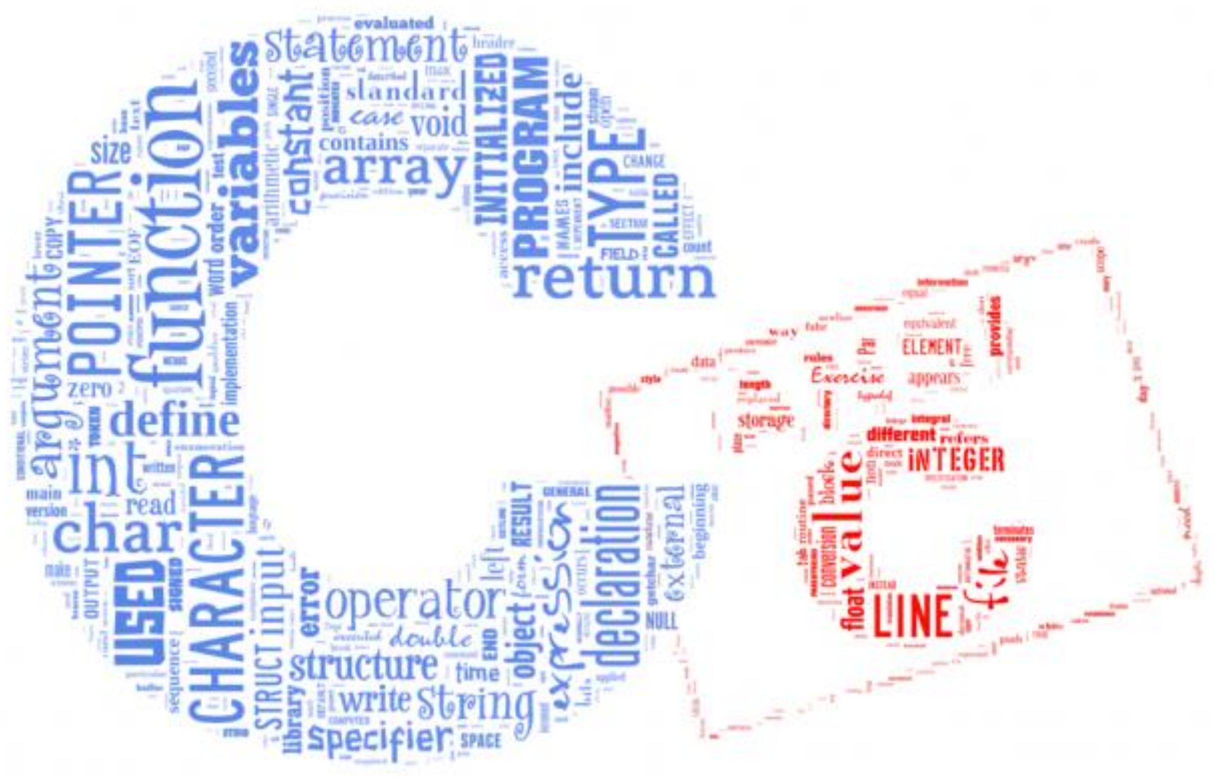
BIRZEIT UNIVERSITY

Faculty of Engineering and Technology كلية الهندسة والتكنولوجيا

Computer Science Department دائرة علم الحاسوب

COMP142/ COMP133

Computer and Programming (Part 2)



Compiled and Prepared by:

Dr. Majdi M. Mafarja

Recursive Programming

Introduction

When we write a method for solving a particular problem, one of the basic design techniques is to break the task into smaller subtasks. For example, the problem of adding (or multiplying) n consecutive integers can be reduced to a problem of adding (or multiplying) n-1consecutive integers:

```
1 + 2 + 3 +... + n = n + [1 + 2 + 3 + .. + (n-1)]

1 * 2 * 3 *... * n = n * [1 * 2 * 3 * .. * (n-1)]
```

Therefore, if we introduce a method `sumR(n)` (or `timesR(n)`) that adds (or multiplies) integers from 1 to n, then the above arithmetics can be rewritten as

```
sumR(n) = n + sumR(n-1)

timesR(n) = n * timesR(n-1)
```

Such functional definition is called a **recursive** definition, since the definition contains a call to itself. On each recursive call the argument of `sumR(n) (or timesR(n))` gets smaller by one. It takes n-1 calls until we reach the **base case -** this is a part of a definition that does not make a call to itself. Each recursive definition requires base cases in order to prevent infinite recursion.

