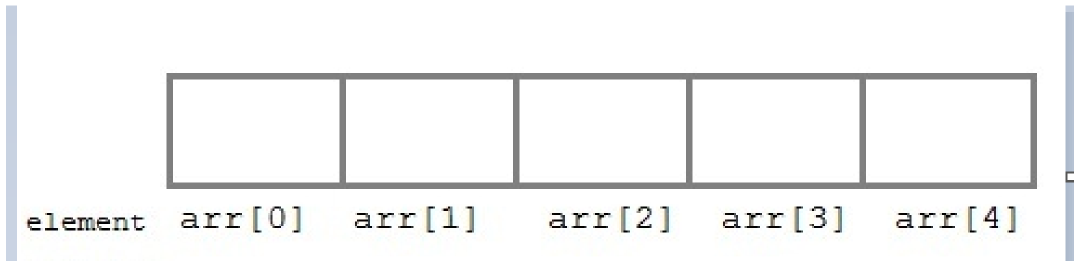
Pointer and Arrays in C

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array `arr`,

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

Assuming that the base address of `arr` is 1000 and each integer requires two/four bytes, the five elements will be stored as follows:



Address 1000 1004 1008 1012 1016

Note that there is a difference of 2/4 bytes between each element because that's the size of an integer. Which means all the elements are stored in consecutive contiguous memory locations in the memory

`arr` is equal to `&arr[0]` by default

We can also declare a pointer of type `int` to point to the array `arr`.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array `arr` using `p++` to move from one element to another.

NOTE: You cannot decrement a pointer once incremented. `p--` won't work.

Pointer to Array

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

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

```
int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;      // same as int*p = &a[0]
    for (i = 0; i < 5; i++)
    {
        printf("%d", *p);
        p++;
    }

    return 0;
}
```

Pointer to Array of Structures in C

Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

Accessing Structure Members with Pointer

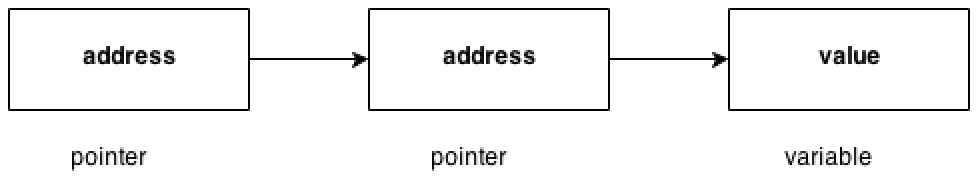
To access members of structure using the structure variable, we used the dot `.` operator. But when we have a pointer of structure type, we use arrow `->` to access structure members.

```
#include <stdio.h>

struct my_structure {
    char name[20];
    int number;
    int rank;
};

int main()
{
    struct my_structure variable = {"StudyTonight", 35, 1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);

    return 0;
}
```

OUTPUT : NAME: StudyTonight NUMBER: 35 RANK: 1

Pointer to Pointer

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.

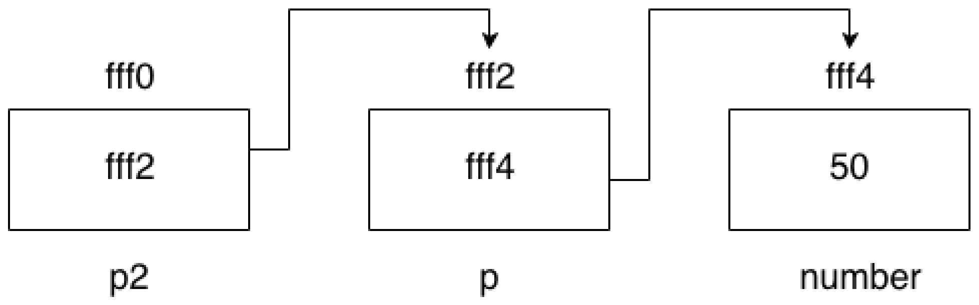


The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
int **p2;//pointer to pointer
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of *p variable is %d \n",*p);
printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50
```

File Input/Output in C

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file.

```
FILE *fp;
```

C provides a number of functions that helps to perform basic file operations. Following are the functions,

FOPEN()

The C library function **FILE *fopen(const char *filename, const char *mode)** opens the **filename** pointed to, by filename using the given **mode**.

