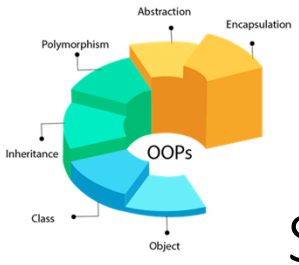
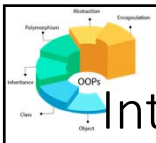


Stream Computation for Console and File Input/Output

Chapter 7

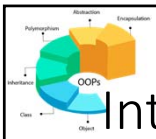
Object Oriented Programming

By DSBaral



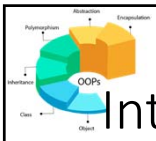
Introduction

- Every program takes some data as input, process it and produce output.
- In most of the languages there are ways of taking input from standard input device (normally keyboard) and displaying output on standard output device (normally monitor).
- Until now, we are using `cin` and `cout` with >> and << operators respectively for input and output.
- There are varieties of ways of input and output in C++.
- The C++ provides stream library for performing input/output operation with console, file and other devices.



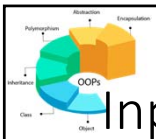
Introduction (Contd…)

- A stream is a general name given to a flow of data.
- Streams are classified into input and output streams.
- A stream is a series of bytes which act either as a source from which input data can be extracted or as a destination to which the output can be sent.
- All streams behave in the same way even though the device they refer to may be different.
  - We use the same technique to write to a file or to pointer or to screen.
- The input/output operations in C++ are supposed to operate on wide variety of devices like console, disks, printers or even network.



Introduction (Contd…)

- Usually, input/output facilities are designed to handle built in data types.
- The input/output facility of C++ allows programs to use built in as well as user defined types.
- The stream input/output facilities in C++ is designed and implemented to handle user defined types as well.
- We can overload stream operators << or >> to work with objects of classes we create.
  - This makes our classes work in the similar fashion as that of standard data types.



Input/Output Stream Class Hierarchy

- The iostream of C++ is used to perform various input/output activities.
- The stream class hierarchy for console input/output is presented below.

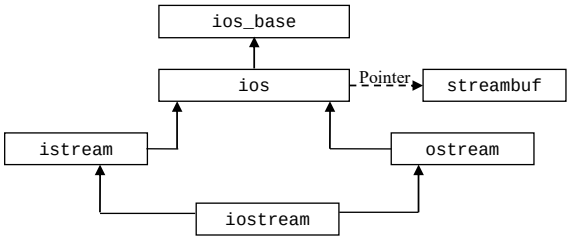
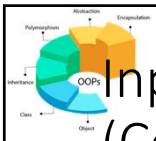
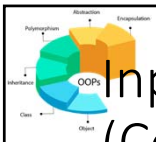


Figure: Stream class hierarchy for console input/output



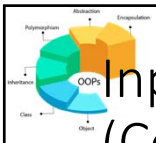
Input/Output Stream Class Hierarchy (Contd…)

- The object `cout` that represents the standard output device (video display) is a predefined object of `ostream` class.
- Similarly, the object `cin` that represents the standard input device (keyboard) is a predefined object of `istream` class.
- There are also other predefined objects `cerr` and `clog` of the `ostream` class.
- The insertion operator << is a member of `ostream` class and the extraction operator >> is a member of `istream` class.
- The stream classes used for console input and output are declared in the header file `<iostream>`.



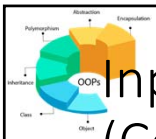
Input/Output Stream Class Hierarchy (Contd…)

- The base class (`basic_ios`) of the stream class hierarchies is defined as a template class.
- The ANSI/ISO C++ standard library creates two specialization of this input/output template class.
  - One of the specializations is for 8-bit characters and another for wide characters.
- Since the operation of 8-bit character and wide character is same, the discussion for 8-bit character will equally be applicable with wide character class.



Input/Output Stream Class Hierarchy (Contd…)

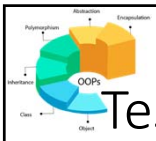
- The base class of the stream class hierarchy which is commonly used is `basic_ios`.
  - It is derived from `ios_base`, which defines several non template components used by `basic_ios`.
- The `basic_ios` is a high level input/output class that provides input/output processing with formatting, error checking, status information related with stream input/output.
- The `basic_ios` serves as a base class for several classes in the stream class such as `basic_istream`, `basic_ostream` and `basic_iostream`.



Input/Output Stream Class Hierarchy (Contd…)

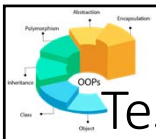
- The stream class template and their specialization for 8-bit character and wide character are given below

Stream class template	Specialization of 8-bit character	Specialization for wide character
`basic_streambuf`	`streambuf`	`wstreambuf`
`basic_ios`	`ios`	`wios`
`basic_istream`	`istream`	`wistream`
`basic_ostream`	`ostream`	`wostream`
`basic_iostream`	`iostream`	`wiostream`



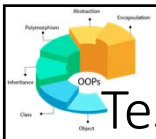
Testing Stream Errors

- The stream objects (`cin`, `cout`, etc.) works well in normal conditions for input and output.
- However, there can be several abnormal situations in I/O process such as entering string instead of digit, pressing enter key without entering value or some hardware failure.
- When errors are occurred during input or output operations, the errors are reported by stream state.
- Every stream (`istream` or `ostream`) has a state associated with it.
- Error conditions are detected and handled by testing the state of the stream.



Testing Stream Errors

- Following are the member functions of `ios` class that are used to test the state of the stream.
  - **`bool ios::good()`**: This function returns true when everything is okay, that is, when there is no error condition.
  - **`bool ios::eof()`**: This function returns true if the input operation reached end of input sequence.
  - **`bool ios::bad()`**: This function returns true if the stream is corrupted and no read/write operation can be performed.
    - For example an irrecoverable read error from a file.
  - **`bool ios::fail()`**: This function returns true if the input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.
    - When in fail condition the stream may not be corrupted.

