

Dynamic Memory Allocation

Introduction

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do always. Our initial judgement of size, if it is wrong, may cause failure of the program or wastage of memory space.

So it is required to allocate memory at run time.

Definition – The process of allocating memory at run time is known as *dynamic memory allocation*.

Dynamic Memory Allocation

C language supports 4 library functions known as “memory management functions” that can be used for allocating and freeing memory during program execution.

The functions are listed below.

1. malloc()
2. calloc()
3. free()
4. realloc()

Allocating a Block of Memory: malloc()

A block of memory may be allocated using the function **malloc**. The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void**. This means that we can assign it to any type of pointer. The general syntax is of the form

ptr = (cast-type *) malloc (byte - size);

ptr is a pointer of type ***cast-type***. The malloc returns a pointer(***of cast-type***) to an area of memory with size ***byte-size***.

Example: -

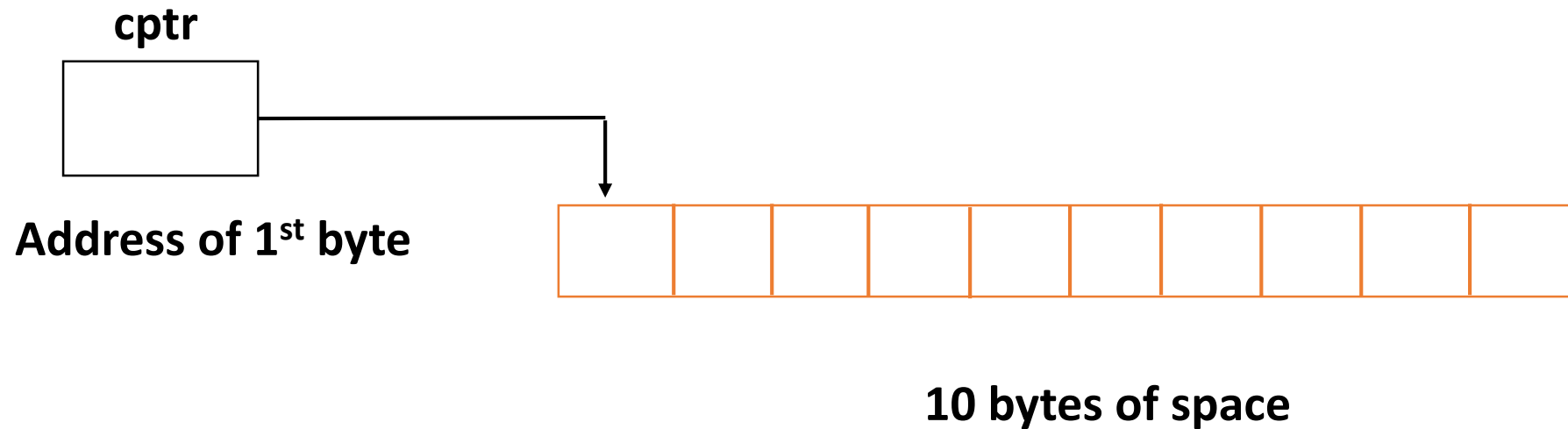
x = (int *) malloc (100 * sizeof (int));

On successful execution of this statement, a memory space equivalent to “ 100 times the size of an int” bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of int.

Example:

```
char *cptr;
```

```
cptr = (char*) malloc(10);
```



Note – For using of these memory management functions you have to include the header file `stdlib.h`

```
#include<stdio.h>
#include<stdlib.h>
vaoid main()
{
    int n,i;
    float *p;
    printf("Enter how many numbers you want to give as input?");
    scanf("%d", &n);
    p = (float *) malloc( n * sizeof(float));
    for (i =0; i < n-1 ; i++)
    {
        scanf("%d", p+i);
    }
    for(i=0; i<n-1; i++)
    {
        printf("%d", *(p+i));
    }
}
```

Example of malloc

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

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

```
void main()
```

```
{
```

```
    int *p, *table;
```

```
    int size;
```

```
    printf("Enter how many numbers you want to enter?");
```

```
    scanf("%d", &size);
```

```
    table = (int *) malloc(size * sizeof(int));
```

```
    for(p = table; p < table + size; p++)
```

```
    {
```

```
        scanf("%d", p);
```

```
    }
```

```
    for(p = table; p < table + size; p++)
```

```
    {
```

```
        printf("%d", *p);
```

```
    }
```

```
}
```

Allocating Multiple blocks of Memory: calloc()

While **malloc** allocates a **single block** of storage space, **calloc** allocates **multiple blocks of storage**, each of the **same size**, and then sets all bytes to **zero(0)**. The general form of **calloc** is:

```
ptr = (cast-type *) calloc (n, elem-size);
```

The above statement allocates contiguous space for ***n blocks***, each of size ***elem-size bytes***.