```
FILE *fopen(const char *filename, const char *mode)
```

Sr.No.	Mode & Description
1	**"r"** Opens a file for reading. The file must exist.
2	**"w"** Creates an empty file for writing. If a file with the same name already exists then its content is erased and the file is considered as a new empty file.
3	**"a"** Appends to a file. Writing operations appends data at the end of the file. The file is created if it does not exist.

4	**"r+"** Opens a file to update both reading and writing. The file must exist.
5	**"w+"** Creates an empty file for both reading and writing.
6	**"a+"** Opens a file for reading and appending.

Example

The following example shows the usage of fopen() function.

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

int main () {
   FILE * fp;

   fp = fopen ("file.txt", "w+");
   fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

   fclose(fp);

   return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content −

```
We are in 2012
```

Now let us see the content of the above file using the following program −

```
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");
   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);

   return(0);
}
```

Let us compile and run the above program to produce the following result −

```
We are in 2012
```

FREOPEN()

The C library function **FILE *freopen(const char *filename, const char *mode, FILE *stream)** associates a new **filename** with the given open stream and at the same time closes the old file in the stream.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream)
```

Parameters

- **filename** − This is the C string containing the name of the file to be opened.
- **mode** − This is the C string containing a file access mode. It includes −

- **stream** − This is the pointer to a FILE object that identifies the stream to be re-opened.

```
#include <stdio.h>

int main () {
   FILE *fp;

   printf("This text is redirected to stdout\n");

   fp = freopen("file.txt", "w+", stdout);

   printf("This text is redirected to file.txt\n");

   fclose(fp);

   return(0);
}
```

FPRINTF()

The C library function **int fprintf(FILE *stream, const char *format, ...)** sends formatted output to a stream.

```
int fprintf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **format** − This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below −

1	**c** Character
2	**d or i** Signed decimal integer
3	**e** Scientific notation (mantissa/exponent) using e character
4	**E** Scientific notation (mantissa/exponent) using E character
5	**f** Decimal floating point
6	**g** Uses the shorter of %e or %f
7	**G** Uses the shorter of %E or %f
8	**o** Signed octal
9	**s** String of characters
10	**u** Unsigned decimal integer
11	**x** Unsigned hexadecimal integer

12	**X** Unsigned hexadecimal integer (capital letters)
13	**p** Pointer address
14	**n** Nothing printed
15	**%** Character

REMOVE()

The C library function int remove(const char *filename) deletes the given filename so that it is no longer accessible.

```
int remove(const char *filename)
```

Parameters

- **filename** − This is the C string containing the name of the file to be deleted.

Example

The following example shows the usage of remove() function.

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

int main () {
   int ret;
   FILE *fp;
   char filename[] = "file.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%s","I am an indian");
   fclose(fp);
   ret = remove(filename);

   if(ret == 0) {
      printf("File deleted successfully");
   } else {
      printf("Error: unable to delete the file");
   }

   return(0);
}
```

RENAME ()

Description

The C library function **int rename(const char *old_filename, const char *new_filename)** causes the filename referred to by **old_filename** to be changed to **new_filename**.

```
int rename(const char *old_filename, const char *new_filename)
```

Parameters

- **old_filename** − This is the C string containing the name of the file to be renamed and/or moved.
- **new_filename** − This is the C string containing the new name for the file.

Example

The following example shows the usage of rename() function.

```
#include <stdio.h>

int main () {
   int ret;
   char oldname[] = "file.txt";
   char newname[] = "newfile.txt";

   ret = rename(oldname, newname);

   if(ret == 0) {
      printf("File renamed successfully");
   } else {
      printf("Error: unable to rename the file");
   }

   return(0);
}
```

FOEF()

The C library function **int feof(FILE *stream)** tests the end-of-file indicator for the given stream.

```
int feof(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

Example

The following example shows the usage of feof() function.

```
#include <stdio.h>

int main () {
   FILE *fp;
   int c;

   fp = fopen("file.txt","r");


   while(1) {
      c = fgetc(fp);
      if( feof(fp) ) {
         break ;
      }
      printf("%c", c);
   }
   fclose(fp);

   return(0);
}
```

FERROR()

The C library function **int ferror(FILE *stream)** tests the error indicator for the given stream.

```
int ferror(FILE *stream)
```

```
#include <stdio.h>

int main () {
   FILE *fp;
   char c;

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

   c = fgetc(fp);
   if( ferror(fp) ) {
      printf("Error in reading from file : file.txt\n");
   }
   clearerr(fp);

   if( ferror(fp) ) {
      printf("Error in reading from file : file.txt\n");
   }
   fclose(fp);

   return(0);
}
```

FFLUSH()

fflush() is typically used for output stream only. Its purpose is to clear (or flush) the output buffer and move the buffered data to console (in case of stdout) or disk (in case of file output stream)/ C program to illustrate situation

```
// where flush(stdin) is required only

// in certain compilers.

