UNIT –V **COMPUTER PROGRAMMING**

Reading and Writing Characters –Reading and Writing Strings –Formatted Console I/O –Printf - Scanf–Standard C Vs Unix File I/O–Streams and Files–File System Basics–Fread and Fwrite–Fseek and Random Access I/O –Fprintf ( ) and Fscanf( )–The Standard Streams –The Preprocessor Directives #define and #include.

CHARACTER I/O FUNCTIONS

Character input functions read one character at a time from a text stream. Character output functions write one character at the time to a text stream.

These functions can be divided into two categories,

1. Terminal Character I/O
2. Terminal and File Character I/O

TERMINAL CHARACTER I/O

C declares a set of character input/output functions that can only be used with the standard streams: standard input (stdin), standard output (stdout).

Read a Character: getchar ()

This function is to read exactly one character from the keyboard; it reads the next character from the standard input stream.

Syntax for using getchar () function is as follows,

```
ch =getchar ();
```

Its return value is integer. Up on successful reading returns the ASCII value of character. If there is an error returns EOF.

Write a Character: putchar ()

This function provides for printing exactly one character to the Monitor.

Syntax for using putchar () function is as follows,

```
ch =getchar ();  /* input a character from keyboard*/

putchar (ch);      /* display character on the Monitor */
```

Its return value is integer. Up on successful writing returns the ASCII value of character. If there is an error returns EOF.

Program:

```
#include <stdio.h>
Void  main( )
{
  int c;

  printf( "Enter a value :");
  c = getchar( );

  printf( "\nYou entered: ");
  putchar( c );
}
```

TERMINAL AND FILE CHARACTER I/O

The terminal character input/output functions are designed for convenience; we don’t need to specify the stream. Here, we can use a more general set of functions that can be used with both the standard streams and a file.

These functions require an argument that specifies the stream associated with a terminal device or a file.

- ❖ When used with a terminal device, the streams are declared and opened by the system, the standard input stream (stdin) for the keyword and standard output stream (stdout) for the monitor.

- ❖ When used with a file, we need to explicitly declare the stream, it is our responsibility to open the stream and associate with the file.

Read a Character: getc () and fgetc ()

The getc functions read the next character from the file stream, which can be a used-defined stream or stdin, and converts it in to an integer. This function has one argument which is the file pointer declared as FILE.

If the read detects an end of file, the function returns EOF, EOF is also returned if any error occurs. The functionality of getc/fgetc same.

Syntax for using getc/fgetc,

```
getc (fp);

fgetc (fp);
```

Example,

```
char ch;
ch = getc (stdin);        /* input from keyboard */
ch =fgetc (fileptr);      /* input from a file */
```

Write a Character: putc () and fputc ()

The putc functions write a character to the file stream specified which can be a user-defined stream, stdout, or stderr. The functionality of putc/ fputc same.

For fputc, the function takes two arguments. The first parameter is the character to be written and the second parameter is the file, The second parameter is the file pointer declared as FILE.

If the character is successfully written, the function returns it. If any error occurs, it returns EOF.

Syntax,

```
putc (char, *fp);

fputc (char, *fp);
```

Example,

```
char ch;
ch = getc (stdin);              /* input from keyboard */
putc (ch, stdout);              /* output to the screen */
putc (ch, outfileptr);     /*output to a file */
```

LINE I/O FUNCTIONS

Reading Strings: gets () and fgets ()

The gets and fgets function take a line (terminated by a new line) from the input stream and make a null-terminated string out of it. These are sometimes called line-to-string input function.

gets ()

- ❖ gets function reads a string from terminal, reading is terminated by new line.
- ❖ The newline (\n) indicates the end of the string, it is replaced by null (\0) character in the memory by the function gets () as shown in Figure 6.5.
- ❖ The source of data for the gets function is standard input.
- ❖ gets function takes only one parameter, the name of the string, its syntax is as follows,

```
gets (string_name);
```