All bytes are **initialised to zero(0)** and a pointer to the **first byte** of the allocated region is returned.

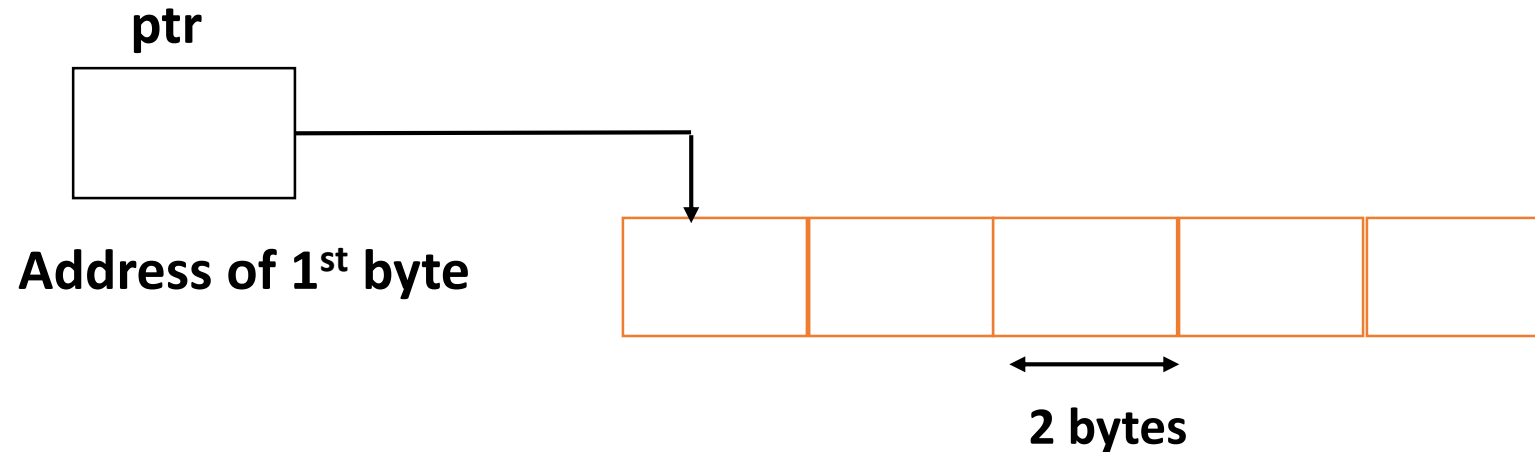
If there is not enough space, a **NULL pointer** is returned.

Example

```
int *ptr;
```

```
ptr = (int *) calloc (5, sizeof(int));
```

On successful execution of the above statement, the allocated memory will be:



```
// Program to calculate the sum of n  
numbers entered by the user
```

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

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

```
void main()
```

```
{
```

```
    int n, i, *ptr, sum = 0;
```

```
    printf("Enter number of elements: ");
```

```
    scanf("%d", &n);
```

```
    ptr = (int*) calloc(n, sizeof(int));
```

```
    printf("Enter elements: ");
```

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

```
    {
```

```
        scanf("%d", ptr + i);
```

```
    }
```

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

```
        {
```

```
            sum = sum + *(ptr + i);
```

```
        }
```

```
        printf("Sum = %d", sum);
```

```
    }
```

Releasing the used space: free()

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the **free** function.

Syntax: free(ptr);

Where **ptr** is a pointer to a memory block, which has already been created by **malloc** or **calloc**.

```
// Program to calculate the sum of n  
numbers entered by the user
```

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

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

```
void main()
```

```
{
```

```
    int n, i, *ptr, sum = 0;
```

```
    printf("Enter number of elements: ");
```

```
    scanf("%d", &n);
```

```
    ptr = (int*) calloc(n, sizeof(int));
```

```
    printf("Enter elements: ");
```

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

```
    {
```

```
        scanf("%d", ptr + i);
```

```
    }
```

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

```
        {
```

```
            sum = sum + *(ptr + i);
```

```
        }
```

```
    printf("Sum = %d", sum);
```

```
    free(ptr);
```

```
}
```

Altering the size of a block: realloc()

It is likely that we discover later, the **previously allocated memory is not sufficient** and we need **additional space** for more elements. It is also possible that the memory allocated is **much larger than** necessary and we want to **reduce** it.

