UNIT IV: Polymorphism and file operations

9. Polymorphism

Definition, early Binding, Polymorphism with pointers, Virtual Functions, late binding, pure virtual functions

(Including function overloading and operator overloading)

10. Working with files

Header file, redirection, Classes for File Stream Operations - Opening and Closing a File - End-of-File Detection - file input and output. File Pointers - Updating a File - Error Handling during File Operations - Command-line Arguments, buffers & iostreams

---

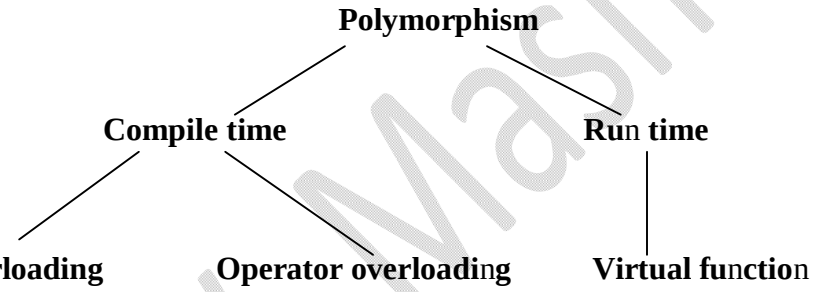
Chapter 9

Polymorphism

Polymorphism means performing different activities using same object or utility.

OR

Polymorphism means an ability to take more than one form using same utility.



1. Compile time polymorphism:

- In this, the overloaded member functions have difference in case of the type of parameters and number of parameters. This information is known to the compiler at the compile time.
- Hence the compiler is able to select the appropriate function at compile time itself. This is known as compile time polymorphism.
- It is also known as early binding.

Example: function overloading, operator overloading

2. Run time polymorphism:

- If the appropriate member function is selected while the program is running is known as run time polymorphism.
- It is also known as dynamic binding or late binding.

Example: virtual function

Jyoti V. Mashalkar

i) Function Overloading

- This is an example of compile time polymorphism.
- When two or more functions share the same name but their parameter declarations are different. Then the functions that share the same name are said to be overloaded and the process is known as **function overloading.**
- The two functions differing in their return types can not be overloaded because they do not provide sufficient information for the compiler to decide which function to use.
- While overloading a function, the type or number of arguments must differ that is:

i) Number of parameters

```
int add(int,int);
int add(int,int,int);
```

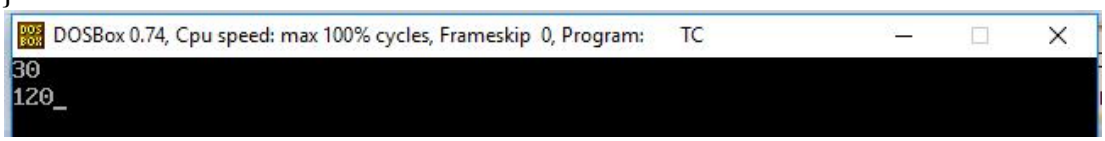
ii) Type of parameters

```
int add(int,float);
int add(int,int);
```

Example:

//Program for function overloading

```
#include<iostream.h>
#include<conio.h>
class function_overload
{
public:
int add(int p,int q)
{
return(p+q);
}
int add(int x,int y,int z)
{
return(x+y+z);
}
};
void main()
{
clrscr();
function_overload obj;
cout<<obj.add(10,20)<<endl;
cout<<obj.add(30,40,50);
getch();
}
```



In the above program,

The first add() function performs the addition of two integer values and returns an integer value as a result.