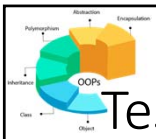


Testing Stream Errors (Contd…)

- **`void ios::clear(ios::iostate f=ios::goodbit)`**: This function clears all the flag if no argument is supplied.
  - It can also be used to clear a particular state flag if supplied as argument.
- **`ios::iostate ios::rdstate()`**: This function returns the state of a stream object of type `iostate`.
- **`void ios::setstate(ios::iostate f)`**: This function adds the flag in argument to the `iostate` flags.

- The state of the stream can also be detected by checking the stream itself as

```
if(streamobj){
   //No error with stream object streamobj
}
```

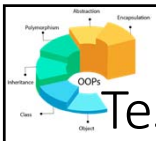


Testing Stream Errors (Contd…)

or

```
if(!streamobj){
   //Error with stream object streamobj
}
```

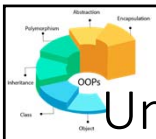
- The state of a stream is represented as a set of flags.
- These flags are defined in class `ios` as a static enumeration type `iostate`.
- The stream state flags are discussed below.
  - **`ios::goodbit`**: This state flag indicates that there is no error with streams. In this case the status variable has value 0.
  - **`ios::eofbit`**: This state flag indicates that the input operation reached end of input sequence.



Testing Stream Errors (Contd…)

- **`ios::badbit`**: This state flag indicates that the stream is corrupted and no read/write operation can be performed.
- **`ios::failbit`**: Indicates that input/output operation failed. The fail condition may not be because of corrupted stream.
- These stream flags can be directly used instead of status functions as

```
ios::iostate st=cin.rdstate();
if(st & ios::badbit){
   //input characters lost
}
//......
cin.setstate(ios::failbit);
        //set the stream state with failbit
```

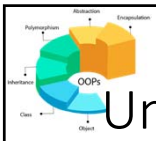


Unformatted Input/Output

- Along with the stream operators (<< and >>) and stream object (`cin` and `cout`), the stream classes `istream` and `ostream` defines different functions for unformatted input/output.
- **Functions `ostream::put()` and `istream::get()`**
  - The `put()` function is used to write a single character to the output device. It is a member function of class `ostream`. The function `put()` has following prototype:

    ```
    ostream& put(char c);
    ```
  - The `get()` function is used to read a single character or string from input device. It is a member of class `istream`. The function `get()` has following prototypes:

    ```
    istream& get(char& c);
    int get();
    ```
  - The `get()` function can take a white-space character like blank space, tab, and new line



Unformatted Input/Output (Contd…)

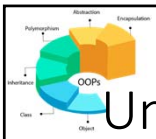
- The following example illustrates the use of `get()` and `put()`.

```
int main()
{   char c;
    cout<<"Enter a character:";
    cin.get(c);
    cout<<"\n The entered character is ";
    cout.put(c);
    return 0;
}
```

- The `get()` function to read a character is used as

```
c=cin.get(); //reads a single character and returns
```

- The function `get()` can also be used to read string.



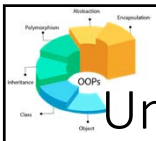
Unformatted Input/Output (Contd…)

- Unlike `cin` with extraction operator (`>>`) to read string, the `get()` function reads the string with embedded spaces and newline.
- The prototype of function get() to read string are

```
istream& get(char *s,streamsize n);
istream& get(char *s,streamsize n,char delim);
```

- Both of these function read at most *n*-1 character and places `'\0'` at the end of character array or until newline or delim character is reached.
- The `streamsize` is defined as some form of integer by the compiler.
- These functions leave delimiting characters in the stream.
- The function can be used as

```
cin.get(str1,MAX); //read string until MAX or new line character
cin.get(str2,MAX,'$'); //read string until MAX or $ character
```

Unformatted Input/Output (Contd…)

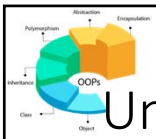
- **Functions `istream::getline()` and `ostream::write()`**
  - Like the function `get()`, the function `getline()` reads string with embedded space and even with newline character and places `'\0'` character at the end.
  - The function `getline()` is also a member of `istream` class.
  - The prototypes of `getline()` function are as follows:

```
istream& getline(char* s, streamsize n);
istream& getline(char* s, streamsize n, char delim);
```

  - The function `getline()` is used as follows:

```
cin.getline(str1,MAX); //read string until MAX or new line
cin.getline(str2,MAX,'$'); //read string until MAX or $
```

  - Unlike `get()`, the function `getline()` removes delimiter character from the stream.



Unformatted Input/Output (Contd…)

- The `write()` function is used to display number of characters specified by the user.
- The `write()` function is a member of ostream class.
- The prototype of `write()` function is

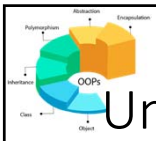
```
ostream& write(const char *s,streamsize n);
```

- While using the function `write()` the programmer should be aware not to exceed `n` then the length of the string.
- The function `write()` is used as follows.

```
char str[]="Beautiful Country Nepal";
cout.write(str,strlen(str)); //display whole string
```

- But the statement

```
cout.write(str,9); //display Beautiful
```

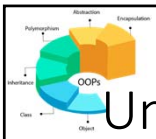
Unformatted Input/Output (Contd…)

- **Functions `istream::read()` and `istream::ignore()`**
  - The `istream` function `read()` reads specified number of characters into the string buffer.
  - The prototype of the `read()` function is

```
istream& read(char* s,streamsize n);
```

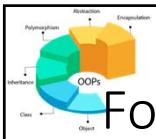
  - Unlike `get()` and `getline()` functions the function `read()` does not rely on delimiter and does not put the null character `'\0'` at the end of the string buffer; it reads `n` characters rather than `n-1` characters.
  - The `read()` function is used as follows

```
char str[10];
cin.read(str,9); //does not put null char at the end
str[9]='\0'; //null character added to make C style string
```



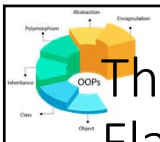
Unformatted Input/Output (Contd…)

- The `istream` function `ignore()` reads characters from input stream like `read()` function but does not store them.
- The function `ignore()` has following prototype
  ```
  istream& ignore(streamsize n=1,int delim=EOF);
  ```
- Like function `read()` the function `ignore()` actually extracts characters either `n` characters have been extracted or the extracted character equals to `delim`.
- The number of characters read by the function ignore() is one and delimiter is end-of-file (EOF) by default.
- The function ignore() is used as
  ```
  cin.ignore();
  ```
- One of the use of the function `ignore()` is to remove delimiter mostly newline from the stream after the function `get()` is called.