In both the cases, we can change the memory size already allocated with the help of the function **realloc**.

This process is called the **reallocation of memory**.

Syntax:

```
ptr = realloc(ptr, x);
```

Here **ptr** is reallocated with a new size **x**

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);
    ptr = (int*) malloc(n1 * sizeof(int));
    for(i = 0; i < n1; ++i)
        scanf("%d", ptr + i);
```

```
printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    for(i = 0; i < n2; ++i)
        printf("%d", ptr + i);
    free(ptr);
}
```

Computer Graphics using C

Introduction

Consider the sample program

```
#include<graphics.h> /* header file */
#include<conio.h>
void main()
{
    /* the following two lines are the syntax for writing a particular
    program in graphics. It's explanation is given after the program.*/

    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\\\TURBOC3\\\\BGI");

    setbkcolor (GREEN);
    getch();
    closegraph();
}
```


What are initgraph, gd and gm?

- gd = graphdriver;
- gm = graphmode;

Syntax for initgraph:

```
void initgraph (int *graphdriver, int *graphmode, char *pathtodriver) ;
```

Description of initgraph()

- It initializes the graphics system by loading the passed graphics driver then changing the system into graphics mode. It also resets or initializes all graphics settings like color, palette, current position etc, to their default values. Below is the description of input parameters of initgraph function.
- **graphicsDriver** : It is a pointer to an integer specifying the graphics driver to be used. It tells the compiler that what graphics driver to use or to automatically detect the drive. In all our programs we will use **DETECT** macro of graphics.h library that instruct compiler for auto detection of graphics driver.
- **graphicsMode** : It is a pointer to an integer that specifies the graphics mode to be used. If *graphdriver is set to DETECT, then initgraph sets *graphmode to the highest resolution available for the detected driver.

Description of initgraph()

- **driverDirectoryPath** : It specifies the directory path where graphics driver files (BGI files) are located. If directory path is not provided, then it will search for driver files in current working directory. In all our sample graphics programs, you have to change path of BGI directory accordingly where you turbo C compiler is installed.

Colors in C Graphics Programming

There are **16 colors** declared in C Graphics. We use colors to set the **current drawing color**, **change the color of background**, **change the color of text**, **to color a closed shape** etc. To specify a color, we can either use color constants like `setcolor(RED)`, or their corresponding integer codes like `setcolor(4)`.

Colors along with their codes

| Color Name | Code |
|------------|------|
| BLACK | 0 |
| BLUE | 1 |
| GREEN | 2 |
| CYAN | 3 |
| RED | 4 |
| MAGENTA | 5 |
| BROWN | 6 |
| LIGHTGRAY | 7 |
| DARKGRAY | 8 |
| LIGHTBLUE | 9 |
| LIGHTGREEN | 10 |

| Color Name | Code |
|--------------|------|
| LIGHTCYAN | 11 |
| LIGHTRED | 12 |
| LIGHTMAGENTA | 13 |
| YELLOW | 14 |
| WHITE | 15 |

Graphics Library Functions

1. circle ()

Syntax - circle(int x, int y, int radius);

Circle function is used to draw a circle with **center (x,y)** and third parameter specifies **the radius** of the circle.

Example of circle()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    circle(100, 100, 50);

    getch();
    closegraph();
}
```

Graphics Library Functions

2. line ()

Syntax - `line(int x1, int y1, int x2, int y2);`

line function is used to draw a line from a point(x1,y1) to point(x2,y2) i.e. (x1,y1) and (x2,y2) are end points of the line.

Example of line()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    line(100, 100, 200, 200);

    getch();
    closegraph();
}
```

Graphics Library Functions

3. rectangle ()

Syntax - rectangle(int left, int top, int right, int bottom);

rectangle function is used to draw a rectangle. Coordinates of left top and right bottom corner are required to draw the rectangle. left specifies the X-coordinate of top left corner, top specifies the Y-coordinate of top left corner, right specifies the X-coordinate of right bottom corner, bottom specifies the Y-coordinate of right bottom corner.

Example of rectangle()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    rectangle(100,100,200,200);
    getch();
    closegraph();
}
```

Graphics Library Functions

4. arc ()

Syntax - arc(int x, int y, int stangle, int endangle, int radius);

"arc" function is used to draw an arc with center (x, y) and stangle specifies starting angle, endangle specifies the end angle and last parameter specifies the radius of the arc. arc function can also be used to draw a circle but for that starting angle and end angle should be 0 and 360 respectively.