The second add() function performs the addition of three integer values and returns an integer value as a result. Here, the add() function is overloaded,

Jyoti V. Mashalkar

ii) Operator overloading

- This is an example of compile time polymorphism.
- In C++, we can change the way operators work for user defined types like objects and structures. This is known as operator overloading.
- To overload an operator, we can use a special operator function provided in C++. This function can be defined inside the class as follows:

Syntax:

```
class class_name
{
        return_type operator symbol (arg_list)
        {
                …..
        }
};
```

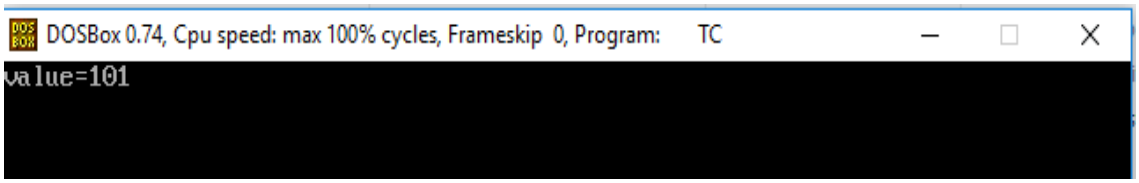
where,

return_type - is the return type of the function
operator- is a keyword
symbol - is the operator we want to overload like ++,-- etc.
arg_list- is the list of arguments passed to the function

//Program for operator overloading of unary ++ operator

```
#include<iostream.h>
#include<conio.h>
class count
{
int value;
public:
count()
{
value=100;
}
void operator++()
{
++value;
}
void display()
{
cout<<"value="<<value;
}
};
void main()
{
count c;
c++;
c.display();
getch();
}
```

Jyoti V. Mashalkar



➢ **Polymorphism with pointers using virtual function**

- Run time polymorphism is achieved using concept of virtual function.
- When we use the same function prototype in both the base and derived class then function in base class is declared as "virtual". So that, the function is not inherited in derived class but redefined by the derived class.

- " A virtual function is a member function that is declared within a base class and redefined by a derived class".

Syntax:

virtual return_type function_name(arg_list)

To execute this function, we have to create the object of base class. This object must be pointer object. This object can access both the base & derived class members.

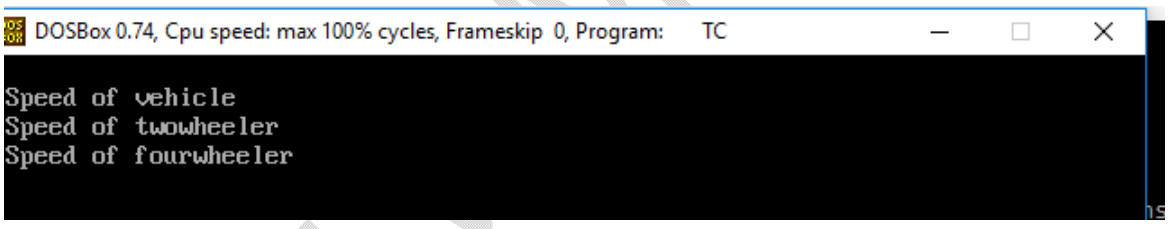
Example:

//Program for virtual function

```
#include<iostream.h>
#include<conio.h>
class vehicle
{
public:
virtual void speed()
{
cout<<"\n Speed of vehicle";
}
};
class twowheeler:public vehicle
{
public:
void speed()
{
cout<<"\n Speed of two wheeler";
}
};
```

Jyoti V. Mashalkar

```
class fourwheeler:public vehicle
{
public:
void speed()
{
cout<<"\n Speed of fourwheeler";
}
};
void main()
{
clrscr();
vehicle v,*p;
p=&v;
p->speed();
twowheeler t;
fourwheeler f;
p=&t;
p->speed();
p=&f;
p->speed();
getch();
}
```



- In the above example, the virtual function speed() is declared inside the base class vehicle.
- When the speed() is redefined by both twowheeler and fourwheeler. The virtual keyword is not needed.
- speed() is redefined in each class relative to that class.
- Four objects are created

Name	**type**
p	base pointer object
v	object of vehicle
t	object of twowheeler
f	object of fourwheeler

- p is assigned the address of v and speed() is called through p. As p stores the address of v, speed() in vehicle class gets executed.
- Next, p is set to the address of t and again speed() is called using p. This time p points to an object of class “twowheeler”. Hence, speed() of twowheeler class gets executed.
- Finally, p is assigned the address of f and p->speed() calls the speed () function of fourwheeler.