In the following example we provide iterative and recursive implementations for the addition and multiplication of n natural numbers.

```
int sum(int n)                    int sumR(int n)
{                                        {
   int res = 0;                            if(n == 1)
   for(int i = 1; i = n; i++)                 return 1;
      res = res + i;                       else
                                              return n + sumR(n-1);
   return res;                           }
}
```

To solve a problem recursively means that you have to first redefine the problem in terms of a smaller sub-problem of the same type as the original problem. In the above summation problem, to *sum-up n* integers we have to know how to *sum-up n-1* integers. Next, you have to figure out how the solution to smaller sub-problems will give you a solution to the problem as a whole. This step is often called as a *recursive leap of faith*. Before using a recursive call, you must be convinced that the recursive call will do what it is supposed to do. You do not need to think how recursive calls works, just assume that it returns the correct result.

Examples of a recursive function:

1. Compute factorial of a number

Example : Let us consider the Factorial Function

n! = n * (n-1) * (n-2) * … * 2 * 1

0! = 1

Iterative solution:

```
int fact(int n)
{
      int p, j;
      p = 1;
      for ( j=n; j>=1; j--)
            p = p* j;
      return ( p );
}
```

Recursive definition:

In the recursive implementation there is no loop. We make use of an important mathematical property of factorials. Each factorial is related to factorial of the next smaller integer:

n! = n * (n-1)!

To make sure the process stops at some point, we define 0! to be 1. Thus the conventional mathematical definition looks like this:

```
n! = 1           if n = 0
n! = n*(n-1)!    if n > 0
```

This definition is recursive, because it defines the factorial of n in terms of factorial of n – 1. The new problem has the same form, which is, now find factorial of n – 1 .

```
int fact(int n)
{
      if (n ==0)
            return (1);
      else
      return (n * fact(n-1));
}
```

The Nature of Recursion

1) One or more simple cases of the problem (called the *stopping cases*) have a simple non-recursive solution.

2) The other cases of the problem can be reduced (*using recursion*) to problems that are closer to stopping cases.

3) Eventually the problem can be reduced to stopping cases only, which are relatively easy to solve.

In general:

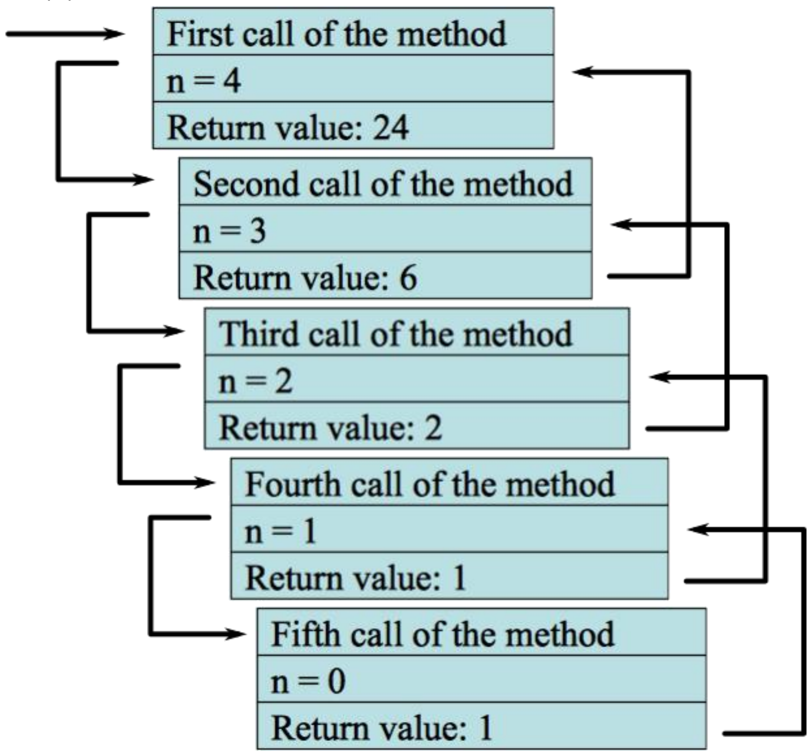
```
if (stopping case)
      solve it
else
      reduce the problem using recursion
```