Formatted Input/Output

- In C++ there are various features to perform input or output in different formats, such as, amount of space for output operation, alignment of fields, the format used for output of numbers.
- For the formatted input/output, C++ provides following features
  - The `ios` stream class member functions and flags
  - Standard manipulators,
  - User defined manipulators.

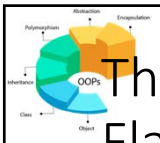


The `ios` Stream class Member Functions and Flags

- The `ios` stream class consists of varieties of functions for formatting the output in different ways.
- Some important member functions of `ios` class used for formatting are discussed below
- **`width()`**: This function of `ios` class is used to define the width of the field to be used while displaying the output.
  - It has the following two forms:

```
streamsize width();//returns current field width
streamsize width(streamsize wide);//sets field width
```

  - The `streamsize` type is defined as an integer type by the compiler.



The `ios` Stream class Member Functions and Flags (Contd…)

- The `width()` without argument returns the current field width setting and `width()` with argument sets the width to the specified integer value and returns previous field width value.
- For example the output of the following code segments is

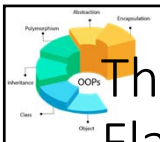
```
cout.width(6);
cout<<894;
```

			8	9	4

- The effect of the `width()` is for only one time and after that it resets to default (0) .
- The output of the following code segments is

```
cout.width(6);
cout<<8941;
cout<<49;
```

		8	9	4	1	4	9



The `ios` Stream class Member Functions and Flags (Contd…)

- To display both numbers with equal field widths we should write the code as follows

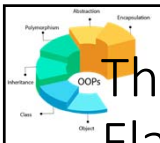
```
cout.width(6);
cout<<8941;
cout.width(6);
cout<<49;
```

```
    8 9 4 1     4 9
```

- If the field width specified is smaller than the required width then it does not have any effect.
- The output of the following code is

```
cout.width(3);
cout<<8941;
```

```
8 9 4 1
```

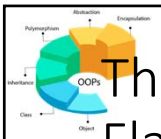
The `ios` Stream class Member Functions and Flags (Contd…)

`fill()`: This `ios` function is used to specify the character to be displayed in the unused portion of the display width; by default blank character is displayed.

- It has following two forms:

```
char fill();
char fill(char);
```

- The function `fill()` without argument returns current fill character where as `fill()` with character argument sets the fill character to be displayed in the unused portion of display area set by the function `width()` and returns the previous fill character.



The `ios` Stream class Member Functions and Flags (Contd…)

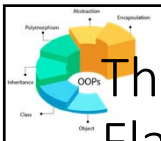
- For example, consider following code segment

```
int x=456;
cout.width(6);
cout.fill('#');
cout<<x<<endl;
```

- It will display output

```
###456
```

- The fill character remains as it is unless it is changed by next call to function `fill()`.

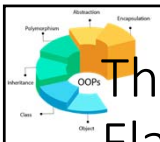


The `ios` Stream class Member Functions and Flags (Contd…)

- **`precision()`**: This `ios` function is used to specify maximum number of digits to be displayed as a whole in general format of floating point number or the maximum number of digits to be displayed in the fractional part of exponential or normal floating point number.
  - It also the has two formats:

```
streamsize precision();
streamsize precision(streamsize n);
```

  - The `precision()` without argument returns the current floating point precision where as `precision()` with argument sets specified precision and returns previous precision value.



The `ios` Stream class Member Functions and Flags (Contd…)

- The precision for all formats is 6 by default.
- The output of the following code segment is

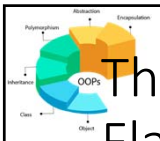
```
float x=5.5005,y=66.769;
cout.precision(3);
cout<<x<<endl;
cout<<y<<endl;
```

displays output as follows:

```
5.5
66.8
```

trailing zeros are not shown
rounding is done during display

- The precision value set by the `precision()` is not changed unless it is set again.

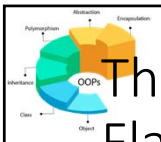


The `ios` Stream class Member Functions and Flags (Contd…)

- **`setf()`**: The `ios` member function `setf()` is used to set flags and bit-fields that controls the input/output formatting in other ways such as left justified, scientific form, show positive sign, show base of displayed number etc..
  - It has following two forms:

```
fmtflags setf(fmtflags flag, fmtflags field);
fmtflags setf(fmtflags flag);
```

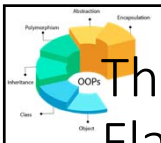
  - The `fmtflags` is a bitmask enumeration defined in `ios`.
  - The `flag` is one of the flags defined in the class `ios` and specifies the format action required for the output.
  - The `field` specifies the group to which the formatting flag belongs.
  - The `setf()` function sets new flag and returns the previous flag option.
  - The flag once set will remain as it is unless it is unset or new value is set.



The `ios` Stream class Member Functions and Flags (Contd…)

- Let's see the various combination of flag and bitfield values required by first form of `setf()` which requires two arguments.