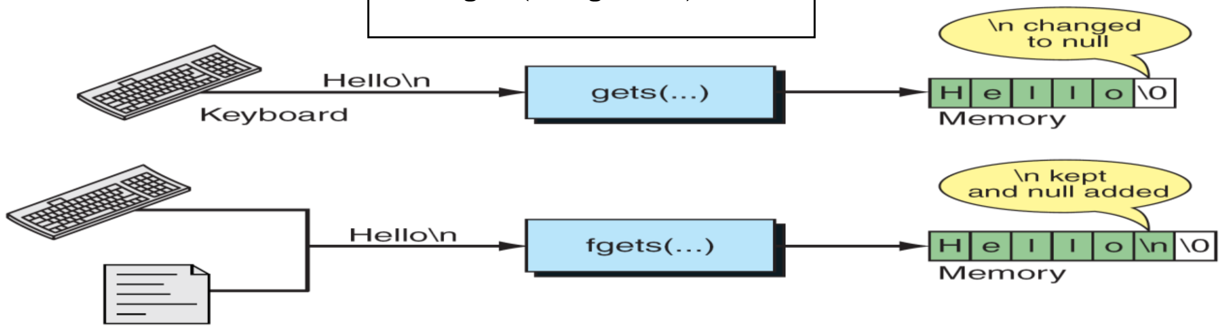


Figure: Working of gets/fgets functions

fgets ()

- fgets function reads a string from file or keyboard.
- It reads characters from the specified file until a new line character has been read or until n-1 characters has been read, whichever occurs first. The function automatically places null character at the end as shown in Figure 6.5.
- The source of data for fgets can be a file or standard input.
- The general format,

```
fgets (string, length, fp);
```

- First argument is name of the string in which the read data from the file is to be placed. Second argument is the length of the string to be read and last argument is the file pointer.
- fgets also reads data from terminal. Below example illustrates this,

```
fgets (str, 10, stdin);
```

Writing Strings: puts () and fputs ()

puts/fputs functions take a null-terminated string from memory and write it to a file or the monitor. These are sometimes called string-to-line output functions.

puts ()

- puts function writes a string to terminal, writing is terminated by null character. the null character is replaced with a new line inputs as shown in Figure 6.6.
- puts function takes only one parameter, the name of the string, its syntax is as follows,

```
puts (string_name);
```

fputs ()

- fputs function writes a string to a file or monitor.
- Characters stored in the string are written to the file identified by fileptr until the null character is reached. The null character is not written to the file as shown in Figure 6.6.
- The general format forms are

```
fputs (string, fp);
```

- First argument is address of the string in which the read data is to be placed. Second argument is the file pointer.

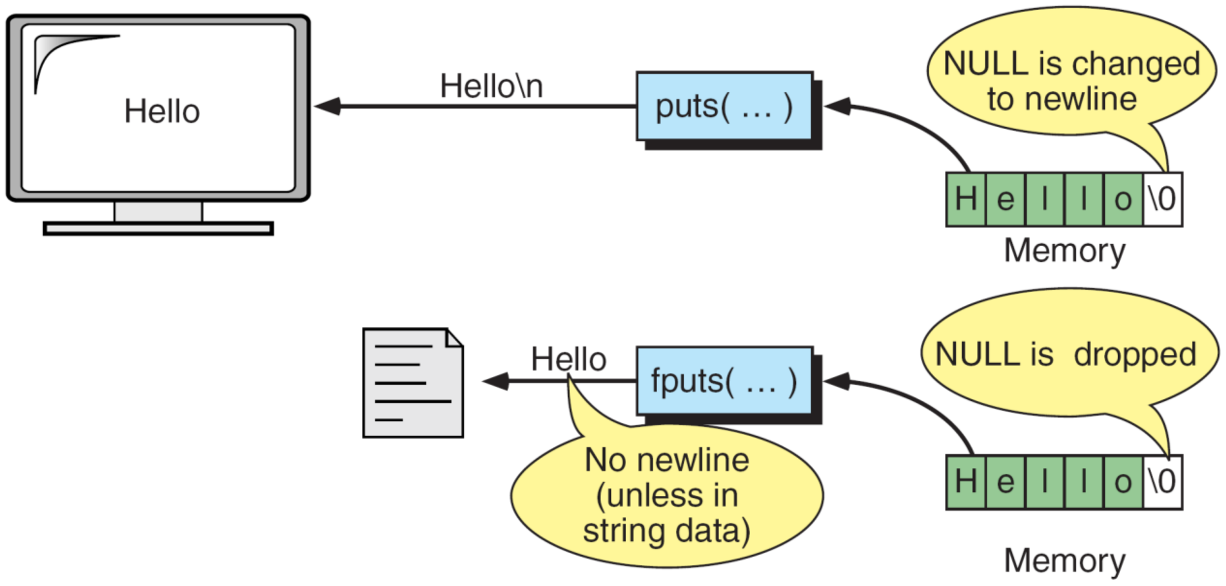


Figure : Working of puts/fputs Functions

FORMATED CONSOLE I/O

We have already familiar with two formatting functions scanf and printf. The functions printf() and scanf() perform formatted output and input –i.e.., they can read and write the data in various formats that are under control.