Tracing a Recursive Function

Let us try to follow the logic the computer uses to evaluate any function call. It uses a stack to keep track of function calls. Whenever a new function is called, all its parameters and local variables are pushed onto the stack along with the memory address of the calling statement (this gives the computer the return point after execution of the function)

In the factorial example, suppose "main" has a statement

f= factorial (4);



2. Find sum of squares of a series.

Here we are interested in evaluating the sum of the series

$m^2 + (m + 1)^2 + (m + 2)^2 + \ldots\ldots\ldots + (n)^2$

We can compute the sum recursively, if we break up the sum in two parts as shown below:

$m^2 + [(m + 1)^2 + (m + 2)^2 + \ldots\ldots\ldots + (n)^2]$

Note that the terms inside the square brackets computes the sum of the terms from m+1 to n. Thus we can write recursively sumsq(m, n) = m^2 + sumsq(m+1, n)

The sum of the terms inside the square brackets can again be computed in similar manner by simply replacing m with m+1. The process can be continued till m reaches the value of n. Then sumsq(n, n) is simply (n)2

Here is the recursive function:

```
int  sumsq ( int m, int n) {
     if (m ==n )
          return n *n;
     else
          return ( m * m + sumsq(m+1, n);
 }
```

Trace the above recursive function to find sumsq(2,5).

```
sumsq(2, 5) = m2  + sumsq (3,5)
          = 4 +     sumsq(3,5)
          = 4 +     9 +     sumsq(4,5)
          = 13 +        16 +     sumsq(5,5)
          = 29 +                           25
          = 54
```

3. Consider the following recursive function:

```
int speed (int N)
{
   if (N == 2) return 5;
   if (N % 2 == 0)
          return (1 + speed(N/2));

   else
          return (2+speed(3 + N));
}
```

Trace the function for N= 7.

```
Speed(7) = 2 + speed(10)
             =  2 + 1 + speed(5)
             =  3 + 2 + speed(8)
             =  5 + 1 + speed (4)
             =  6 + 1 + speed (2)
             =  7 + 5
             = 12
```

4. Consider the following recursive function

```
int value(int a, int b) {

     if (a <= 0)
          return 1;
     else
          return (b*value(a-1,b+1));
}
```

Let us trace the calls

```
a) value(1, 5)
=  5 * value( 0, 6)
=  5 * 1
= 5

b) value(3, 3)
= 3 * value(2, 4)
= 3 * 4 * value(1, 5)
= 3 * 4 * 5 * value( 0, 6)
= 3 * 4 * 5 * 1
= 60
```

5. To print a user-entered string in reverse order

Here is a recursive function that reads the characters of a string from the keyboard, as they are being typed, but prints them out in the reverse order. The function needs to know how many characters would be read before it starts printing. Obviously, printing cannot start until all the characters have been read. The function uses an internal stack of the computer to store each character as it is being read.

```
void print_reverse(int n)
{
     char next;

     if (n == 1) {        /* stopping case */
          scanf("%c",&next);
          printf("%c", next);
     }
     else {
          scanf("%c", &next);
          print_reverse(n-1);
          printf("%c",next);
     }
     return;
}
```