Flag value	Bit field value	Effect on output
`ios::left`	`ios::adjustfield`	Justifies output to left
`ios::right`	`ios::adjustfield`	Justifies output to right
`ios::internal`	`ios::adjustfield`	Filling character is displayed between sign and number or base indicator and number
`ios::dec`	`ios::basefield`	Integer number is displayed in decimal
`ios::oct`	`ios::basefield`	Integer number is displayed in octal
`ios::hex`	`ios::basefield`	Integer number is displayed in hexadecimal
`ios::scientific`	`ios::floatfield`	Floating point number is displayed in exponential format
`ios::fixed`	`ios::floatfield`	Floating point number is displayed in normal integer part dot fractional part format



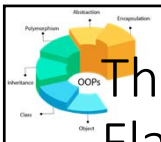
The `ios` Stream class Member Functions and Flags (Contd…)

- Since only one flag value is effective for one bit field, the two argument version of `setf()` resets the previous flag associated with the passed bit field and sets the new flag passed during function call.
- The following code segments:

```
int x=456;
cout.setf(ios::left,ios::adjustfield);
cout.width(6);
cout.fill('#');
cout<<x<<endl;
```

- display

```
456###
```



The `ios` Stream class Member Functions and Flags (Contd…)

- The following code segment

```
int x=255;
cout.setf(ios::oct,ios::basefield);
cout<<x<<endl;
```

- displays number as

```
377
```

- The code segment

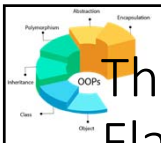
```
double x=123456789.0123;
cout.setf(ios::scientific,ios::floatfield);
cout<<x<<endl;
```

- displays number as

```
1.234568e+008
```

- With `ios::fixed` it is displayed as

```
123456789.012300
```

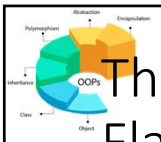
The `ios` Stream class Member Functions and Flags (Contd…)

- To revert back to general format, the function is called as

```
cout.setf(ios::fmtflag(0),ios::floatfield)
```

- Let's see various flag values of second form of `setf()` which requires one argument.

Flag value	Effect on output
`ios::showbase`	prefix `0` on octal number and `0x` on hexadecimal number to indicate the base of number.
`ios::showpoint`	show point and trailing zeros on fraction part .
`ios::showpos`	show + sign for positive number.
`ios::uppercase`	show letter in number in uppercase like x in 0x, e in exponential format and digits a-e in hexadecimal number.
`ios::skipws`	skip whitespace character on input
`ios::boolalpha`	display '`true`' or '`false`' instead of `1` or `0`.
`ios::unitbuf`	flush output stream after each output operation



The `ios` Stream class Member Functions and Flags (Contd…)

- The following code segment

```
int x=118;
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::showbase);
cout<<x;
```

- displays following output

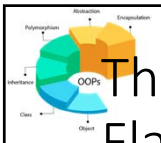
```
0x76
```

- The following code

```
double x=98.4012;
cout.precision(4);
cout.setf(ios::showpoint);
cout<<x;
```

- displays output as

```
98.40
```

The `ios` Stream class Member Functions and Flags (Contd…)

- **`unsetf()`**: This `ios` function is used to unset the specified flag value.
  - The function has the following form:

```
void unsetf(fmtflags flag);
```

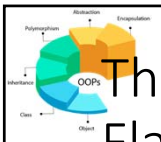
  - The following code

```
cout.unsetf(ios::showpos);
```

  - clears the format bit corresponding to `showpos`.
- **`flags()`**: This ios function is used to read and set the flag values.
  - The function has the following forms

```
fmtflags flags();
fmtflags flags(fmtflags flags);
```

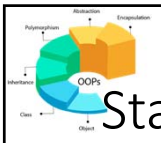
  - The first function returns the current format flag setting and second function sets the passed flag setting and return the previous format flag setting.



The `ios` Stream class Member Functions and Flags (Contd…)

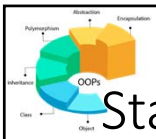
- The flag values can be set as
  ```
  cout.flags(ios::left|ios::oct|ios::fixed);
  ```
- Through the use of function `flags()`, a single flag value can be set as
  ```
  cout.flags(cout.flags()|ios::showpos);
  ```
- This statement is like
  ```
  cout.setf(ios::showpos);
  ```
- Once set, a flag continuously affects the output unless it is unset or changed by next value set.

- **`flush()`**: This `ostream` function flushes the output stream and send it to the real output destination.
  - It has the following form:
    ```
    ostream& flush();
    ```
  - It is used as
    ```
    cout.flush();
    ```

Standard Manipulators

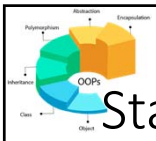
- To call `ios` function for formatting, we need to write separate statement and call the function through stream objects `cin` and `cout`.
- Manipulators are the formatting function for input/output that are embedded directly into the C++ input/output statements.
- The predefined manipulators are defined in header `<iostream>` and `<iomanip>`.
- Manipulators are used in following ways:
  ```
  cout<<manip1<<manip2<<manip3<<item1;
  cout<<manip1<<item1<<manip2<<item2<<item3;
  ```



Standard Manipulators (Contd…)

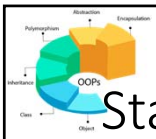
- Manipulators are of following types.
  - Non-parameterized Manipulators
  - Parameterized Manipulators
- Manipulators provide same functionality as that of `ios` class functions for formatting.

```
left
right
internal
dec
hex
oct
```

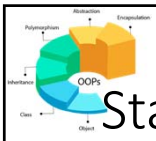
Standard Manipulators (Contd…)

```
scientific
fixed
showbase
noshowbase
showpoint
noshowpoint
showpos
noshowpos
uppercase
nouppercase
```



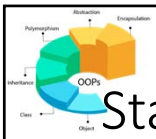
Standard Manipulators (Contd…)

```
skipws
noskipws
boolalpha
noboolalpha
unitbuf
nounitbuf
endl
ends
flush
ws
```

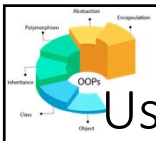
Standard Manipulators (Contd…)

- Following table shows different parameterized manipulators
  - `setw(int n)` ->equivalent to `ios` function `width()`
  - `setprecision(int n)` ->equivalent to `ios` function `precision()`
  - `setfill(int c)` ->equivalent to `ios` function `fill()`
  - `setbase(int b)` ->sets base for integer output, argument 0 is passed for decimal, 8 for octal, 10 for decimal and 16 for hexadecimal.
  - `setiosflags(ios::fmtflags f)` ->equivalent to `ios` function `setf()`
  - `resetiosflags(ios::fmtflags f)` ->equivalent to ios function `unsetf()`
- The header file `<iomanip>` must be included when using parameterized manipulators.