The printf() function writes data to the console. The scanf() function reads data from the keyboard.These two functions can be used only with the keyboard and monitor.

The C library defines two more general functions, fscanf and fprintf, that can be used with any txt stream.

Formatted Output with printf ()

- ❖ This function provides for formatted output to the screen. The syntax is:

  printf (“format string”, var1, var2 …);
- ❖ The “format string” includes a listing of the data types of the variables to be output and, optionally, some text and control character(s).
- ❖ Example:

```
float a=10.5; int b=15;
printf (“You entered %f and %d \n”, a, b);
```

Format Conversion Specifiers

d -- displays a decimal (base 10) integer
l -- used with other Specifiers to indicate a "long"
e -- displays a floating point value in exponential notation
f -- displays a floating point value
g -- displays a number in either "e" or "f" format
c -- displays a single character
s -- displays a string of characters

Formatted Input with scanf ()

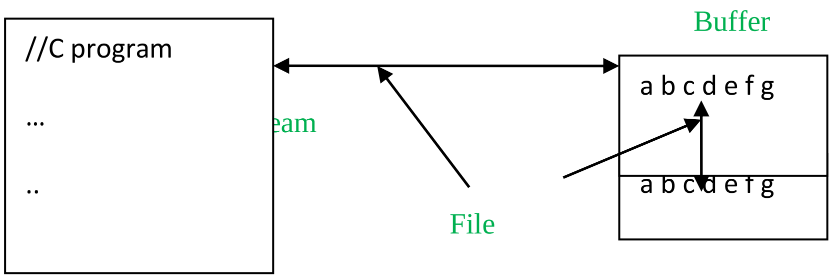
- ❖ This function provides for formatted input from the keyboard. The syntax is:

  scanf (“format string” , &var1, &var2, …) ;
- ❖ The “format string” is a listing of the data types of the variables to be input and the & in front of each variable name tells the system Where to store the value that is input.
- ❖ It provides the address for the variable.
- ❖ Example:

```
float a; int b;
scanf (“%f %d”, &a, &b);
```

STREAM

- ❖ A stream is a general name given to a flow of data.
- ❖ All input and output is performed with streams.
- ❖ A "stream" is a sequence of characters organized into lines.
- ❖ Each line consists of zero or more characters and ends with the "newline" character.
- ❖ ANSI C standards specify that the system must support lines that are at least 254 characters in length (including the new line character).
- ❖ A stream can be associated with a physical device, terminal, or with the file stored in memory. The following Figure 6.1 illustrates the data flow between external device(c Program), buffer and file.



Text and binary Streams

- ❖ C uses two types of streams: text and binary.
- ❖ A text stream consists of a sequence of characters divided into lines with each line terminated by a new line (\n).

- ❖ A binary stream consists of sequence of data values such as integer, real, or complex using their memory representation.

System Created Streams

- ❖ Standard input stream is called "stdin" and is normally connected to the keyboard
- ❖ Standard output stream is called "stdout" and is normally connected to the display screen.
- ❖ Standard error stream is called "stderr" and is also normally connected to the screen.

STREAM FILE PROCESSING

A file exists as an independent entity with a name known to the O.S. A stream is an entity created by the program. To use a file in our program, we must associate the program's stream name with the file name.

In general, there are four steps to processing a file.

1. Create a stream
2. Open a file
3. Process the file (read or write data)
4. Close file

Creating a Stream

We can create a stream when we declare it. The declaration uses the FILE type as shown below,

```
FILE *fp;
```

The FILE type is a structure that contains the information needed for reading and writing a file, fp is a pointer to the stream.

Opening File

Once stream has been created, we can ready to associate to a file. In the next section we will discuss in detail.

Closing the Stream

When file processing is complete, we close the file. After closing the file the stream is no longer available.