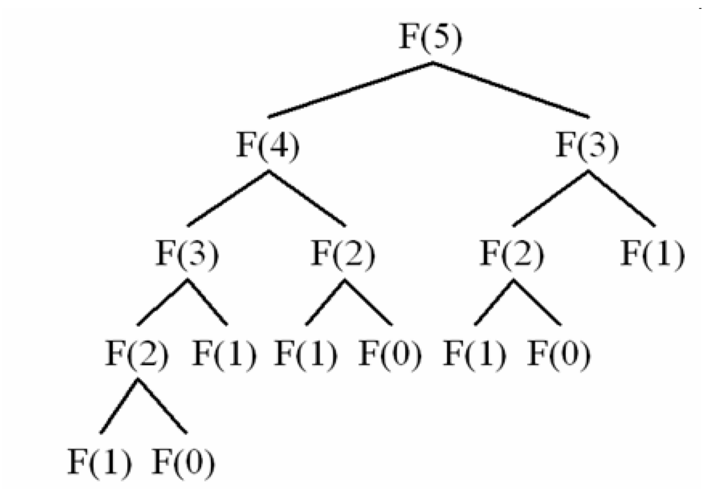
6. Fibonacci Numbers

The Fibonacci number is defined as the sum of the two preceding numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

This recursive definition translates directly into code

```
public int fibonacci(int n)
{
   if (n <= 0) return 0;
   else if (n == 1)
       return 1
   else return
       fibonacci(n-1) + fibonacci(n-2);
}
```



This is a binary tree of recursive calls for fibonacci(5). The picture shows that the tree for fibonacci(5) has 5 levels, and thus, the total number of nodes is about 2^5.

Chapter 7

7 Structures

7.1 Structures

A structure is a customised user-defined data type in C. It is by definition a collection of variables of any type that are referenced under one name, providing a convenient means of keeping related information together.

Some terminology :-

structure definition :- the template used to create structure variables.
structure elements :- the member variables of the structure type

Defining Structures

Syntax :

```
struct  tag   {
type  var_1 ;
type var_2 ;
...
     type var_n ;
}  ;
```

The keyword **struct** tells the compiler we are dealing with a structure and must be present whenever we refer to the new type, **tag** is an identifier which is the name given to the customised "type".

A variable of this new type can now be defined as follows for example. Note that the keyword struct has to be used in conjunction with our own name for the structure, *tag*.

struct tag variable ;

For Example :-

```
     struct RECORD {
          int rec_no ;
          char name[30] ;
          char town[40] ;
          char country[ 20 ]
          } ;
     struct RECORD person ;
```

The compiler will automatically allocate enough storage to accommodate all the elements. To find out how much storage is required one might do the following

size = sizeof (person) ;

or

```
size = sizeof( struct RECORD ) ;
```

NB : The name of a structure is not the address of the structure as with array names.

Accessing Structure Elements

For example define a complex type structure as follows.

```
struct complex {
        double real ;
        double imaginary ;          // Note that a variable may also be
        } cplx ;                            // defined at structure definition time
```

The elements of the structure are accessed using the **dot operator**, **.** , as follows

```
cplx.real  =  10.0 ;
cplx.imag  =  20.23 ;
scanf ( "%lf", &cplx.real ) ;
```

or if we want to access struct RECORD already defined

```
puts( person.name ) ;
```

or character by character

```
person.name[i] = 'a' ;
```

Thus we treat structure elements exactly as normal variables and view the dot operator as just another appendage like the indirection operator or an array index.

Initialising Structures

Structure elements or fields can be initialised to specific values as follows :-

```
struct id {
        char name[30] ;
        int id_no ;
        } ;
struct id student = { "John", 4563 } ;
```

Structure Assignment

The name of a structure variable can be used on its own to reference the complete structure. So instead of having to assign all structure element values separately, a single assignment statement may be used to assign the values of one structure to another structure of the same type.