Example of arc()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\\\TURBOC3\\\\BGI");

    arc(100, 100, 0, 135, 50);
    getch();
    closegraph();
}
```

Graphics Library Functions

5. ellipse ()

Syntax - `ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);`

Ellipse is used to draw an ellipse (x,y) are coordinates of center of the ellipse, stangle is the starting angle, end angle is the ending angle, and fifth and sixth parameters specifies the X and Y radius of the ellipse. To draw a complete ellipse strangles and end angle should be 0 and 360 respectively.

Example of ellipse()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    ellipse(100, 100, 0, 360, 50, 25);

    getch();
    closegraph();
}
```

Graphics Library Functions

6. **drawpoly ()**

Syntax - drawpoly(int num, int *polypoints);

Drawpoly function is used to draw polygons i.e. triangle, [rectangle](#), pentagon, hexagon etc.

num indicates $(n+1)$ number of points where n is the number of vertices in a polygon, polypoints points to a sequence of $(n*2)$ integers . Each pair of integers gives x and y coordinates of a point on the polygon.

Example of drawpoly()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd=DETECT, gm, points[]={320,150,420,300,250,300,320,150};
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    drawpoly(4, points);
    getch();
    closegraph();
}
```

Graphics Library Functions

7. **fillpoly ()**

Syntax - fillpoly(int num, int *polypoints);

Drawpoly function is used to draw polygons i.e. triangle, [rectangle](#), pentagon, hexagon etc. and fills it.

num indicates (n+1) number of points where n is the number of vertices in a polygon, polypoints points to a sequence of (n*2) integers . Each pair of integers gives x and y coordinates of a point on the polygon.

Example of fillpoly()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd=DETECT, gm, points[]={320,150,420,300,250,300,320,150};
    initgraph(&gd, &gm, "C:\\TURBOC3\\BGI");

    fillpoly(4, points);
    getch();
    closegraph();
}
```

Graphics Library Functions

outtextxy ()

Syntax - outtextxy(int x, int y, char *string);

outtextxy function display text or string at a specified point(x,y) on the screen.

x, y are coordinates of the point and third argument contains the address of string to be displayed.

Example of outtextxy()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\\\TURBOC3\\\\BGI");

    outtextxy(100, 100, " My name is Atul");

    getch();
    closegraph();
}
```

Graphics Library Functions

setcolor ()

Syntax - setcolor(int color);

setcolor function is used to change the current drawing color.e.g. setcolor(RED) or setcolor(4) changes the current drawing color to RED. Remember that default drawing color is WHITE.

Example of setcolor()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");

    circle(100,100,50);      /* drawn in white color */
    setcolor(RED);
    circle(200,200,50);      /* drawn in red color  */

    getch();
    closegraph();
}
```

Graphics Library Functions

setbkcolor ()

Syntax - setbkcolor(int color);

setbkcolor function changes current background color e.g. setbkcolor(YELLOW) changes the current background color to YELLOW.

Remember that **default drawing color is WHITE** and **background color is BLACK**.

Example of setbkcolor()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,"C:\\\\TURBOC3\\\\BGI");
    setbkcolor(BLUE);
    circle(100,100,50);    /* drawn in white color */

    getch();
    closegraph();
}
```

Graphics Library Functions

fillellipse ()

Syntax - fillellipse(int x, int y, int xradius, int yradius);

x and y are coordinates of center of the ellipse, xradius and yradius are x and y radius of ellipse respectively.

Example of fillellipse()

```
#include<graphics.h>
#include<conio.h>
main()
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,"C:\\TURBOC3\\BGI");
    fillellipse(100, 100, 50, 25);
    getch();
    closegraph();
}
```


Command Line Arguments

```
void main()
{
    int r;
    float area;
    printf("Enter the radius");
    scanf("%d", &r);
    Arae = 3.14 * r * r;
    printf("%f", area);
}
```

Introduction

What is command line argument? – It is a parameter supplied to a program when the program is invoked at command prompt.

For example – if we want to execute a program to find the area of a circle, and the program is saved with name Area.c, and after compilation Area.exe file is created. Now we can execute the program at command prompt by typing the exe file name.

Like c:\Desktop\Test> Add 10 15

In order to access the command line arguments, we must declare the main function and its parameters as follows:

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

Here the variable argc is an argument counter that contains the number of arguments on the command line. The argv represents an array of character pointers that points to the command line arguments. The size of this array will be the equal to the value of argc.

For example, if at the command line we write