Files:

When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, these information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language. In this tutorial, you will learn to handle standard I/O(High level file I/O functions) in C.

High level file I/O functions can be categorized as:

1. Text file
2. Binary file

File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

File System Basics

While working with file, you need to declare a pointer of type file. This declaration is needed for communication between file and program.

```
FILE *ptr;
```

Opening a file

Opening a file is performed using library function fopen(). The syntax for opening a file in standard I/O is:

```
ptr=fopen("fileopen","mode")
```

For Example:

fopen("E:\\cprogram\program.txt","w");

```
/* ------------------------------------------------------- */
 E:\\cprogram\program.txt is the location to create file.
 "w" represents the mode for writing.
/* ------------------------------------------------------- */
```

Here, the program.txt file is opened for writing mode.

Opening Modes in Standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.

Closing a File

The file should be closed after reading/writing of a file. Closing a file is performed using library function fclose().

```
fclose(ptr);
```

//ptr is the file pointer associated with file to be closed.

The Functions fprintf() and fscanf() functions.

The functions fprintf() and fscanf() are the file version of printf() and fscanf(). The only difference while using fprintf() and fscanf() is that, the first argument is a pointer to the structure FILE

Writing to a file

```
#include <stdio.h>
int main()
{
   int n;
   FILE *fptr;
   fptr=fopen("C:\\program.txt","w");
   if(fptr==NULL){
      printf("Error!");
      exit(1);
   }
   printf("Enter n: ");
   scanf("%d",&n);
   fprintf(fptr,"%d",n);
   fclose(fptr);
```

```
  return 0;
}
```

This program takes the number from user and stores in file. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open that file, you can see the integer you entered.
Similarly, fscanf() can be used to read data from file.

Reading from file

```
#include <stdio.h>
int main()
{
  int n;
  FILE *fptr;
  if ((fptr=fopen("C:\\program.txt","r"))==NULL){
     printf("Error! opening file");
     exit(1);        /* Program exits if file pointer returns NULL. */
  }
  fscanf(fptr,"%d",&n);
  printf("Value of n=%d",n);
  fclose(fptr);
  return 0;
}
```

If you have run program above to write in file successfully, you can get the integer back entered in that program using this program.
Other functions like fgetchar(), fputc() etc. can be used in similar way.

Binary Files

Depending upon the way file is opened for processing, a file is classified into text file and binary file.

If a large amount of numerical data it to be stored, text mode will be insufficient. In such case binary file is used.

Working of binary files is similar to text files with few differences in opening modes, reading from file and writing to file.

Opening modes of binary files

Opening modes of binary files are rb, rb+, wb, wb+,ab and ab+. The only difference between opening modes of text and binary files is that, b is appended to indicate that, it is binary file.

Reading and writing of a binary file.

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Function fwrite() takes four arguments, address of data to be written in disk, size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_data,numbers_data,pointer_to_file);
```

Function fread() also take 4 arguments similar to fwrite() function as above.

Write a C program to read name and marks of n number of students from user and store them in a file

```
#include <stdio.h>
int main(){
  char name[50];
  int marks,i,n;
  printf("Enter number of students: ");
  scanf("%d",&n);
  FILE *fptr;
  fptr=(fopen("C:\\student.txt","w"));
  if(fptr==NULL){
     printf("Error!");
     exit(1);
  }
  for(i=0;i<n;++i)
  {
    printf("For student%d\nEnter name: ",i+1);
    scanf("%s",name);
    printf("Enter marks: ");
    scanf("%d",&marks);
    fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
  }
  fclose(fptr);
  return 0;
}
```

fread() and fwrite()

fread():

The C library function

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

reads data from the given **stream** into the array pointed to, by **ptr**.

Declaration

Following is the declaration for fread() function.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** − This is the pointer to a block of memory with a minimum size of *size*nmemb* bytes.
- **size** − This is the size in bytes of each element to be read.
- **nmemb** − This is the number of elements, each one with a size of **size** bytes.
- **stream** − This is the pointer to a FILE object that specifies an input stream.

Return Value

The total number of elements successfully read are returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, then either an error had occurred or the End Of File was reached.

Example

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

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

int main()
{
  FILE *fp;
  char c[] = "This is Computer Programming";
  char buffer[100];

  /* Open file for both reading and writing */
  fp = fopen("file.txt", "w+");

  /* Write data to the file */
  fwrite(c, strlen(c) + 1, 1, fp);

  /* Seek to the beginning of the file */
  fseek(fp, SEEK_SET, 0);

  /* Read and display data */
  fread(buffer, strlen(c)+1, 1, fp);
  printf("%s\n", buffer);
  fclose(fp);

  return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** and write a content *this is tutorialspoint*. After that, we use **fseek()** function to reset writing pointer to the beginning of the file and prepare the file content which is as follows −