For Example :-

```
struct {
int a, b ;
        }  x = {1, 2 }, y ;

y = x ;            // assigns values of all fields in x to fields in y
```

Creating more Complex Structures with Structures

Once again emphasising that structures are just like any other type in C we can create arrays of structures, nest structures, pass structures as arguments to functions, etc.

For example we can nest structures as follows creating a structure employee_log that has another structure as one of its members.

```
struct time {
        int hour ;
        int min ;
        int sec ;
                } ;
struct employee_log {
        char name[30] ;
        struct time start, finish ;
        } employee_1 ;
```

To access the hour field of time in the variable employee_1 just apply the dot operator twice

```
employee_1.start.hour = 9 ;
```

Typically a company will need to keep track of more than one employee so that an array of *employee_log* would be useful.

```
        struct employee_log  workers[100] ;
```

To access specific employees we simply index using square braces as normal, e.g. workers[10]. To access specific members of this structure we simply apply the dot operator on top of the index.

```
        workers[10].finish.hour = 10 ;
```

When structures or arrays of structures are not global they must be passed to functions as parameters subject to the usual rules. For example

```
function1( employee_1 ) ;
```

implements a call to function1 which might be prototyped as follows

```
void  function1( struct employee_log emp ) ;
```

Note however that a full local copy of the structure passed is made so if a large structure is involved memory the overhead to simply copy the parameter will be high so we should employ call by reference instead as we will see in the next section.

Passing an array of structures to a function also follows the normal rules but note that in this case as it is impossible to pass an array by value no heavy initialisation penalty is paid - we essentially have call by reference. For example

```
        function2( workers ) ;
```

passes an array of structures to *function2* where the function is prototyped as follows.

```
        function2( struct employee_log staff[ ] ) ;
```

Structure Pointers

As we have said already we need call by reference calls which are much more efficient than normal call by value calls when passing structures as parameters. This applies even if we do not intend the function to change the structure argument.

A structure pointer is declared in the same way as any pointer for example

```
        struct address {
                char name[20] ;
                char street[20] ;
                } ;
        struct address person ;
        struct address *addr_ptr ;
```

declares a pointer addr_ptr to data type *struct address*.

To point to the variable person declared above we simply write

```
        addr_ptr =  &person ;
```

which assigns the address of person to addr_ptr.

To access the elements using a pointer we need a new operator called the arrow operator, ->, which can be used **only** with structure pointers. For example

```
        puts( addr_ptr -> name ) ;
```

For Example :- Program using a structure to store time values.

```
 #include <stdio.h>

 struct time_var {
        int hours, minutes, seconds ;
        } ;
 void display ( const struct time_var * ) ;                /* note structure pointer and const */

 void main()
 {
 struct time_var time ;

 time.hours = 12 ;
 time.minutes = 0 ;
 time.seconds = 0 ;
 display( &time ) ;
 }
 void display( const struct time_var *t )
 {
 printf( "%2d:%2d;%2d\n", t -> hours, t -> minutes, t -> seconds ) ;
 }
```

Note that even though we are not changing any values in the structure variable we still employ call by reference for speed and efficiency. To clarify this situation the *const* keyword has been employed.

7.2 Dynamic allocation of structures

The memory allocation functions may also be used to allocate memory for user defined types such as structures. All malloc() basically needs to know is how much memory to reserve.

For Example :-

```
                struct coordinate {
                int x, y, z ;
                } ;
        struct coordinate *ptr ;

        ptr = (struct coordinate * ) malloc( sizeof ( struct coordinate )  ) ;
```

7.3 Enumerations

An enumeration is a user defined data type whose values consist of a set of named integer constants, and are used for the sole purpose of making program code more readable.

Syntax: **enum tag { value_list } [enum_var] ;**

where tag is the name of the enumeration type, value_list is a list of valid values for the enumeration, and where enum_var is an actual variable of this type.