```
C:\> Area 5
```

Here argc is 2 and argv is an array of 2 pointers to strings as shown below:

```
argv[0] -> Area
```

```
argv[1] -> 5
```

Example of command line argument

```
//program to find the area of circle
#include<stdio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
    int r;
    r = atoi( argv[1]);
    area = 3.14 * r * r;
    printf("%d", area);
}
```

Example of command line argument

```
//program to add two numbers
#include<stdio.h>
#include<stdlib.h>
void main(int argc, char *argv[])
{
    int num1, num2;
    num1 = atoi( argv[1]);
    num2 = atoi(argv[2]);
    result = num1 + num2;
    printf("%d", result);
}
```

Write a program to copy the contents of File1 to File 2. File1 contains a paragraph. Both the file names passed as command line arguments.

File Handling in C

Introduction

Till now we have been using the functions such as scanf and printf to read and write data. This works fine **as long as the data is small**.

The major problem is **the entire data is lost** when either the program is **terminated** or the **computer is turned off**.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of ***file*** to store data.

A file is a place on the disk where a group of related data is stored.

Basic File Operations

- Naming a file
- Opening a file
- Reading from a file
- Writing into a file
- Closing a file

C language supports a number of functions that have the ability to perform the above file operations.

Defining and Opening a File

Following is the syntax for declaring and opening a file.

```
FILE *fp;
```

```
fp = fopen("filename", "mode");
```

The first statement declares the variable fp as a “**pointer to the data type FILE**”. FILE is a data structure that is defined in the I/O library.

The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.

The “**mode**” in the 2nd statement specifies the purpose of opening this file.

The mode can be one of the following.

r open the file for **reading** only.

w open the file for **writing** only.

a open the file **appending** (or adding) data to it.

Things may happen during Opening of a file

1. When the mode is “w”, a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the mode is “a”, the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the mode is “r”, and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Example

```
void main()
{
    FILE *p1, *p2;
    p1 = fopen("Data", "r");
    p2 = fopen("Results", "w");
}
```

Here **Data** is a file opened for **reading** purpose and **Results** is a file opened for **writing** purpose.


Closing a file

A file must be closed as soon as all operations on it have been completed. This ensures all the links to the file are broken.

Syntax :- `fclose(file_pointer);`

Example

```
void main()
{
    FILE *p1, *p2;
    p1 = fopen("Data", "r");
    p2 = fopen("Results", "w");
    -----
    -----
    -----
}
```



Operations on file

Input/Output Operations on file

Input to a file – Writing into the file.

Output from a file – Reading from the file.

Input/Output Operations on file

Use of `getc` and `putc` Functions

`getc` – Used to read a single character from a file.

`putc` – Used to write a single character into a file.

Assume that a file is opened with mode “**w**” and the file pointer **fp1**. Then, the statement **`putc(ch, fp1);`**

writes the character contained in the variable **ch** to the file associated with file pointer **fp1**.

The statement

`ch = getc(fp2);`

Would read a character from the file whose pointer is **fp2**, and assigns the value to the variable **ch**.

Write a program to read some characters from keyboard, write it to a file named INPUT, again read the same data from INPUT file, and display it on the screen.

```
void main()
{
    FILE *f1;
    char ch;
    f1= fopen("INPUT", "w");
    printf("Input some characters");
    while((ch=getchar()) !=EOF)
    {
        putc(ch, f1);
    }
    fclose(f1);
```

```
f1= fopen("INPUT", "r");
printf("Displaying the characters on screen");
while((ch=getc()) !=EOF)
{
    printf("%c", ch);
}
fclose(f1);
}
```

```
#include<stdio.h>
void main()
{
    FILE *p1;
    char ch;
    p1 = fopen("INPUT", "w");
    while((ch = getchar())!=EOF)
    {
        putc(ch, p1);
    }
    close(p1);
```