```
This is Computer programming
```

Fwrite():

The C library function

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

writes data from the array pointed to, by **ptr** to the given **stream**.

Declaration

Following is the declaration for fwrite() function.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

Parameters

- **ptr** − This is the pointer to the array of elements to be written.
- **size** − This is the size in bytes of each element to be written.
- **nmemb** − This is the number of elements, each one with a size of **size** bytes.
- **stream** − This is the pointer to a FILE object that specifies an output stream.

Return Value

This function returns the total number of elements successfully returned as a size_t object, which is an integral data type. If this number differs from the nmemb parameter, it will show an error.

Example

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

```
#include<stdio.h>

int main ()
{
  FILE *fp;
  char str[] = "This is Computer Programming";

  fp = fopen( "file.txt" , "w" );
  fwrite(str , 1 , sizeof(str) , fp );

  fclose(fp);

  return(0);
}
```

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

```
This is Computer programming
```

Now let's 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

```
This is Computer Programming
```

Fseek() and Random Access I/O:
We can perform random read and write operations using the **C I/O** system with the help of **fseek().**
The C library function

int fseek(FILE *stream, long int offset, int whence)

sets the file position of the **stream** to the given **offset**.

Declaration

Following is the declaration for fseek() function.

```
int fseek(FILE *stream, long int offset, int whence)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **offset** − This is the number of bytes to offset from whence.
- **whence** − This is the position from where offset is added. It is specified by one of the following constants −

Constant	**Description**
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

Return Value

This function returns zero if successful, or else it returns a non-zero value.

Example

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

```
#include <stdio.h>

int main ()
{
  FILE *fp;

  fp = fopen("file.txt","w+");
  fputs("This is Computer Programming", fp);

  fseek( fp, 7, SEEK_SET );
  fputs(" C Programming Language", fp);
  fclose(fp);

  return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content. Initially program creates the file and writes *This is Computer Programming* but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content −

```
This is C Programming Language
```

Now let's 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

```
This is the C Programming Language
```

fprintf():
The C library function

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

sends formatted output to a stream.
Declaration
Following is the declaration for fprintf() function.

```
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**.

Return Value
If successful, the total number of characters written is returned otherwise, a negative number is returned.
Example
The following example shows the usage of fprintf() 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", 2015);

  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 2015
```
Now let's 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 above program to produce the following result.

```
We are in 2015
```

fscanf():

The C library function **int fscanf(FILE *stream, const char *format, ...)** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(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 one or more of the following items − *Whitespace character, Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example

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

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


int main()
{
  char str1[10], str2[10], str3[10];
  int year;
  FILE * fp;

  fp = fopen ("file.txt", "w+");
  fputs("We are in 2012", fp);

  rewind(fp);
  fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

  printf("Read String1 |%s|\n", str1 );
  printf("Read String2 |%s|\n", str2 );
  printf("Read String3 |%s|\n", str3 );
  printf("Read Integer |%d|\n", year );
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2012|
```

The Standard Streams:

In computer programming, **standard streams** are preconnected input and output communication channels between a computer program and its environment when it begins execution.

The three I/O connections are called

1. **standard input** (**stdin**),
2. **standard output** (**stdout**) and
3. **standard error** (**stderr**).

Originally I/O happened via a physically connected system console (input via keyboard, output via monitor).

standard input (**stdin**)

The System class provides a stream for reading text--the standard input stream.

