n");
```

```
printf("An integer %d\n", a);
```



```
printf("A real: %f \n and a char: %c\n", x, c);
```



printf()

Formato	Expresión	Resultado
%d %i	integer	integer with sign
%u	integer	integer without sign
%x %X	integer	hexadecimal integer without sign
%f	real	real
%e %E	real	real in scientific notation
%c	character	character
%p	pointer	memory direction
%s	string	string
%ld %lu ...	large integer	large integer

scanf ()

- The format of the function scanf () is:

`scanf (format, dir1, dir2, dir3, ..., dirn);`

where:

- *format* : string with the data input format
- *dir*_{*i*} : memory direction where the result is stored

scanf()

Example

```
int a,*pa;
```

```
float x;
```

```
char c;
```

```
scanf("%d",&a); /* Read an integer and  
store it in a */
```

```
scanf("%f %c",&x,&c); /* Read x and c */
```

```
scanf("%d",pa); /* Dangerous */
```

```
pa=&a; scanf("%d",&a); /* OK. Read a */
```

scanf () Reading strings

Example:

```
char *pc;
```

```
char str[82];
```

```
scanf ("%s", pc);    /* Dangerous    */
```

```
scanf ("%s", str);    /* Read till space or  
    line end */
```

Reading characters

- getchar - returns the next character of keyboard input as an int
- As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

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

```
main()
```

```
{
```

```
    int ch, i = 0;
```

```
    while ((ch = getchar ()) != EOF)
```

```
        i ++;
```

```
    printf("%d\n", i);
```

```
}
```

Writing characters

- putchar puts its character argument on the standard output (usually the screen).
- The following example program converts any typed input into capital letters.

```
#include <ctype.h> /* For definition of toupper */
#include <stdio.h> /* For definition of getchar, putchar, EOF */
main()
{
    int ch;
    while((ch = getchar()) != EOF)
        putchar(toupper(ch));
}
```


Reading/writing lines

- **gets** reads a whole line of input into a string until a newline or EOF is encountered. The string should be large enough to hold any expected input lines.
- **puts** writes a string to the output, and follows it with a newline character.
- Program which uses gets and puts to double space typed input.

```
#include <stdio.h>
main ()
{
    char line[256]; /* Define string to store a line of input */
    while (gets(line) != NULL) /* Read line */
    {
        puts (line); /* Print line */
        printf ("\n"); /* Print blank line */
    }
}
```

Handling Files in C

- C File Handling - File Pointers
- Opening a file pointer using fopen
- Closing a file using fclose
- Reading and writing using file pointers

Handling Files in C

- Linux has a facility called redirection which allows a program to access a single input file and a single output file very easily.
- The program is written to read from the keyboard and write to the terminal screen as normal.
- To run prog1 but read data from file infile instead of the keyboard, you would type `prog1 < infile`
- To run prog1 and write data to outfile instead of the screen, you would type `prog1 > outfile`
- Both can also be combined as in `prog1 < infile > outfile`
- Redirection is simple, and allows a single program to read or write data to or from files or the screen and keyboard.

Handling Files in C

- C communicates with files using a new datatype called a file pointer.
- This type is defined within `stdio.h`, and written as `FILE *`.
- A file pointer called `output_file` is declared in a statement like

```
FILE *output_file;
```

Handling Files in C

- **Opening a file pointer using fopen**
- Your program must open a file before it can access it.
- This is done using the fopen function, which returns the required file pointer. If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows

```
if ((output_file = fopen("output_file", "w")) == NULL)
    fprintf(stderr, "Cannot open %s\n", "output_file");
```

fopen takes two arguments, both are strings, the first is the name of the file to be opened, the second is an access character, which is usually one of:

"r"	Open file for reading
"w"	Create file for writing
"a"	Open file for appending

Handling Files in C

- **Closing a file using fclose**
- The fclose command can be used to disconnect a file pointer from a file.
- This is usually done so that the pointer can be used to access a different file.
- Systems have a limit on the number of files which can be open simultaneously, so it is a good idea to close a file when you have finished using it.
- This would be done using a statement like

```
fclose(output_file);
```

Handling Files in C

- `fscanf ()` - puts formatted data into a string which must have sufficient space allocated to hold it. This can be done by declaring it as an array of `char`. The data is formatted according to a control string of the same form as that for `printf`.
- `char *fgets (char *s, int n, FILE *file);`
- `int fgetc (FILE *file);`
- `size_t fread (void *ptr, size_t size, size_t nmemb, FILE *file);`
- EOF is a character which indicates the end of a file.
- NULL is a character or pointer value. Where it is used as a character, NULL is commonly written as `'\0'`. It is the string termination character which is automatically appended to any strings in your C program.

Handling Files in C

- Writing one character or one byte
fputc (char data, FILE *file);
- Write strings
fputs(char *string, int numchar, FILE *file);
- Write with format
fprintf (FILE *file, char *format, arguments..);
- Structures
fwrite (void *buffer, int size, int noRepet, FILE *file);