```
p1 = fopen("INPUT", "r");
while( (ch = getc(p1) != EOF)
{
    printf("%c", ch);
}
fclose(p1);
}
```


Input/Output Operations on file

Use of `getw` and `putw` Functions

`getw` – Used to read an integer value from a file.

`putw` – Used to write an integer value into a file.

Assume that a file is opened with mode “**w**” and the file pointer **fp1**. Then, the statement **`putw(x, fp1);`**

writes the integer contained in the variable `x` to the file associated with file pointer **fp1**.

The statement

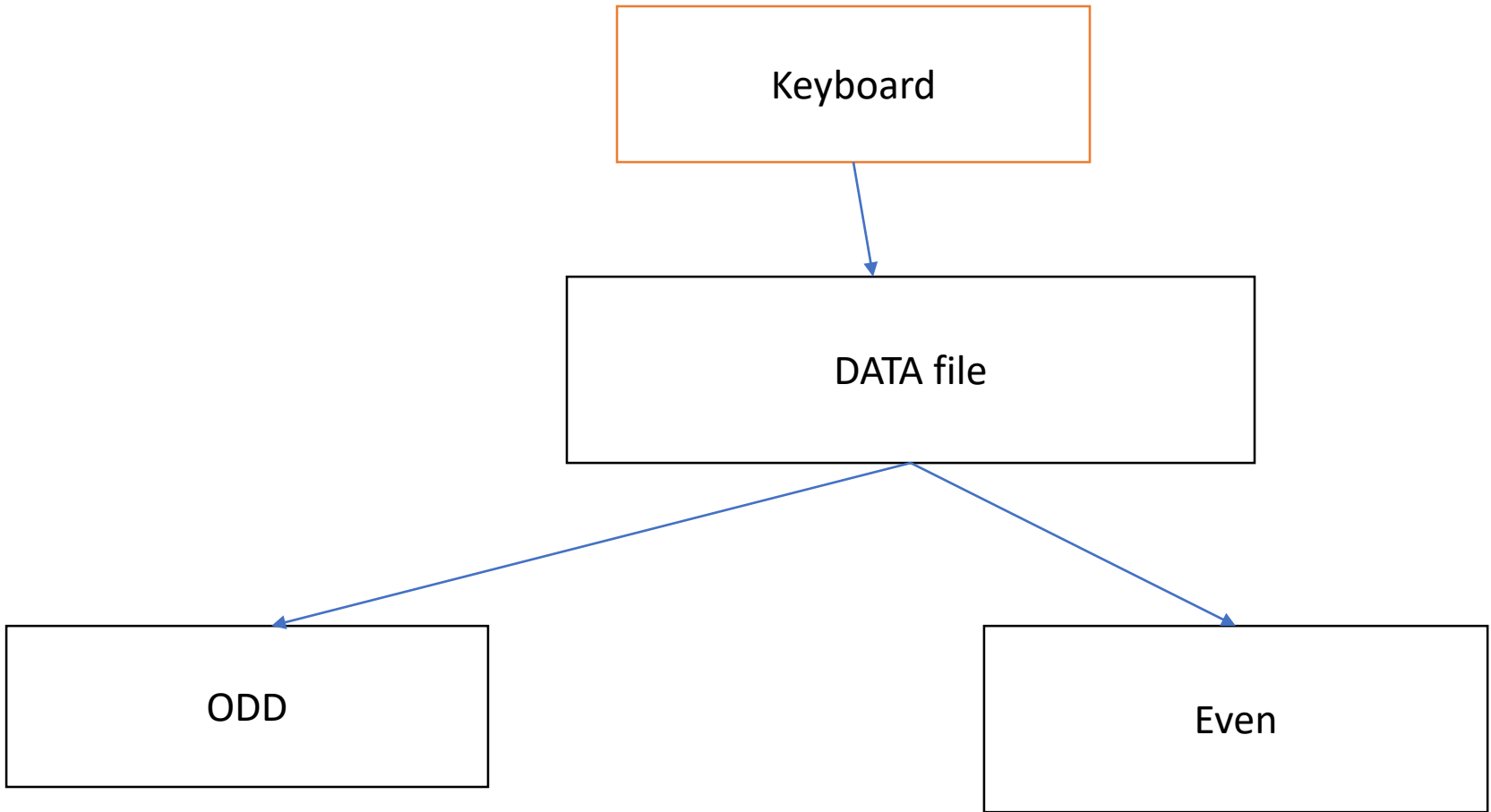
`x = getw(fp2);`

Would read a integer from the file whose pointer is **fp2**, and assigns the value to the variable **x**.

A set of integer numbers entered into a file named DATA from keyboard. Write a program to read these numbers and then write all the odd numbers into a file named ODD and write all the even numbers into a file named EVEN

```
void main()
{
    FILE *f1 *f2, *f3;
    int i,number;
    f1 = fopen ("DATA", "w");
    for(i = 1; i<=30; i++)
    {
        scanf("%d", &number);
        putw(number, f1);
    }
    fclose(f1);
```