Jyoti Mashalkar

- The object to which p points determine which version of speed() is to be executed.
- So this process is known as run time polymorphism.
- As the base pointer can access the members of base as well as derived class but reverse is not true.

➢ **Pure virtual functions**

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder.

A "do nothing" function may be defined as follows:

virtual void disp()=0;

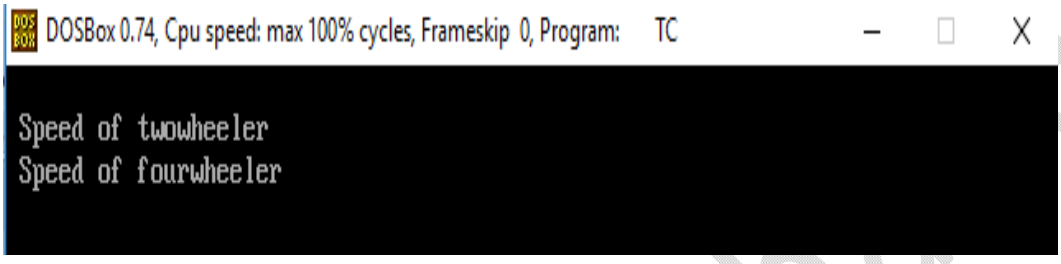
Such functions are called pure virtual functions. A pure virtual function is a function declared relative to the base class. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function.

Example:

```
//Program for pure virtual function
#include<iostream.h>
#include<conio.h>
class vehicle
{
public:
virtual void speed()=0;
};
class twowheeler:public vehicle
{
public:
void speed()
{
cout<<"\n Speed of twowheeler";
}
};
class fourwheeler:public vehicle
{
public:
void speed()
{
cout<<"\n Speed of fourwheeler";
}
};
```

Jyoti V. Mashalkar

```
void main()
{
clrscr();
twowheeler t;
fourwheeler f;
t.speed();
f.speed();
getch();
}
```



We have not defined any object of class “vehicle” & therefore the function speed() in the base class has been defined “empty”. Such functions are called “do nothing” functions.

Jyoti V. Mashalkar

Chapter 10. Working with files

- **Header file**

In C++, the header file used to gain access to the file handling functions is fstream.h and it is dealt with the help of three classes known as ofstream, ifstream and fstream.

- **ofstream class**: helps to create and write the data to the file and also known as output stream.
- **ifstream class:** helps to read data from files and also known as input stream.
- **fstream class**: is a combination of both ofstream and ifstream. It provides the capability of creating, writing and reading a file.

To access these classes, we must include the fstream as a header file as follows:

#include<fstream.h>

- **Redirection**

Sometimes, it is required to do some redirection of programs. Redirection can be from standard input, standard output or standard error.

Example:

1) To redirect the data file named "data1" into the program "prog1" then write as follows:

./prog1 < data1

Here less than symbol followed by filename that contains the data to send to the running program.

2) To save the output of the "prog2" into a file called "out2":

./prog2 > out2

3) To append data of "prog1" to the end of a file "out2", write as follows:

./prog2 >> out2

- **Classes for File Stream Operations**

Refer notes of point C++ Stream classes in chapter 1

- **Opening and Closing a File**

In C++, to open a file, first obtain a stream. There are the following three types of streams:

- input
- output
- input/output

i) Create an Input Stream

To create an input stream, declare the stream to be of class ifstream.

Example: ifstream fin;

ii) Create an Output Stream

To create an output stream, declare the stream to be of class ofstream.

Example: ofstream fout;

iii) Create both Input/Output Streams

To create a stream for performing both input and output operations, declare it of class fstream.