For Example :-

```
        enum colours { red, green, blue, orange } shade ;
                                //  values red - 0, green - 1, blue - 2, orange - 3

        enum day { sun = 1, mon, tue, wed = 21, thur, fri, sat } ;
```

```
    enum day weekday ;
                                // values are 1, 2, 3, 21, 22, 23, 24
```

Variables declared as enumerated types are treated exactly as normal variables in use and are converted to integers in any expressions in which they are used.

For Example :-

```
    int i ;

    shade = red ;   // assign a value to shade enum variable

    i = shade ;     // assign value of enum to an int

    shade = 3 ;     // assign valid int to an enum, treat with care
```

7.4 The typedef Keyword

C makes use of the **typedef** keyword to allow new data type **names** to be defined. No new type is created, an existing type will now simply be recognised by another name as well. The existing type can be one of the in-built types or a user-defined type.

Syntax : **typedef type name ;**

where type is any C data type and name is the new name for this type.

For Example :-

```
    typedef  int INTEGER ;
    INTEGER i ;                 // can now declare a variable of type 'INTEGER'

    typedef double * double_ptr ;
    double_ptr  ptr ;                   //  no need of * here as it is part of the type

    typedef struct coords {
            int x, y ;
            } xycoord ;         //  xycoord is now a type name in C
    xycoord  coord_var ;
```

The use of typedef makes program code easier to read and when used intelligently can facilitate the porting of code to a different platform and the modification of code. For example in a first attempt at a particular program we might decide that floating point variables will fill our needs. At a later date we decide that all floating point variables really should be of type double so we have to change them all. This problem is trivial if we had used a typedef as follows :-

```
    typedef float FLOATING ;
```

To remedy the situation we modify the user defined type as follows

```
        typedef double FLOATING ;
```

7.5 Dynamic Memory Allocation

This is the means by which a program can obtain and release memory at run-time. This is very important in the case of programs which use large data items e.g. databases which may need to allocate variable amounts of memory or which might have finished with a particular data block and want to release the memory used to store it for other uses.

The functions **malloc()** and **free()** form the core of C's dynamic memory allocation and are prototyped in <malloc.h>. malloc() allocates memory from the heap i.e. unused memory while available and free() releases memory back to the heap.

The following is the prototype for the malloc() function

```
        void * malloc( size_t num_bytes ) ;
```

malloc() allocates num_bytes bytes of storage and returns a pointer to type void to the block of memory if successful, which can be cast to whatever type is required. If malloc() is unable to allocate the requested amount of memory it returns a NULL pointer.

For example to allocate memory for 100 characters we might do the following

```
 #include <malloc.h>

 void main()
 {
 char *p ;

 if (  !( p = malloc( sizeof( char  ) * 100  ) )
        {
        puts( "Out of memory" ) ;
        exit(1) ;
        }
 }
```

The return type void * is automatically cast to the type of the lvalue type but to make it more explicit we would do the following

```
 if (  !( (char * )p = malloc( sizeof( char  ) * 100  ) )
        {
        puts( "Out of memory" ) ;
        exit(1) ;
        }
```

To free the block of memory allocated we do the following

```
free ( p ) ;
```

Note :- There are a number of memory allocation functions included in the standard library including calloc( ), _fmalloc( ) etc. Care must be taken to ensure that memory allocated with a particular allocation function is released with its appropriate deallocation function, e.g. memory allocated with malloc() is freed only with free() .

7.6 Exercises

1. Write a program which will simulate the action of a digital clock continuously as closely as possible. Use a structure to hold the values of hours, minutes and seconds required. Your program should include a function to display the current time and to update the time appropriately using a delay function/loop to simulate real time reasonably well. Pointers to the clock structure should be passed to the display and update functions rather than using global variables.