```
    f1 = fopen ("DATA", "r");
    f2 = fopen("EVEN", "w";
    f3 = fopen("ODD", "w");
    while((number = getw(f1))!=EOF)
    {
        if(number % 2==0)
            putw(number, f2)
        else
            putw(number, f3)
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
}
```



```
void main()
{
    FILE *p1, *p2, *p3;
    int number,i;
    p1 = fopen("DATA", "w");
    for(i = 1; i<=10; i++)
    {
        scanf("%d", &number);
        putw(number, p1);
    }
    fclose(p1);
```

```
p1 = fopen("DATA", "r");
    p2 = fopen("EVEN", "w");
    p3 = fopen("ODD", "w");
    while((number =getw(p1)) != EOF)
    {
        if(number %2 == 0)
            putw(number, p2);
        else
            putw(number,p3);
    }
    fclose(p1);
    fclose(p2);
    fclose(p3);
```

```
f2 = fopen("EVEN", "r");
printf("The contents of Even
file\n");
while((number = getw(f2)) !=EOF)
{
    printf("%d", number);
}
fclose(f2);
```

```
f3 = fopen("ODD", "r");
printf("The contents of Odd
file\n");
while((number = getw(f3))!=EOF)
{
    printf("%d", number);
}
fclose(f3);
}
```

Input/Output Operations on file

Use of fprintf and fscanf Functions

fprintf and fscanf can handle a group of mixed data simultaneously.

The **syntax** of fprintf is

fprintf(fp, “control string”, list);

Where **fp** is a **file pointer** associated with a file that has been opened for writing. The **control string** contains **output specification** for the items in the list. The **list** may contain **variables or constants**.

Ex- fprintf(f1, “%s %d %f”, name, age, per);

```
fprintf(fp, "controlstring",varname);
```

```
fscanf(fp, "control string", &varname, &varname.....);
```

Input/Output Operations on file

Use of fprintf and fscanf Functions

The **syntax** of fscanf is

fscanf(fp, “control string”, list);

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.

Ex- fscanf(f2, “%s %d”, &item, &quantity);

Example of fprintf()

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

```
void main () {
```

```
    FILE * fp;
```

```
    fp = fopen ("file.txt", "w");
```

```
    fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
```

```
    fclose(fp);
```

```
}
```

Example of fscanf()

```
#include <stdio.h>

void main () {
    FILE *fp;
    char str1[10];
    char str2[10];
    char str3[10];
    int num;
    fp = fopen("file.txt","r");
    while(1) {
        fscanf(fp, "%s %s %s %d", &str1, &str2, &str3, &num);
        if( feof(fp) ) {
            break;
        }
        printf("%s %s %s %d", str1,str2,str3,num);
    }
    fclose(fp);
}
```

Till now we have discussed

1. `getc()` & `putc`
2. `getw()` & `putw()`
3. `fscanf()` & `fprintf()`

Use of fwrite() & fread()

```
#include <stdio.h>
#include <stdlib.h>
struct person
{
    int id;
    char fname[20];
    char lname[20];
};

int main ()
{
    FILE *outfile;
    outfile = fopen ("person.dat", "w");
    struct person input1 = {1, "rohan", "sharma"};
    struct person input2 = {2, "mahendra", "dhoni"};
    fwrite (&input1, sizeof(struct person), 1, outfile);
    fwrite (&input2, sizeof(struct person), 1, outfile);
    fclose (outfile);
}
```

```
Void main()
{
    struct person
    {
        int id;
        char fname[20];
        char lname[20];
    };

    int main ()
    {
        FILE *outfile;
        int i;
        outfile = fopen ("person.dat", "w");
        struct person p[10];
        for (i = 0; i<10; i++)
        {
            scanf("%d%s%s", &p[i].id, &p[i].fname,
            &p[i].lname);
        }
    }
}
```

```
fwrite(&p, sizeof (struct person p), 1, outfile);  
fclose(outfile);  
}
```

fwrite() & fread()

fwrite() – is used to write a record or array of records at a time into a file.

fread() – is used to read a record or array of records at a time from a file.

Syntax – fwrite(&structVariable, sizeof(struct), 1, filepointer);
fread(&structVariable, sizeof(struct), 1, filepointer);