Example: fstream fio;

Jyoti V. Washalkar

1. Opening a file using open() function

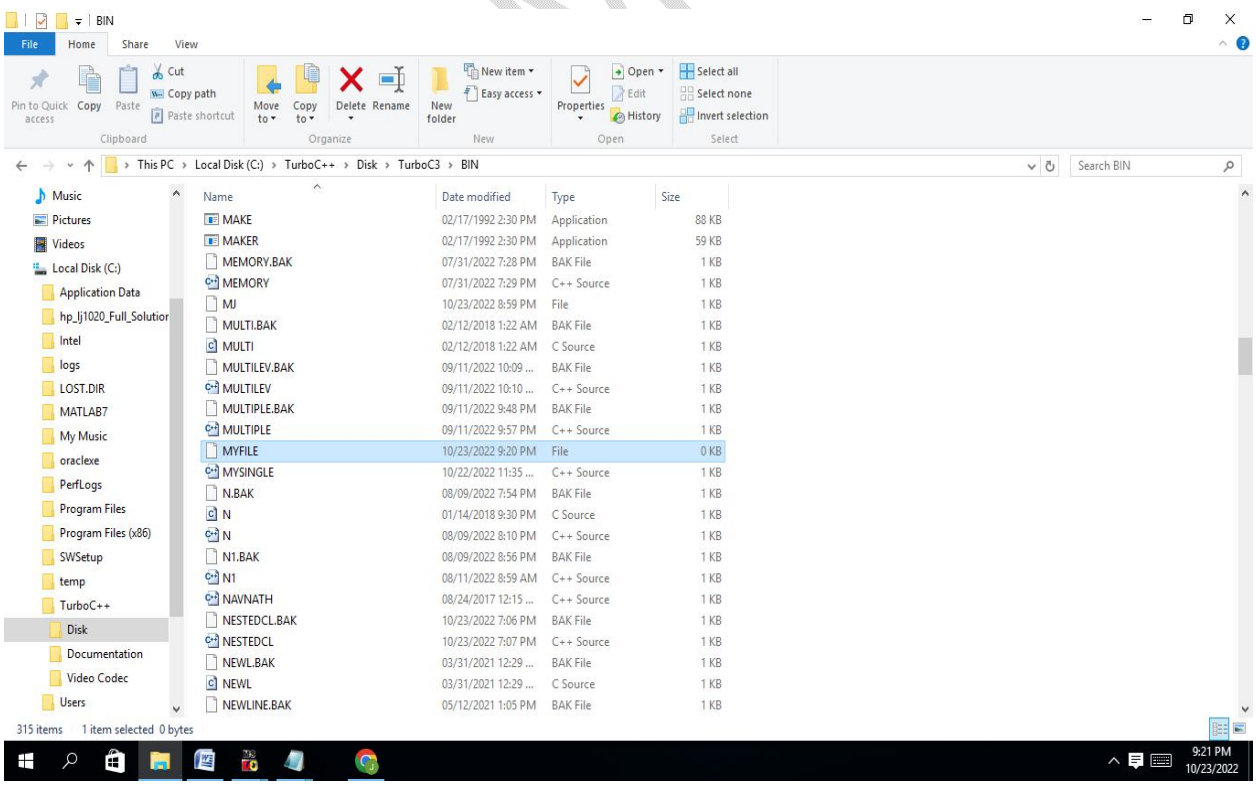
- Before performing any operation on a file, it is compulsory to first open it.
- To write the contents to the file, open it using fstream or ofstream object.
- To read from the file, open it using the fstream or ifstream object.
- Once a stream has been created, next step is to associate a file with it. And thereafter the file is available (opened) for processing.
- The three objects, of classes: fstream, ofstream, and ifstream, have the open() function definition.

Syntax: **stream_object.open (file_name, mode);**

- stream_object may be of object of one of class fstream, ofstream, or ifstream depending on the purpose.
- The file_name parameter denotes the name of the file to be open.
- The mode parameter is optional. It can take any of the following values:

Value	Description
ios:: app	The Append mode. The output sent to the file for append at the end.
ios::ate	It opens the file for the output then moves the read and write control to file's end.
ios::in	It opens the file for a read.
ios::out	It opens the file for a write.

It is possible to use two modes at the same time using the | (OR) operator.