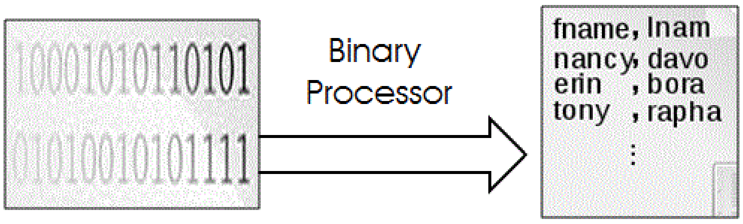
2. Complex numbers are not supported as a distinct type in C but are usually implemented by individual users as user defined structures with a set of functions representing the operations possible on them. Define a data type for complex values and provide functions to support addition, subtraction, multiplication and division of these numbers. Use call by reference to make these functions as efficient as possible and try to make their use as intuitive as you can.

3. Write a program which will

a) Set up an array of structures, which will hold the names, dates of birth, and addresses of a number of employees.
b) Assign values to the structure from the keyboard.
c) Sort the structures into alphabetical order by name using a simple bubble sort algorithm.
d) Print out the contents of a specific array element on demand.

8 Binary Files

As shown in the figure below, binary file is stored in Binary Format (in 0/1). This Binary file is difficult to read for humans. So generally Binary file is given as input to the Binary file Processor. Processor will convert binary file into equivalent readable file.



Many programs produce output files that are used as input files for other programs. If there is no need for a human to read the file, it is a waste of computer time for the first program to convert its internal data format to a stream of characters, and then for the second program to have to apply an inverse conversion to extract the intended data from the stream of characters. We can avoid this unnecessary translation by using a binary file rather than a text file.

A **binary file** is created by executing a program that stores directly in the file the computer's internal representation of each file component.
For example, the following code fragment creates a binary file named "nums.bin" , which contains the even integers from 2 to 500.

FIGURE 11.3 Creating a Binary File of Integers

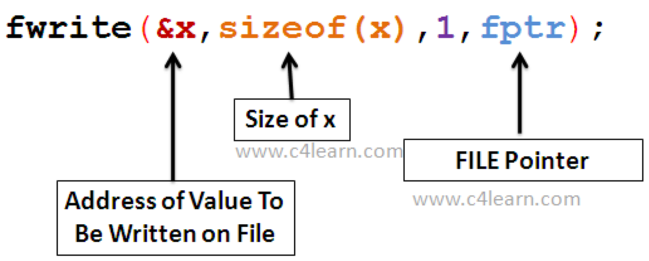
```
FILE *binaryp;
int   i;

binaryp = fopen("nums.bin", "wb");

for  (i = 2; i <= 500; i += 2)
     fwrite(&i, sizeof (int), 1, binaryp);

fclose(binaryp);
```

"wb" (write binary) for output files
"rb" (read binary) for input files.


fwrite(&x,sizeof(x),1,fptr);
Size of x
www.c4learn.com
FILE Pointer
www.c4learn.com
Address of Value To Be Written on File

8.1 Function fwrite requires four arguments

1. Address of value to be written to a memory.
2. Size of one value.
3. Maximum number of elements to be written to the binary file.

4. File pointer to a binary file opened in mode "wb" using function fopen.

```
fwrite(&i, sizeof (int), 1, binaryp);
```
will write one integer to the binary file

```
fwrite(score, sizeof (int), 10, binaryp);
```
will write an array of 10 integers to a binary file.

8.2 Function fread also requires four arguments

1. Address of first memory cell to fill.
2. Size of one value.
3. Maximum number of elements to copy from the file into memory.
4. File pointer to a binary file opened in mode "rb" using function fopen .

It is very important not to mix file types. A binary file created (written) using fwrite must be read using fread . A text file created using fprintf must be read using a text file input function such as fscanf .

```
#define STRSIZ 10
#define MAX 40
typedef struct {
     char name[STRSIZ];
     double diameter; /* equatorial diameter in km */
     int moons; /* number of moons */
     double orbit_time, /* years to orbit sun once */
     rotation_time; /* hours to complete one
revolution on axis */
} planet_t;
. . .
double nums[MAX], data;
planet_t a_planet;
int i, n, status;
FILE *plan_bin_inp, *plan_bin_outp, *plan_txt_inp,
*plan_txt_outp;
FILE     *doub_bin_inp,     *doub_bin_outp,     *doub_txt_inp,
*doub_txt_outp;
```