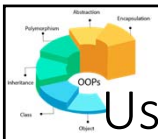
Standard Manipulators (Contd…)

- The manipulators without arguments are used as
  ```
  cout<<2357<<','<<hex<<2357<<','<<oct<<2357<<endl;
  ```
- The output will be displayed as
  ```
  2357, 935, 4465
  ```
- The statement
  ```
  cout<<setw(5)<<setfill('#')<<37<<endl;
  ```
- generates the output as
  ```
  ###37
  ```

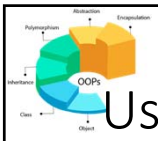
User Defined Manipulators

- In C++, similar to the predeined built in manipulators user can define non parameterized as well as parameterized manipulators.
- The syntax for designing non parameterized manipulator is as follows
  ```
  ostream& manipulator_name(ostream& output)
  {
     //body of user defined manipulator
     return output;
  }
  ```
- Let's see the following example for creating and using nonparameterized user defined manipulator.



User Defined Manipulators (Contd…)

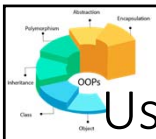
```
ostream& sp(ostream& os)
{
    os<<' '<<flush;
    return os;
}
int main()
{
    int a=1,b=2,c=3,d=4;
    cout<<a<<sp<<b<<sp<<c<<sp<<d<<endl;
    return 0;
}
```

User Defined Manipulators (Contd…)

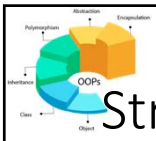
- Following example illustrates user defined parameterized manipulator

```
class testm {
private:
    int n;
public:
    testm(int num):n(num){}
    friend ostream& operator<<(ostream& os,testm& tm);
};
ostream& operator<<(ostream& os, testm& tm){
    for(int i=0;i<tm.n;++i)
        os<<' ';
    os<<flush;
    return os;
}
```



User Defined Manipulators (Contd…)

```
testm sp(int n)
{
    return testm(n);
}
int main()
{
    int a=1,b=2,c=3,d=4;
    cout<<a<<sp(1)<<b<<sp(2)<<c<<testm(3)<<d<<endl;
    return 0;
}
```

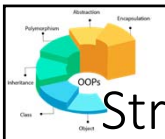



Stream Operator Overloading

- The standard C++ allows us to perform input/output operation with user defined type in the same way as basic data types.
- The user defined types cannot be read with following statement

```
complex c;
cin>>c; //error
```

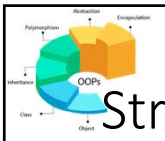
- To do input/output operation of user defined types similar to basic types, stream operators extraction (>>) and insertion (<<) need to be overloaded.
- After overloading the stream operators we can perform input/output operation similar to basic data types as above.



Stream Operator Overloading (Contd…)

- The syntax of insertion operator overloading is as follows:

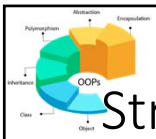
```
ostream& operator<<(ostream& os, myclass& myobj)
{
   //......
   return os;
}
istream& operator>>(istream& is, myclass& myobj)
{
   //......
   return is;
}
```

Stream Operator Overloading (Contd…)

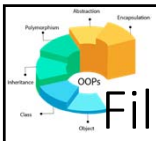
- Following example illustrates the concept of stream operator overloading.

```
class Time
{
private:
   int hour,min,sec;
public:
   Time(int hr=0,int mn=0,int sc=0)
   {hour=hr;min=mn;sec=sc;}
   friend ostream& operator<<(ostream& os, Time& tm);
   friend istream& operator>>(istream& is, Time& tm);
};
```



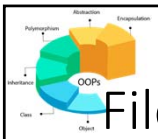
Stream Operator Overloading (Contd…)

```
ostream& operator<<(ostream& os, Time& tm){
   os<<tm.hour<<":"<<tm.min<<":"<<tm.sec<<flush;
   return os;
}
istream& operator>>(istream& is, Time& tm){
   is>>tm.hour>>tm.min>>tm.sec;
   return is;
}
int main()
{   Time t1;
    cout<<"Enter current time: ";
    cin>>t1;
    cout<<"Time entered is: "<<t1<<endl;
}
```

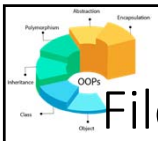
File Input/Output with Streams

- When programs require a large amount of data to be read, processed, and also saved for later use then in such case the console input/output does not solve the program.
- In such case we need to store the information in auxiliary memory device in the form of data file.
  - A file is a collection of bytes that is given a name.
- Like the console input/output, C++ provides file stream library for file manipulation.
- File manipulation is done through stream classes `ifstream`, `ofstream` and `fstream`.
- The file stream classes are declared in file `<fstream>`, so, it is to be included in a program for file processing.



File Stream Class Hierarchy

- The data files in C++ are handled in the form of stream objects similar to `cin` and `cout`.
- Objects of file stream class are created and used in file handling.
- The class `ifstream` is used for handling input files, class `ofstream` for handling output files and the class `fstream` is used for handling files on which both input and output can be performed.
- The file stream class `ifstream` is derived from stream class `istream`, the file stream class `ofstream` is derived from stream class `ostream` and the file stream class `fstream` is derived from stream class `iostream`.



File Stream Class Hierarchy (Contd…)

- The stream class hierarchy including file stream is shown in the following figure.

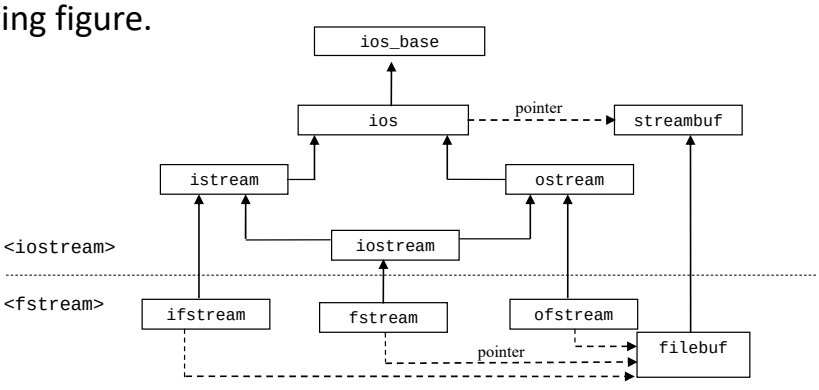
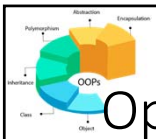
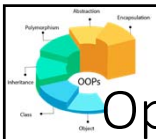