2. Closing a file using close()

A file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form:

stream_object.close();

The fstream, ofstream, and ifstream objects have the close() function for closing files.

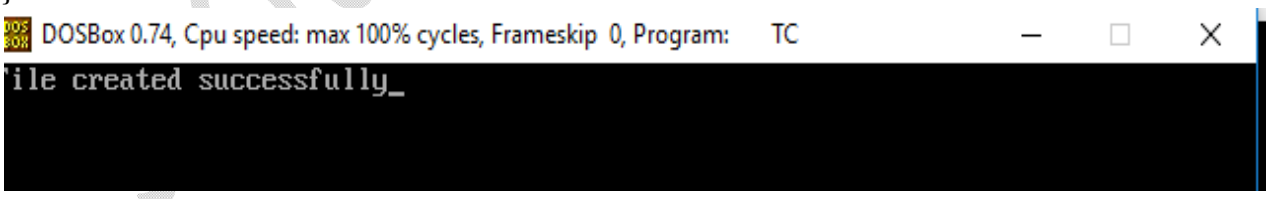
For example, if a file “myfile” is connected with an ofstream object fout, its connections with the stream fout can be terminated by the following statement :

fout.close() ;

Example:

//Program for opening and closing a file

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
clrscr();
fstream myfile;
myfile.open("myfile",ios::out);
if(!myfile)
{
cout<< "Error while creating the file";
}
else
{
cout<<"File created successfully";
myfile.close();
}
getch();
}
```



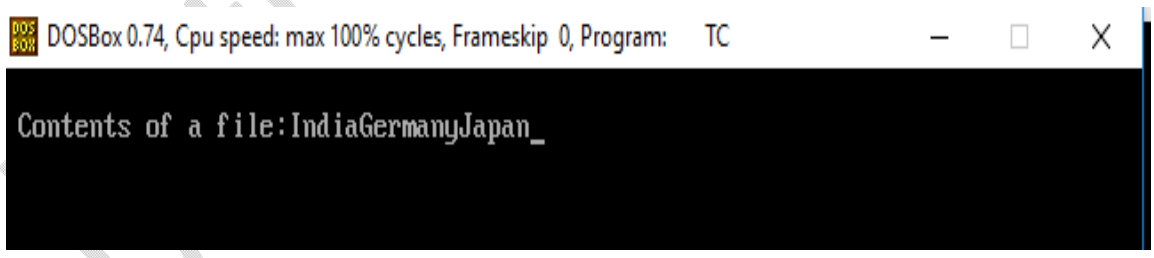
Jyoti V. Mashalkar

- **End-of-File Detection**

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file. This is demonstrated in the following program:

Example:

```
//Program to detect end-of-file
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{clrscr();
ofstream fout;
fout.open("country");
fout<<"India";
fout<<"Germany";
fout<<"Japan";
fout.close();
//Reading the file
char line[80];
ifstream fin;
fin.open("country");
cout<<"\n Contents of a file:";
while(fin)
{
fin.getline(line,80);
cout<<line;
}
fin.close();
getch();
}
```



A ifstream object that is fin in the above program, returns a value 0 if any error occurs in the file operation including the end-of-file condition. Thus, the while loop terminates when fin returns a value of zero on reaching the end-of-file condition.

There is another way to detect end-of-file condition.