Standard Output and Error Streams

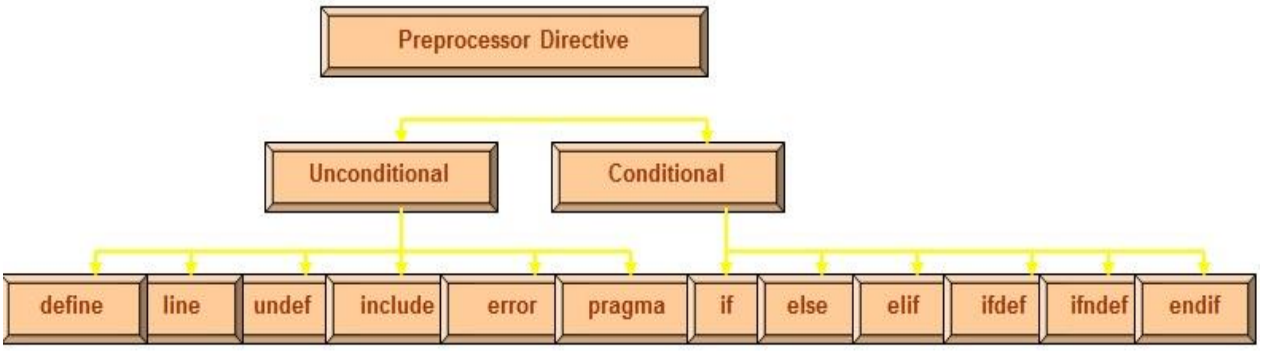
Probably the most often used items from the System class are the the standard output and standard error streams, which you use to display text to the user. The standard output stream is typically used for command output, to display the results of a command to the user. The standard error stream is typically used to display any errors that occur when a program is running.

THE PREPROCESSOR DIRECTIVE

- The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of preprocessor directive which is placed in the source program before the main().
- Before the source code is passed through the compiler, it is examined by the preprocessor for any preprocessor directives. In case, the program has some preprocessor directives, appropriate actions are taken (and the source program is handed over to the compiler.
- The preprocessor directives are always preceded by a hash sign (#).
- The preprocessor is executed before the actual compilation of program code begins. Therefore, the preprocessor expands all the directives and take the corresponding actions before any code is generated by the program statements.
- No semicolon (;) can be placed at the end of a preprocessor directive.

The advantages of using preprocessor directives in a C program include:

- Program becomes readable and easy to understand
- Program can be easily modified or updated
- Program becomes portable as preprocessor directives makes it easy to compile the program in different execution environments
- Due to the aforesaid reason the program also becomes more efficient to use.



<u>#define</u>

- To define preprocessor macros we use #define. The #define statement is also known as macro definition or simply a macro. There are two types of macros- object like macro and function like macro.

<u>Object like macro</u>

- An *object-like macro* is a simple identifier which will be replaced by a code fragment. They are usually used to give symbolic names to numeric constants. Object like macros do not take ant argument. It is the same what we have been using to declare constants using #define directive. The general syntax of defining a macro can be given as:

```
#define identifier string
```

- The preprocessor replaces every occurrence of the identifier in the source code by a string.

```
#define PI 3.14
```

Function-like macros

- They are used to stimulate functions.
- When a function is stimulated using a macro, the macro definition replaces the function definition.
- The name of the macro serves as the header and the macro body serves as the function body. The name of the macro will then be used to replace the function call.
- The function-like macro includes a list of parameters.
- References to such macros look like function calls. However, when a macro is referenced, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments, and the text is inserted into the program stream. Therefore, macros are considered to be much more efficient than functions as they avoid the overhead involved in calling a function.
- The syntax of defining a function like macro can be given as

```
# define identifier(arg1,arg2,...argn) string
```

- The following line defines the macro MUL as having two parameters a and b and the replacement string (a * b):

```
#define MUL(a,b) (a*b)
```

- Look how the preprocessor changes the following statement provided it appears after the macro definition.

```
int a=2, b=3,c;
c = MUL(a,b);
```

#include

- An external file containing function, variables or macro definitions can be included as a part of our program. This avoids the effort to re-write the code that is already written.
- The #include directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.
- The #include directive can be used in two forms. Both forms makes the preprocessor insert the entire contents of the specified file into the source code of our program. However, the difference between the two is the way in which they search for the specified.

#include <*filename*>

- This variant is used for system header files. When we include a file using angular brackets, a search is made for the file named *filename* in a standard list of system directories.

#include "*filename*"

- This variant is used for header files of your own program. When we include a file using double quotes, the preprocessor searches the file named *filename* first in the directory containing the current file, then in the quote directories and then the same directories used for <*filename*>.