```
void main()
{
    struct student
    {
        int roll;
        char name[30];
    };
    FILE *fp1,*fp2;
    struct student s1 ={1, "Rohan"};
    struct student s2 = {2, "Rakesh"};
    fp1 = fopen("DATA1", "w");
    fp2 = fopen("DATA2", "w");
    fwrite(&s1, sizeof(struct student), 1, fp1);
    fwrite(&s2, sizeof(struct student),1, fp2);
    fclose(fp1);
}
```

Use of fwrite() & fread()

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

```
struct person
```

```
{
```

```
    int id;
```

```
    char fname[20];
```

```
    char lname[20];
```

```
};
```

```
int main ()
```

```
{
```

```
    FILE *infile;
```

```
    struct person input;
```

```
    infile = fopen ("person.dat", "r");
```

```
    while(fread(&input, sizeof(struct person), 1, infile))
```

```
        printf ("id = %d name = %s %s", input.id, input.fname, input.lname);
```

```
        fclose (infile);
```

```
}
```



```
void main()
{
    struct student
    {
        int roll;
        char Name[30];
    };
    FILE *p1;
    int i;
    struct student s[10];
    struct student x[10];
    P1 = fopen("DATA", "w");
    for( i = 0; i<10; i++)
    {
        scanf("%d %s", &s[i].roll, &s[i].Name);
    }
    fwrite(&s, sizeof(struct student s), 1, p1);
    fclose(p1);
```

```
p1 = fopen("DATA", "r");
fread(&x, sizeof(struct student s), 1, p1);
for( i = 0; i<10; i++)
{
    printf("%d %s", x[i].roll, x[i].name);
    printf("\n");
}
fclose(p1);
}
```

Random Access to Files

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek**, **ftell**, and **rewind** available in the I/O library.

ftell() –

Syntax – ftell(fp);

ftell takes a file pointer and returns a number of type long, that corresponds to the current position.

Example –

n = ftell(fp);

n would give the relative offset(in bytes) of the current position. This means that n bytes have already been read(or written).

rewind() -

Syntax – `rewind(fp);`

`rewind` takes a file pointer and resets the position to the start of the file.

Example –

```
rewind (fp);
```

```
n = ftell(fp);
```

This would assign **0** to **n** because the file position has been set to the start of the file by **rewind**.

fseek()

fseek function is used to move the file position to a desired location within the file.

Syntax – fseek(file_ptr, offset, position);

file_ptr is a pointer to the file concerned, **offset** is a number or variable of **type long**, and **position** is an **integer number**. The offset specifies the number of positions (bytes) to be moved from the location specified by position.

The position can take one of the following three values:

- 0 ----- Beginning of the file
- 1----- Current position
- 2 ----- End of file

The offset may be positive, meaning move forward, or negative, meaning move backwards.

Note – fseek() will return a value of 0, when the operation is successful, otherwise will return a -1 value.

Operations of fseek function

| Statement | Meaning |
|--------------------------------|--|
| <code>fseek(fp, 0L, 0)</code> | Go to the beginning(similar to rewind) |
| <code>fseek(fp, 0L, 1)</code> | Stay at the current position |
| <code>fseek(fp, 0L, 2)</code> | Go to the end of the file |
| <code>fseek(fp, m, 0)</code> | Move to (m+1) byte in the file |
| <code>fseek(fp, m, 1)</code> | Go forward by m bytes from the current position |
| <code>fseek(fp, -m, 1)</code> | Go backward by m bytes from the current position |
| <code>fseek(fp, -m, 2)</code> | Go backward by m bytes from the end |

```
void main()
{
FILE *fp;
long int m;
m = 5L;
fp = fopen("DATA", "r");
fseek(fp,m,1);
```

Example of fseek

Create a file named DATA, enter the characters A to Z into it. Print every 5th character of this file. Now move to the end of the file and print the contents of the file in reverse order.

```
#include<stdio.h>
void main()
{
    FILE *p1;
    long n;
    char ch;
    p1 = fopen("DATA", "w");
    while(( ch = getchar()) != EOF)
    {
        putc(ch, p1);
    }
    fclose(p1);
```

```
p1 = fopen("DATA", "r");
    n = 0L;
    while(feof() == 0)
    {
        fseek(p1, n,0);
        ch = getc(p1);
        printf("%c", ch);
        n = n +5L;
    }
```


Example of fseek cont...

```
fseek(p1, -1L, 2)//position to last character
do
{
    ch = getc(p1);
    printf("%c", ch);
} while((fseek(p1, -1L, 1) == 0);
fclose(p1);
}
```