Figure: Stream class hierarchy for file input/output



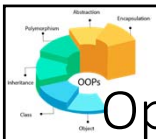
Operations on Files

- There are several operations related to files such as opening, closing, reading, writing, appending or checking errors.
- For every types of file operation, the first thing is to open a file and after completion of each process it is closed.
- The file manipulation involves following processes:
  - Naming a file on the disk
  - Opening the file
  - Processing the file (reading/writing/manipulating)
  - Checking errors
  - Closing the file after its use



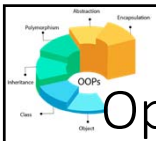
Opening and Closing files

- For file handling, a file is opened by either of two ways.
  - by the constructor function of the file stream class
  - by using member function `open( )` of the same class.
- A file is opened in either read, write, append or other modes.
- The file can also be closed by two ways.
  - by the destructor of the file stream class automatically
  - by using the member function `close( )` of the file stream class.



Opening and Closing files (Contd…)

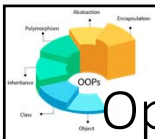
- **Opening and Closing Files using Constructor and Destructor**
  - Using constructor, a file can be opened by passing a filename as an argument to the constructor
  - To open a file in output mode using constructor we write as
    ```
    ofstream outfile("myfile.txt");
    ```
  - We can write data to this file using stream operator as
    ```
    outfile<<"Text to be written to file";
    ```
  - Similarly, to open a file in input mode using constructor we write as
    ```
    ifstream infile("data.dat");
    ```
  - Reading from this file can be done as
    ```
    int age;
    infile>>age; //There should be numbers in the file
    ```

Opening and Closing files (Contd…)

- The constructors are declared as
  ```
  explicit ofstream(const char *path, openmode mode=ios::out);
  explicit ifstream(const char *path, openmode mode=ios::in);
  explicit fstream(const char *path, openmode mode=ios::in|ios::out);
  ```
- The file closing is done automatically by the destructor.

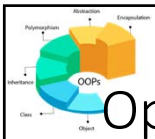
- **Opening and Closing Files Explicitly**
  - Apart from constructor, file can be opened explicitly by making use of member function `open()`.
  - The prototype of the `open()` member function is similar to the constructor
    ```
    void ofstream::open(const char *path, openmode mode=ios::out);
    void ifstream::open(const char *path, openmode mode=ios::in);
    void fstream::open(const char *path,openmode mode=ios::in|ios::out);
    ```



Opening and Closing files (Contd…)

- The function `open()` is used to open file in write mode as follows:

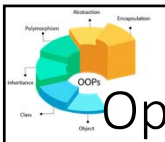
```
ofstream fout;
fout.open("data.txt");
//some operations
//......
fout.close();
      //closes the file first before opening different file
fout.open("mydata.doc");
      //opens another file by the same object fout
```

Opening and Closing files (Contd…)

- **File Modes**
  - Apart from file name, the file opening functions (constructor or `open()` function) require another argument to specify filemode which has default value.
  - With `ifstream` class default mode is input and with `ofstream` class the default mode is output, and with `fstream` class the default mode is both input and output.
  - But we can also explicitly specify the mode while opening the file. The format for specifying file mode is as follows:

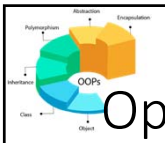
```
fstream stobj1;
stobj1.open("filename",mode);
fstream stobj2("filename",mode);
//......
```



Opening and Closing files (Contd…)

- File modes are defined as bit mask enumeration type in class `ios`.
- Different file modes are discussed as follows:
  - **`ios::in`**: This mode opens a file for reading.
  - **`ios::out`**: This mode opens a file for writing.
  - **`ios::ate`**: This mode sets the file access pointer is set at the end of file.
  - **`ios::app`**: When a file is opened in this mode, file is opened in write mode with the file access pointer at the end of file.
  - **`ios::trunc`**: When a file is opened in this mode, the file is truncated if a file with specified name already exists
  - **`ios::binary`**: When a file is opened in this mode, the file is opened as a binary file.
- When opening file, we can open file in more than one mode as

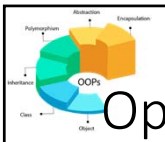
```
fstream mystream("filedata",ios::in|ios::out);
fstream mystream("mydata",ios::binary|ios::in);
```

Opening and Closing files (Contd…)

- Let's see the following example that copies a text file.

```
int main()
{
    char ch, ifn[25],ofn[25];
    cout<<"\n Enter input filename: ";cin>>ifn;
    cout<<"\n Enter output filename: ";cin>>ofn;
    ifstream infile;
    infile.open(ifn);
    if(!infile){
        cerr<<"Error opening: "<<ifn<<endl;
        exit(1);
    }
```



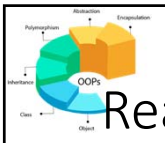
Opening and Closing files (Contd…)

```
    ofstream outfile(ofn);
    infile>>resetiosflags(ios::skipws);
    while(infile){
        infile>>ch;
        outfile<<ch;
    }
    return 0;
}
```

- The input filename and output filename can also be supplied from command line arguments as

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

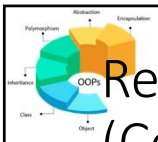
`argv[0]` -> filename, `argv[1]` -> first argument, `argv[2]` -> second argument



Read/Write from File with Member Functions

- For character input/output with file streams the member functions `get()` and `put()` can be used in the same way as the console input/output.
- The function `get()` is used to read a single character from the file and the function `put()` is used to write a single character to the file.
- The function `put()` is used as follows as follows:

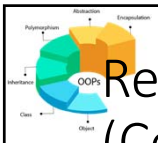
```
ofstream file("myfile.txt");
char var;
//.........
file.put(var);
```



Read/Write from File with Member Functions (Contd…)

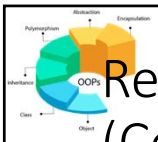
- The following program illustrates the use of these functions.

```
int main()
{
    char ch, str[50];
    fstream file;
    file.open("sample.txt",ios::out);
    cout<<"Enter a String: ";
    cin.getline(str,50);
    for(int i=0;str[i];i++)
        file.put(str[i]);
    file.close();
```