TABLE 11.5 Data I/O Using Text and Binary Files

Example	Text File I/O	Binary File I/O	Purpose

1	`plan_txt_inp =`
	`    fopen("planets.txt", "r");`
	
	`doub_txt_inp =`
	`    fopen("nums.txt", "r");`
		`plan_bin_inp =`
		`    fopen("planets.bin", "rb");`
		
		`doub_bin_inp =`
		`    fopen("nums.bin", "rb");`
			Open for input a file of planets and a file of numbers, saving file pointers for use in calls to input functions.

2	`plan_txt_outp =`
	`    fopen("pl_out.txt", "w");`
	
	`doub_txt_outp =`
	`    fopen("nm_out.txt", "w");`
		`plan_bin_outp =`
		`    fopen("pl_out.bin", "wb");`
		
		`doub_bin_outp =`
		`    fopen("nm_out.bin", "wb");`
			Open for output a file of planets and a file of numbers, saving file pointers for use in calls to output functions.

3	`fscanf(plan_txt_inp,`
	`    "%s%lf%d%lf%lf",`
	`    a_planet.name,`
	`    &a_planet.diameter,`
	`    &a_planet.moons,`
	`    &a_planet.orbit_time,`
	`    &a_planet.rotation_time);`
		`fread(&a_planet,`
		`      sizeof (planet_t),`
		`      1, plan_bin_inp);`
			Copy one planet structure into memory from the data file.

4	`fprintf(plan_txt_outp,`
	`    "%s %e %d %e %e",`
	`    a_planet.name,`
	`    a_planet.diameter,`
	`    a_planet.moons,`
	`    a_planet.orbit_time,`
	`    a_planet.rotation_time);`
		`fwrite(&a_planet,`
		`       sizeof (planet_t),`
		`       1, plan_bin_outp);`
			Write one planet structure to the output file.

5	`for  (i = 0; i < MAX; ++i)`
	`    fscanf(doub_txt_inp,`
	`           "%lf", &nums[i]);`
		`fread(nums, sizeof (double),`
		`      MAX, doub_bin_inp);`
			Fill array `nums` with type `double` values from input file.

6	`for (i = 0; i < MAX; ++i)`
	`    fprintf(doub_txt_outp,`
	`            "%e\n", nums[i]);`
		`fwrite(nums, sizeof (double),`
		`       MAX, doub_bin_outp);`
			Write contents of array `nums` to output file.

7	`n = 0;`
	`for (status =`
	`       fscanf(doub_txt_inp,`
	`       "%lf", &data);`
	`     status != EOF &&`
	`       n < MAX;`
	`     status =`
	`       fscanf(doub_txt_inp,`
	`       "%lf", &data))`
	`     nums[n++] = data;`
		`n = fread(nums,`
		`          sizeof (double),`
		`          MAX, doub_bin_inp);`
			Fill `nums` with data until `EOF` encountered, setting `n` to the number of values stored.

8	`fclose(plan_txt_inp);`
	`fclose(plan_txt_outp);`
	`fclose(doub_txt_inp);`
	`fclose(doub_txt_outp);`
		`fclose(plan_bin_inp);`
		`fclose(plan_bin_outp);`
		`fclose(doub_bin_inp);`
		`fclose(doub_bin_outp);`
			Close all input and output files.

8.3 Example

```
struct student
{
     char name[50];
     int roll;
};
main()
{
     FILE *fptr;
     struct student st[20];
     int num;
     if((fptr = fopen("ip.txt","wb+"))==NULL)
     {
          printf("nError in Opening File");
          exit(0);
     }
     printf("How many Students : ");
     scanf("%d",&num);

     for(i=0;i<num;i++)
     {
          printf("nEnter the Name and Roll Number");
          scnaf("%s %d",st.name,&st.roll);
          fwrite(&st,sizeof(st),1,fptr);
     }
     //Structure is Written on File
     getch();
}
```