```
if(fin1!=0)
{
exit(1);
}
```

Prof. V. Mashalkar

- **File input and output**

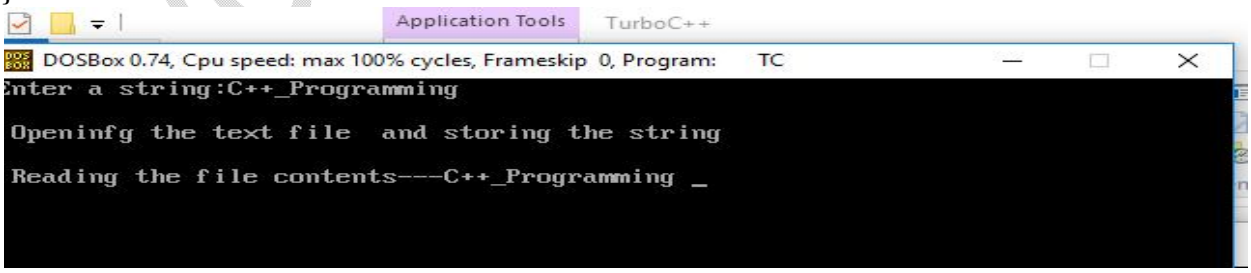
  The file stream classes support a number of member functions for performing the input and output operations on files.

  Example: put() and get(), write () and read()

put()	get()
writes a single character to the associated stream.	reads a single character from the associated stream.

//Program for put() and get() functions

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<fstream.h>
void main()
{
char str[80];
clrscr();
cout<<"Enter a string:";
cin>>str;
int len=strlen(str);
fstream file;
cout<<"\n Openinfg the text file  and storing the string \n";
file.open("TEXT",ios::in|ios::out);
for(int i=0;i<len;i++)
file.put(str[i]);
file.seekg(0);
char ch;
cout<<"\n Reading the file contents---";
while(file)
{
file.get(ch);
cout<<ch;
}
getch();
}
```



- **File Pointers**

  Each file has two associated pointers known as the file pointers. One of them is called input pointer (or get pointer) and the other is called the output pointer (or put pointer).

Jyoti V. Mashalkar

We can move these pointers to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place; the appropriate pointer is automatically advanced.

i) Default actions:
When we open a file in read-only mode, the input pointer is automatically set at the beginning so that we can read the file from the start.
Similarly, when we open a file in write-only mode, the existing pointer is set at the ending.

Example:
If we want to open an existing file to add more data, then file is opened in append mode. This moves the output pointer to the end of the file.

ii) Functions for manipulation of file pointers
In order to move a file pointer to any other desired position inside the file, the file stream classes support the following functions to manage such situations:

Function Name	Purpose
seekg()	Moves get pointer(input) to a specified location
seekp()	Moves put pointer(output) to a specified location
tellg()	Gives the current position of the get pointer
tellp()	gives the current position of the put pointer

Example:

```
file.seekg(10);
```

It will move the file pointer to the byte number 10. The bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11^{th} byte in the file.