Read/Write from File with Member Functions (Contd…)

```
    file.open("sample.txt",ios::in);
    cout<<"\nString from file is: ";
    while(1)
    {
        file.get(ch);
        if(!file) break;
        cout<<ch;
    }
    return 0;
}
```



Read/Write from File with Member Functions (Contd…)

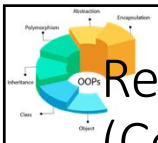
- To store and retrieve binary data, member functions `write()` and `read()` of are used respectively.
- The number of bytes required to represent an integer in character form depends on its value but the binary format does not depend on value.
- The member function `write()` is used as follows:

```
file_obj.write(reinterpret_cast<char*>(&variable),
   sizeof(variable));
```

- Similarly, the member function `read()` is used as follows:

```
file_obj.read(reinterpret_cast<char*>(&variable),
   sizeof(varible));
```

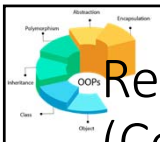
- Functions `read()` and `write()` take two arguments, first is the address of the variable and second is the size of variable.



Read/Write from File with Member Functions (Contd…)

- Following example illustrates how to write data in binary files

```
int main()
{
   int inum1=765,inum2;
   float fnum1=835.21,fnum2;
   ofstream outfile("num.bin", ios::binary);
   outfile.write(reinterpret_cast<char*>(&inum1),
      sizeof(inum1));
   outfile.write(reinterpret_cast<char*>(&fnum1),
      sizeof(fnum1));
   outfile.close();
```

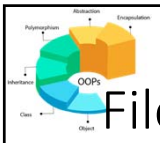


Read/Write from File with Member Functions (Contd…)

```
        ifstream infile("num.bin", ios::binary);
        infile.read(reinterpret_cast<char*>(&inum2),
            sizeof(inum2));
        infile.read(reinterpret_cast<char*>(&fnum2),
            sizeof(fnum2));
        cout<<"Data from file: "<<inum2<<", "<<fnum2;
        return 0;
    }
```

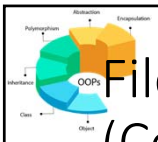
- Using `read()` and `write()` functions any object can we written to and read from file as

```
infile.read(reinterpret_cast<char*>(&obj),sizeof(obj));
outfile.write(reinterpret_cast<char*>(&obj),sizeof(obj));
```

File Access Pointers and their Manipulators

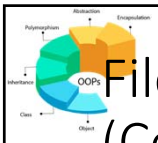
- In C++, there are two types of file access pointers, one for input called get pointer and another for output called put pointer.
- The get pointer facilitates the movement in the file while reading and put pointer facilitates the movement while writing.
- The get pointer specifies a location from where the next reading operation is performed.
- The put pointer specifies a location from where the next writing operation is performed.
- Initially, the file pointer are set to an appropriate location based on the mode in which file is opened.



File Access Pointers and their Manipulators (Contd…)

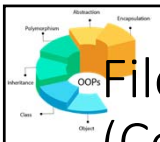
- In C++, there are functions for manipulation of file access pointers.
- The file access pointer manipulation functions are used to set a file access pointer to any desired position inside the file or to get the current file access pointer.
- The file access pointer position value specifies the byte number in the file where writing or reading will take place.
- The different functions for manipulation of file access pointers as follows
- **`seekg()`**: This function moves get file access pointer to a specific location.
  - It is a member of `ifstream` class.
  - The prototype is as follows:

```
istream& seekg(pos_type pos);
istream& seekg(off_type off,ios::seekdir dir);
```

File Access Pointers and their Manipulators (Contd…)

- The first function sets the access position from the beginning of a file and the second function sets it from a point indicated by `dir`.
- The `pos_type` is an integer type capable of holding a character position in a file from the beginning.
- The `off_type` is an integer type that is capable of holding the offset from a point indicated by `seekdir`.
- The `ios::seekdir` is an enumeration defined by `ios` that determines how seek will take place.
- The values for `ios::seekdir` can be
  - `ios::beg` to seek from the beginning of file
  - `ios::cur` to seek from current location
  - `ios::end` to seek from the end of file



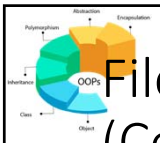
File Access Pointers and their Manipulators (Contd…)

- For example, the statements

```
char ch;
ifstream infile("myfile.txt",ios::binary);
infile.seekg(5);
infile.read(&ch,sizeof(ch));
infile.seekg(-3,ios::cur);
infile.read(&ch,sizeof(ch));
```

  reads a character from 5th character of the file "myfile.txt" and another character form 3rd position of the file because first read moves the file access pointer to 6th position.

- Attempting to seek beyond the beginning or end of a file puts the stream into the bad state.



File Access Pointers and their Manipulators (Contd…)

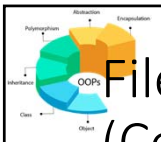
- **`seekp()`**: This function moves put file pointer to a specific location.
  - It is a member of `ofstream` class.
  - The prototype is as follows:

```
ostream& seekp(pos_type pos);
ostream& seekp(off_type off,ios::seekdir dir);
```

  - All of the argument types and return value are same as the `seekg()` function only difference is it sets the put pointer.
  - For example, the statements

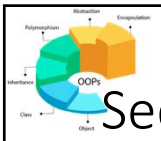
```
ofstream outfile("myfile.txt");
outfile.seekp(15);
outfile<<"**";
outfile.seekp(-5,ios::cur);
outfile<<"#";
```

    write ** to 15th and 16th character position of the file "myfile.txt". This causes the file access pointer to move to 16th position. The next `seekp()` moves the access pointer to 5 position back. So, "#" is written at 12th position.