#include <stdio.h>

#include<stdlib.h>

int main()

{

        char str[20];

        int i;

        for (i=0; i<2; i++)

        {

                scanf("%[^\n]s", str);

                printf("%s\n", str);

                // fflush(stdin);

        }

        return 0;

}
```

Input:

```
geeks
geeksforgeeks
```

Output:

```
geeks
geeks
```

```
// C program to illustrate flush(stdin)
// This program works as expected only
// in certain compilers like Microsoft
// visual studio.
#include <stdio.h>
#include<stdlib.h>
int main()
{
    char str[20];
    int i;
    for (i = 0; i<2; i++)
    {
        scanf("%[^\n]s", str);
        printf("%s\n", str);

        // used to clear the buffer
        // and accept the next string
        fflush(stdin);
    }
    return 0;
}
```

Input:

```
geeks
geeksforgeeks
```

Output:

```
geeks
geeksforgeeks
```

Opening a File or Creating a File

The `fopen()` function is used to create a new file or to open an existing file.

General Syntax:

```
*fp = FILE *fopen(const char *filename, const char *mode);
```

Here, `*fp` is the FILE pointer (`FILE *fp`), which will hold the reference to the opened(or created) file.

filename is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

mode	Description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The `fclose()` function is used to close an already opened file.

General Syntax :

```
int fclose( FILE *fp);
```

Here `fclose()` function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. `getc()` and `putc()` are the simplest functions which can be used to read and write individual characters to a file.

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

int main( )
{

    // Declare the file pointer
    FILE *filePointer ;

    // Get the data to be written in file
    char dataToBeWritten[50]
        = "GeeksforGeeks-A Computer Science Portal for Geeks";

    // Open the existing file GfgTest.c using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("GfgTest.c", "w") ;

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if ( filePointer == NULL )
    {
        printf( "GfgTest.c file failed to open." ) ;
    }
    else
    {

        printf("The file is now opened.\n") ;

        // Write the dataToBeWritten into the file
        if ( strlen (  dataToBeWritten  ) > 0 )
        {
            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer) ;
            fputs("\n", filePointer) ;
        }

        // Closing the file using fclose()
        fclose(filePointer) ;

        printf("Data successfully written in file GfgTest.c\n");
        printf("The file is now closed.") ;
    }
    return 0;
}
```

Reading and Writing to File using `fprintf()` and `fscanf()`

```
#include<stdio.h>

struct emp
{
    char name[10];
    int age;
};

void main()
{
    struct emp e;
    FILE *p,*q;
    p = fopen("one.txt", "a");
    q = fopen("one.txt", "r");
    printf("Enter Name and Age:");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
    do
    {
        fscanf(q,"%s %d", e.name, e.age);
        printf("%s %d", e.name, e.age);
    }
    while(!feof(q));
}
```

FGETPOS()

The C library function **int fgetpos(FILE *stream, fpos_t *pos)** gets the current file position of the **stream** and writes it to **pos**.

Declaration

Following is the declaration for fgetpos() function.

```
int fgetpos(FILE *stream, fpos_t *pos)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **pos** − This is the pointer to a fpos_t object.

Return Value

This function returns zero on success, else non-zero value in case of an error.

Example

The following example shows the usage of fgetpos() function.

```
#include <stdio.h>

int main () {
   FILE *fp;
   fpos_t position;

   fp = fopen("file.txt","w+");
   fgetpos(fp, &position);
   fputs("Hello, World!", fp);

   fsetpos(fp, &position);
   fputs("This is going to override previous content", fp);
   fclose(fp);

   return(0);
}
```

OUTPUT

```
This is going to override previous content
```

SPRINTF()

sprintf()	Declaration: int **sprintf**(char *string, const char *format, …) sprintf function is used to write formatted output to the string, In a C program, we use fgets function as below. sprintf (string, "%d %c %f", value, c, flt) ; where,

	string – buffer to put the data in. value – int variable, c – char variable and flt – float variable. There are for example only. You can use any specifiers.

```
#include <stdio.h>

#include <string.h>

Void main( )

{

  int value = 50 ;

  float flt = 7.25 ;

  char c = 'Z' ;

  char string[40] = {‘\0’} ;

  printf ( "int value = %d \n char value = %c \n " \

        "float value = %f", value, c, flt ) ;

  /*Now, all the above values are redirected to string

    instead of stdout using sprint*/

  printf(“\n Before using sprint, data in string is %s”, string);

  sprintf ( string, "%d %c %f", value, c, flt );

  printf(“\n After using sprint, data in string is %s”, string);

}
```

Output:

```
int value = 50
char value = Z
float value = 7.25
Before using sprint, data in string is NULL
After using sprint, data in string is “50 Z 7.25”
```

Command Line Argument in C

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the `main()` method.

Syntax:

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

Here `argc` counts the number of arguments on the command line and `argv[ ]` is a pointer array which holds pointers of type `char` which points to the arguments passed to the program.

Example for Command Line Argument

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

int main(int argc, char *argv[])
{
    int i;
    if( argc >= 2 )
    {
        printf("The arguments supplied are:\n");
        for(i = 1; i < argc; i++)
        {
            printf("%s\t", argv[i]);
        }
    }
    else
    {
        printf("argument list is empty.\n");
    }
    return 0;
}
```

Remember that `argv[0]` holds the name of the program and `argv[1]` points to the first command line argument and `argv[n]` gives the last argument. If no argument is supplied, `argc` will be 1

```
C:\TurboC++\Disk\TurboC3\SOURCE\yourprj.exe a b c
```