Example: Consider the following statements:

```
ofstream fileout;
fileout.open("hello",ios::app);
p=fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of the file "hello" and the value of p will represent the number of bytes in the file.

iii) Specifying the offset
There are two functions used in C++ to move a file pointer to a desired location. These are:
seekg () and seekp().

The syntax of these functions is as follows:

seekg(offset, refposition);
seekp(offset, refposition);

Jyoti V Mashalkar

where,

- The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.
- The refposition takes one of the following three constants defined in the ios class:

a) ios::beg	**Start of the file**
b) ios::cur	**Current position of the pointer**
c) ios::end	**End of the file**

The seekg() function moves the associated files "get" pointer while the seekp() function moves the associated files "put" pointer.

- **Updating a File**

Updating is a routine task in the maintenance of any data files. The updating would include one or more of the following tasks:

- Displaying the contents of a file
- Modifying an existing item
- Adding a new item
- Deleting an existing item

These actions require the file pointers to move to a particular location that correspondence to the item/object under consideration. This can be easily implemented if the file contains a collection of items/objects of equal lengths. In such cases, the size of each object can be obtained using the statement:

int object_length=sizeof(object);

Then the location of a desired object may be obtained as follows:

int location=m*object_length;

The location gives the byte number of the first byte of the m^{th} object. Now we can set the file pointer to reach this byte with help of **seekg() and seekp().**

We can also find out the total of objects in a file using the object_length as follows:

int n=file_size/object_length;

The file_size can be obtained using the function tellg() or tellp() when the file pointer is located at the end of the file.

- **Error handling during file operations**

The errors may occur during file handling such as:

1. A file which we are attempting to open for reading does not exist.
2. The file name used for a new file may already exist.
3. We may attempt an invalid operation such as reading fast the end of file.
4. There may not be any space in the disk for storing more data.
5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

Jyoti V. Mashalkar