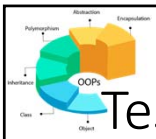
File Access Pointers and their Manipulators (Contd…)

- **`tellg()`**: This function returns the current file access pointer position of the get pointer.
  - The `tellg()` function is member function of `ifstream`.
  - The function `tellg()` has the following prototype
    ```
    pos_type tellg();
    ```
  - The statement
    ```
    int pos=infile.tellg();
    ```
    assigns the value of get pointer to the integer variable `pos`.
- **`tellp()`**: This function returns current file access pointer position of the put pointer.
  - The `tellp()` function is member function of `ofstream`.
  - The function `tellp()` is declared as
    ```
    pos_type tellp();
    ```
  - The statement
    ```
    int pos=outfile.tellp();
    ```
    assigns the value of put pointer to the integer variable `pos`.



Sequential and Random Access to File

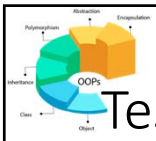
- In sequential access, to access a particular data in the file, all data from the start must be accessed and discarded up to that location.
- The main disadvantage is all the preceding data must be read.
- In some cases, it is the access method of choice, for example if we simply want to process a sequence of data elements in order.
- In random access, the data in any desired location can be accessed directly.
- Random access is necessary for writing and reading contents of a file at the specific location, modifying particular record or deleting some desired records.
- For random access file pointers must be moved to the location where the record is located or where the record is to be written using file access pointer function `seekg()` or `seekp()` for random access to file.



Testing Errors during File Operations

- There are many situations where errors may occur during file operations.
- The errors must be detected if any appropriate action is to be taken.
- Some potential situations where errors may arise during file manipulation and the mechanism for checking errors during those situations are given below
- While attempting to open a non existing file in read mode.

```
ifstream infile("data.txt");
if(!infile)
{
   //file does not exist
}
```

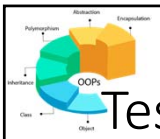
Testing Errors during File Operations (Contd…)

- While attempting to open a file in write mode for file marked as read only file

```
ofstream outfile("data.txt");
if(!outfile)
{
   //file for read only
}
```

- While trying to open a file by specifying filename not permitted by operating system

```
infile.open("#_/?.dat");
if(!infile)
{
   //invalid file name
}
```



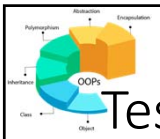
Testing Errors during File Operations (Contd…)

- While attempting invalid operation such as read a file beyond the end of the file.

```
while(1)
{
   //…
   if(infile.eof())
      break;
}
```

- While attempting to manipulate an unopened file.

```
infile.read(reinterpret_cast<char*>(&obj),sizeof(obj))
if(infile.fail())
{
   //file not opened
}
```

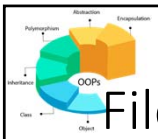
Testing Errors during File Operations (Contd…)

- While attempting to manipulate a file stored in malfunctioning.

```
infile.read(reinterpret_cast<char*>(&obj),sizeof(obj))
if(infile.bad())
{
   //file cannot be read
}
```

- While attempting to write a file where there is insufficient disk space

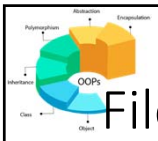
```
ofstream outfile;
outfile.open("data.txt");
if(outfile.bad())
{
   //writing not possible
}
```



File Input/Output with Member Functions

- The classes can be designed to include file input/output operations as member functions of the class.
- We can make normal member function or static member functions.
- The normal member functions are responsible for reading and writing their object to the file, where as static member functions are used to read and write the objects of the class which is similar to using global functions.

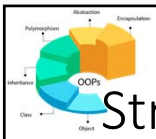
```
class student{
    private:
        char name[25];int rollno;
    public:
        //…
        void write2file();
        void readfromfile();
};
```

File Input/Output with Member Functions

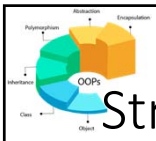
- The functions can be defined as

```
void student::write2file(){
    ofstream outfile("record.dat",ios::binary|ios::app);
    //…
    outfile.write(reinterpret_cast<char*>(this),sizeof(*this));
}
void student::readfromfile(){
    ifstream infile("record.dat",ios::binary);
    while(!infile.eof())
    {
       if(infile.read(reinterpret_cast<char*>(this),
          sizeof(*this))>0);
       //…
    }
}
student st; st.write2file();
```



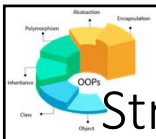
String Streams

- In C++ there is another type of stream called string streams into which formatted output can be written as text or the data stored as text can be read into variables using stream operators similar to console input/output.
- The string stream provides the facility similar to C library's `sprint()` and `sscanf()` functions.
- For the string stream input/output, C++ provides stream classes `istringstream`, `ostringstream` and `stringstream` which are 8-bit character specialization of `basic_*` template classes.



String Streams (Contd…)

- The classes `istringstream`, `ostringstream` and `stringstream` are derived from `istream`, `ostream` and `iostream` respectively.
- String stream classes are declared in header `<sstream>`.
- The copy of the string in the string stream object can be acquired and set by the member function `str()`.
- The `str()` member function uses C++ standard library's string class data as argument and return type.

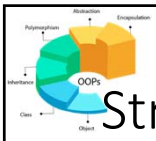


String Streams (Contd…)

- Similarly, the constructor of string stream accepts argument of type string.
- The constructor of `stringstream` are declared as

```
explicit stringstream(ios::openmode mode=ios::out|ios::in);
explicit stringstream(string st,ios::openmode mode=ios::out|ios::in);
```

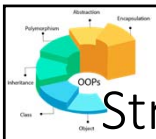
- Similarly, the constructor of `istringstream` is declared with default input open mode and the constructor of `ostringstream` is declared with default output open mode.



String Streams (Contd…)

- Following example illustrates the use of string streams

```
int main()
{
    char ch='B';
    int var=4;
    char str[]={"You"};
    //Output to string stream
    ostringstream ostr;
    ostr<<ch<<" "<<var<<" "<<str<<endl;
    cout<<ostr.str();
```



String Streams (Contd…)

```
        //Read from string stream
        istringstream istr("5 Apples cost 923.75");
        int count;
        char name[8], st2[8];
        float price;
        istr>>count>>name>>st2>>price;
        cout<<"Data read from istringstream are:"<<endl;
        cout<<count<<" "<<name<<" "<<st2<<" "<<price<<endl;
        return 0;
    }
```