The C++ file stream inherits a 'stream-state' member from the class ios. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of the error conditions stated above.
The class **ios** supports several member functions that can be used to read the status recorded in a file stream.
These functions along with their meanings are listed below:

Function	Meaning
eof()	Returns true (non zero value) if end-of-file is encountered while reading. Otherwise returns false(zero value).
fail()	Returns true when an input or output operation has failed.
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is false, it may be possible to recover from any other error reported and continue operation.
good()	Returns true if an error has occurred. This means all the above functions are false. **Example:** If file.good() is true, we can proceed to perform I/O operations

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby to take necessary corrective measures.

- **Command-line Arguments**

- When the main function of a program contains arguments and these arguments are passed from command line then such arguments are known as command line arguments.
- This concept is used to control the program from outside.
- To support command line argument cocept, write the main() function as follows:

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

Here,

- The argc counts the number of arguments. It counts the file name as the first argument.
- The argv[] is an argument vector which contains the values of arguments passed.
The name of the program is stored in argv[0], the first command line argument in argv[1], and the last argument in argv[n].

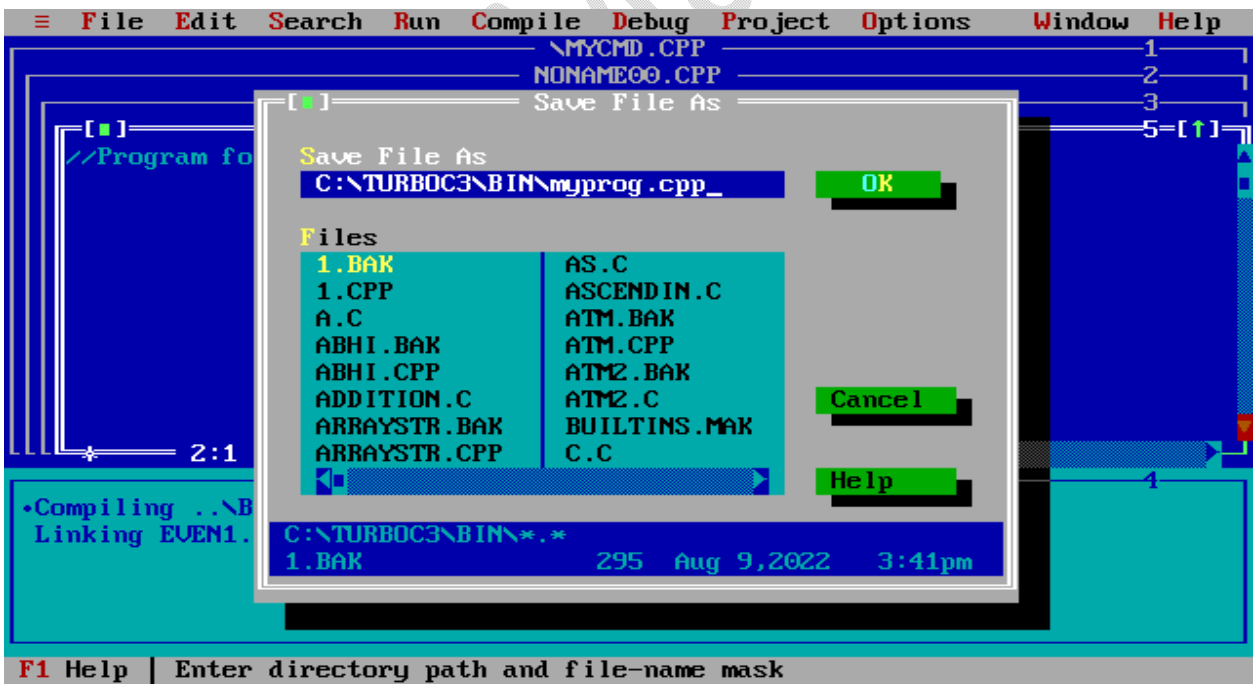
Jyoti V. Mashalkar

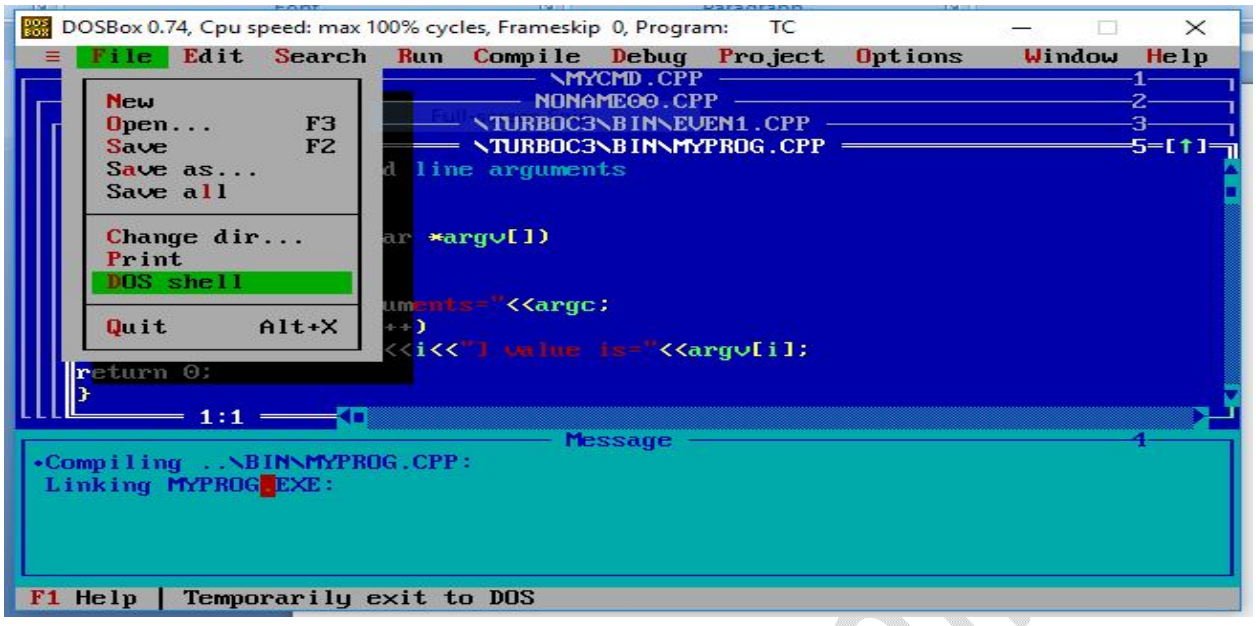
Steps to follow for command line arguments program:

Step 1: Type the program and save with the name “myprog.cpp”

//Program for command line arguments

```
#include<iostream.h>

#include<conio.h>

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

{

clrscr();

cout<<"Number of arguments="<<argc;

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

cout<<"\n Argument["<<i<<"] value is="<<argv[i];

return 0;

}
```

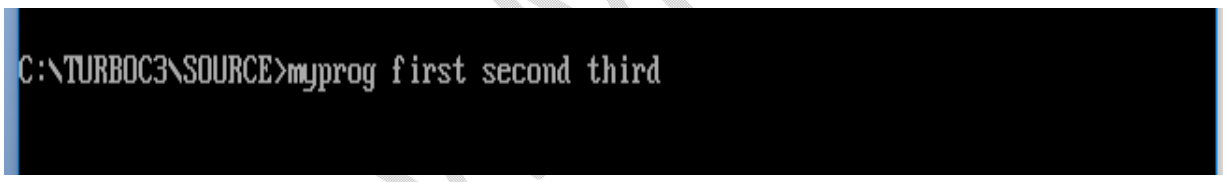




Step 2: Compile the program by F9 or click on Compile menu→ Select Make option. It will start linking.

Step3: Go to command prompt by selecting File Menu→DOS Shell

Step 4: Change the directory and pass the command line arguments while executing the program as shown below:



Step 5: It will show output as follows: