Dr. SNS RAJALAKSHMI COLLEGE OF ARTS AND SCIENCE
(Autonomous)
Coimbatore-641049.
Accredited by NAAC (Cycle-III) with 'A+' Grade
(Recognised by UGC, Approved by AICTE, New Delhi and
Affiliated to Bharathiar University,Coimbatore)

STREAM CLASSES

Course Name: Object Oriented Programming

Course code: 21UCU403

UNIT: II

Prepared By : Dr.A.DEVI

OOP

- Stream and files
- Stream Classes
- Stream Errors
- Disk File I/O with Streams, Manipulators
- **File I/O Streams with Functions**
- Error Handling in File

OOP

OOP

C++ Streams

A Channel to send or receive data

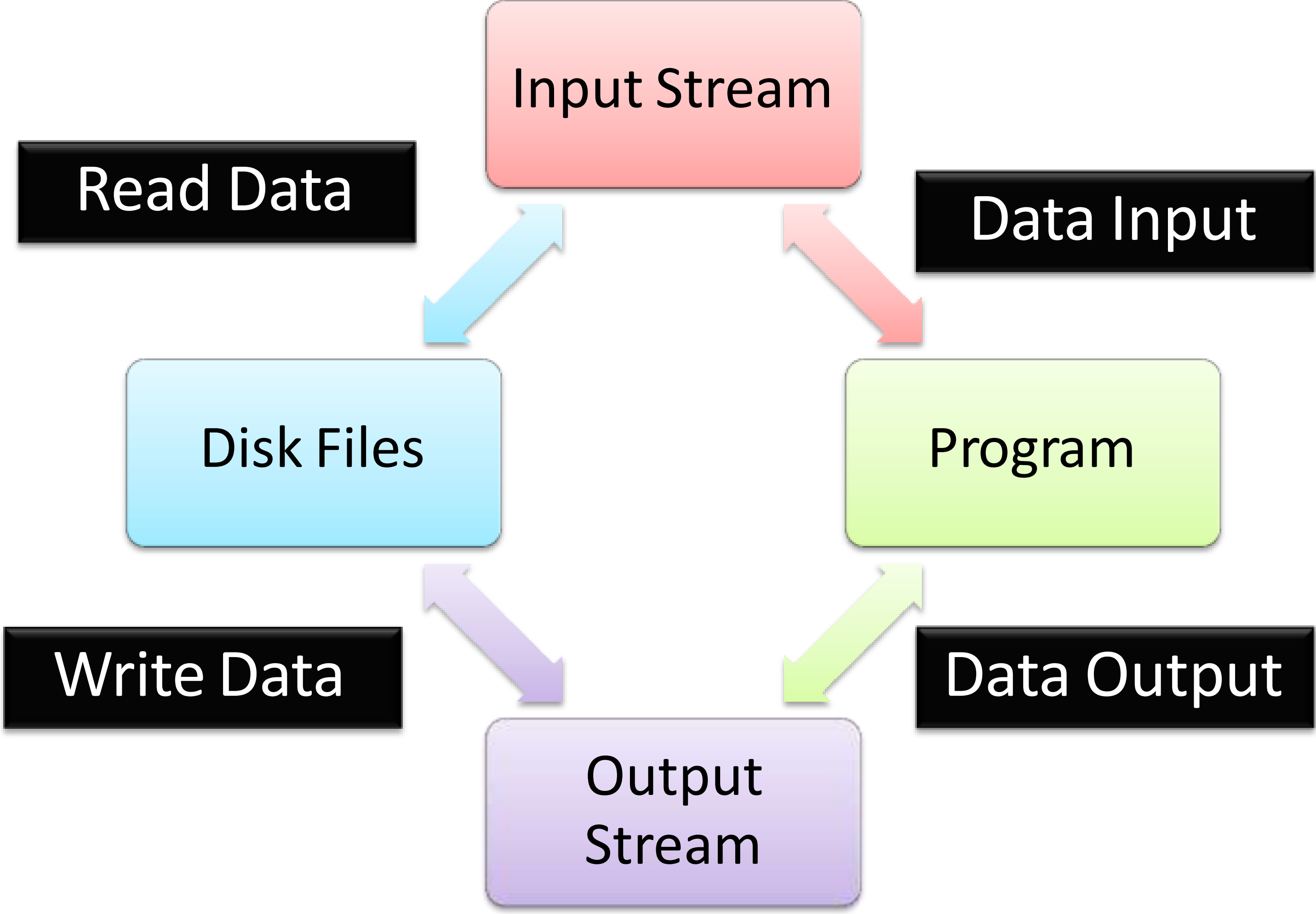
Data is sent by output stream and received by input stream

Character Stream : Is sequence of characters

Byte Stream : Sequence of Bytes …..

OOP

C++ Streams



OOP

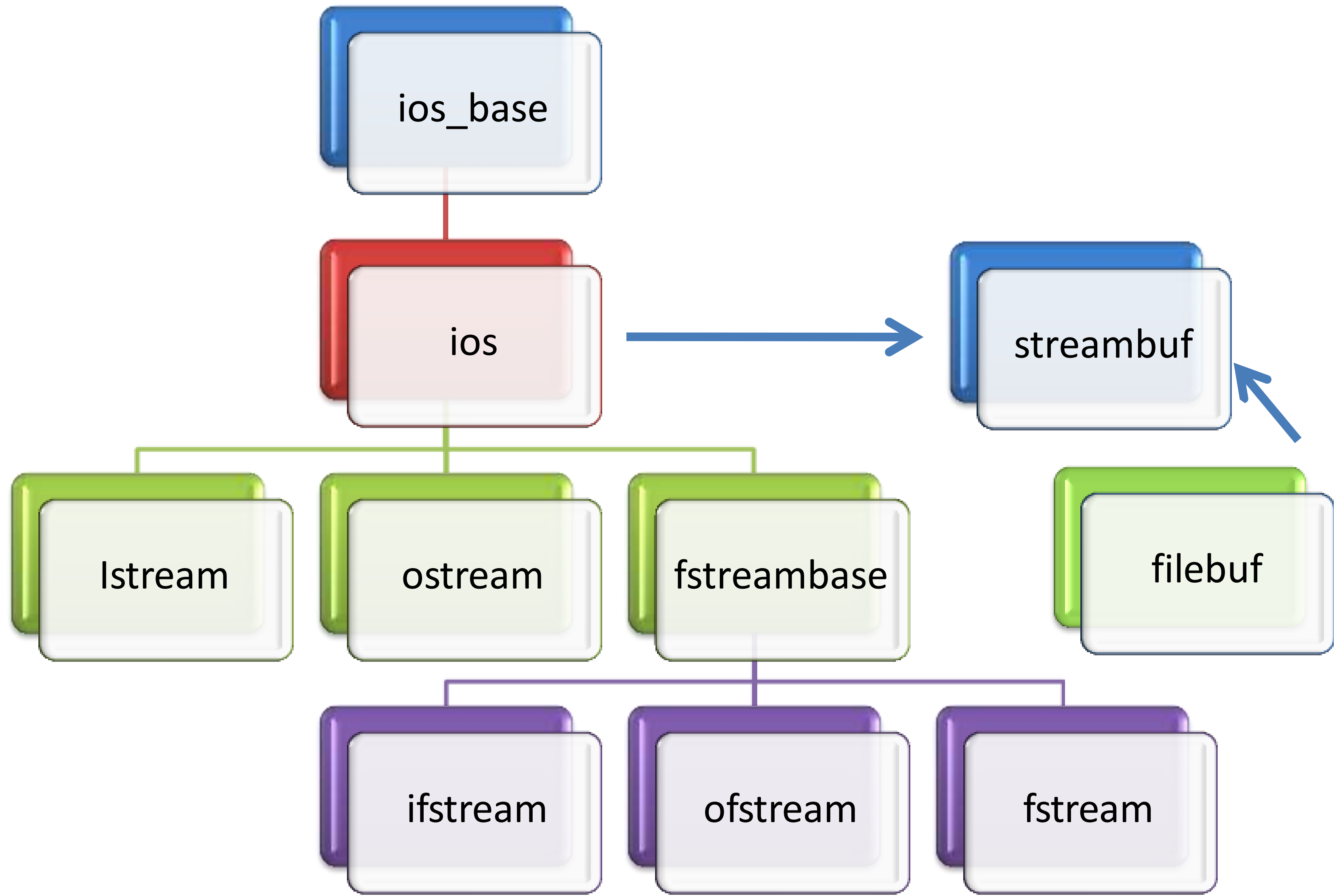
Standard I/O Streams

Stream	Description
cin	Standard input stream
cout	Standard output stream
cerr	Standard error stream
clog	Buffered version of cerr

When program begins execution these 4 streams are automatically opened

OOP

Class hierachy of Stream Classes



OOP

Unformatted I/O Functions

.get ( ) ;

Example:

```
char ch ;
ch = cin.get ( ) ;        // gets one character from keyboard
                          // & assigns it to the variable "ch"
```

.get (character) ;

Example:

```
char ch ;
cin.get (ch) ;            // gets one character from
                          // keyboard & assigns to "ch"
```

OOP

Unformatted I/O Functions

.get line(array_name, max_size) ;

OR

.read(array_name, max_size) ;

Example:

```
char  name[40]  ;
cin.get line(name, 40) ;      // Gets up to 39 characters
                // and inserts a null at the end of the
                // string "name".   If a delimiter is
                // found, the read terminates.  The
                // delimiter is not stored in the array,
                // but it is left in the stream.
cin.read(name, 40) ;
```

OOP

Unformatted I/O Functions

.put (character) ;

Example:

```
char ch ;
cout.put (ch) ;        // gets one character from
                       // keyboard & assigns to "ch"
```

OOP

Unformatted I/O Functions

.write(array_name, max_size) ;

Example:

```
char name[40] ;
cout.write (name, 40) ;
```

```
#include<iostream.h>
int main()
{
char *string1="C++";
char *string2="Program";
int m=strlen(string1);
int n=strlen(string2);
for(int i=1;i<n;i++)
{
   cout.write(string2,i);
   cout<<"\n";
}
```

```
for(i=n;i>0;i--)
{
    cout.write(string2,i);
    cout<<"\n";
}
cout.write(string1,m).write(string2,n);      //concatenating strings

cout<<"\n";

//crossing the boundary
cout.write(string1,8);
return 0;
}
```

Output of program:

p
pr
pro
prog
progr
progra
program
progra
progr
prog
pro
pr
p
C++ program
C++ progr

cout.write(string1,m).write(string2,n);

is equivalent to the following two statements:

```
cout.write(string1,m);
cout.write(string2,n);
```

OOP

FORMATTED CONSOLE I/O OPERATIONS

C++ supports a number of features that could be used for formatting the output. These features include:

- ios class functions and flags.
- Manipulators.
- User-defined output functions.

The ios class contains a large number of member functions that would help us to format the output in a number of ways.

The most important ones among them are listed in Table.

Function	Task
Width()	To specify the required field size for displaying an output value.
precision()	To specify the number of digits to be displayed after the decimal point of a float value.
fill()	To specify a character that is used to fill the unused portion of a field.
setf()	To specify format flags that can control the form of output display(such as left-justification and right-justification)
unsetf()	To clear the flags specified.

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.

To access manipulators, the file iomanip should be included in the program.

Manipulators	Equivalent ios function
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

Defining Field Width: width ()

We can use the width() function to define the width of a field necessary for the output of an item.

cout.width (w);

Where w is the field width(number of columns).

The field width should be specified for each item separately.

For example,the statements

```
cout.width(5);
cout<<543<<12<<"\n";
```

will produce the following output:

		5	4	3	1	2

Setting Precision:precision()

By default ,the floating numbers are printed with six digits after the decimal point.However ,we can specify the number of digits to be displayed after the decimal point while printing the floating-point numbers.

This can be done by using the precision() member function as follows:

```
cout.precision(d);
```

Where d is the number of digits to the right of the decimal point.

Default precision is 6 digits.

For example ,the statements

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout<<3.14159<<"\n";
cout<<2.50032<<"\n";
```

will produce the following output:

1.141(truncated)

3.142(rounded to the nearest cent)

2.5(no trailing zeros)

We can set different values to different precision as follows:

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout.precision(5);//reset the precision
cout<<3.14159<<"\n";
```

We can also combine the field specification with the precision setting. Example:

```
cout.precision(2);
cout.width(5);
cout<<1.2345;
```

The output will be:

	1	.	2	3

Filling and Padding :fill()

We can use the fill() function to fill the unused positions by any desired character.It is used in the following form:

cout.fill(ch);

Where ch represents the character which is used for filling the unused positions.

Example:

```
cout.fill('*');
cout.width(10);
cout<<5250<<"\n";
```

The output would be:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.

Formatting Flags,Bit-fields and setf()

The setf() function can be used as follows:

cout.setf(arg1,arg2);

The arg1 is one of the formatting flags defined in the class ios. The formatting flag specifies the format action required for the output.

Another ios constant, arg2, known as bit field specifies the group to which the formatting flag belongs.

Table shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive.

Format required	Flag(arg1)	Bit-Field(arg2)
Left-justified output Right-justified output Padding after sign or base Indicater(like +##20)	ios::left ios::right ios::internal	ios::adjustfield ios::adjustfield ios::adjustfield
Scientific notation Fixed point notation	ios::scientific ios::fixed	ios::floatfield ios::floatfield
Decimal base Octal base Hexadecimal base	ios::doc ios::oct ios::hex	ios::basefield ios::basefield ios::basefield

Examples:

```
cout.setf(ios::left,ios::adjustfied);
cout.setf(ios::scientific,ios::floatfield);
```

Consider the following segment of code:

```
cout.filll(‘*’);
cout.setf(ios::left,ios::adjustfield);
cout.width(15);
cout<<”12345”<<”\n”;
```

This will produce the following output:

1	2	3	4	5	*	*	*	*	*	*	*	*	*	*

The statements

```
cout.fill('*');
cout.precision(3);
cout.setf(ios::internal,ios::adjustfield);
cout.setf(ios::scientific,ios::floatfield);
cout.width(15);
cout<<-12.34567<<"\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

Displaying Trailing Zeros And Plus Sign

If we print the numbers 10.75, 25.00 and 15.50 using a field width of, say, eight positions, with two digits precision, and then the output will be as follows:

			1	**0**	**.**	**7**	**5**
						2	**5**
				1	**5**	**.**	**5**

Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

10.75

25.00

15.50

The setf() can be used with the flag ios::showpoint as a single argument to achieve this form of output.

For example,

```
cout.setf(ios::showpoint);//display trailing zeros
```

Similarly, a plus sign can be printed before a positive number using the following statement:

```
cout.setf(ios::showpos);//show + sign
```

For example, the statements

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed,ios::floatfield);
cout.setf(ios::internal,ios::adjustfield);
cout.width(10);
cout<<275.5<<"\n";
```

Will produce the following output:

+			2	7	5	.	5	0	0

Table lists the flags that do not possess a named bit field.

Flag	Meaning
ios::showbase ios::showpos ios::showpoint ios::uppercase	Use base indicator on output Print + before positive numbers Show trailing decimal point and zeroes Use uppercase letters for hex output
ios::skipus	skip white space on input
ios::unitbuf ios::stdio	Flush all streams after insertion Flush stdout and stderr after insertion

MANAGING OUTPUT WITH MANIPULATORS

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output formats.

Two or more manipulators can be used as a chain in one statement as shown below:

```
cout<<manip1<<manip2<<manip3<<item;
cout<<manip1<<item1<<manip2<<item2;
```

The most commonly used manipulators are shown in table.

Manipulator	Meaning	Equivalent
setw(int w) **setprecision(int d)**	Set the field width to w Set the floating point precision to d.	width() precision()
setfill(int c)	Set the fill character to c	fill()
setiosflags(long f)	Set the format flag f	setf()
resetiosflags(long f)	Clear the flag specified by f	unsetf()
Endif	Insert new line and flush stream	"\n"

Some examples of manipulators are given below:

```
cout<<setw(10)<<12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters.

The output can be made left-justified by modifying the statement as follows:

```
cout<<setw(10)<<setiosflags(ios::left)<<12345;
```

One statement can be used to format output for two or more values.

For example,the statement

```
cout<<setw(5)<<setprecision(2)<<1.2345
<<setw(10)<<setprecision(4)<<sqrt(2)
<<setw(15)<<setiosflags(ios::scientific)
<<sqrt(3)<<endl;
```

Will print all the three values in one line with the field size of 5,10,and 15 respectively.

There is a major difference in the way the manipulators are implemented as compared to the ios member functions. The ios member function return the previous format state which can be used later. In case, we need to save the old format states, we must use the ios member function rather than the manipulators.

Example:

cout.precision(2);//previous state

int p=cout.precision(4);//current state;

When these statements are executed, p will hold the value of 2(previous state) and the new format state will be 4.We can restore the previous format state as follows:

cout.precision(p)//p=2

Designing Our Own Manipulators

We can design our own manipulators for certain special purpose.The general form for creating a manipulator without any arguments is:

```
ostream & manipulator(ostream & output)
{
…………
…………(code)
…………
return output
}
```

Here the manipulator is the name of the manipulator under creation.

The following function defines a manipulator called unit that dispalys”inches”:

```
ostream & unit(ostream &output)
{
output<<”inches”;
return output;
}
```

The statement

```
cout<<36<<unit;
```

will produce the following output

```
36 inches
```

We can also create manipulators that could represent a sequence of operations.

Example:

```
ostream & show(ostream & output)
{
output.setf(ios::showpoint);
output.setf(ios::showpos);
output<<setw(10);
return output;
}
```

Program illustrates the creation and use of the user-defined manipulators. The program creates two manipulators called currency and form which are used in the main program.

```
#include<iostream.h>
#include<iomanip.h>
ostream & currency(ostream & output)
{
output<<"Rs";
return output;
}
```

```
ostream& form(ostream & output)
{
	output.setf(ios::showpos);
	output.setf(ios::showpoint);
	output.fill('*');
	output.precision(2);
	output<<setiosflags(ios::fixed)<<setw(10);
	return output;
}
int main()
{
	cout<<currency <<form<<7864.5;
	return 0;
}
```

The output of Program would be:

Rs**+7864.50

OOP

File I/O Streams

Stream	Description
ifstream	Reads from files
ofstream	Writes on files
fstream	Read & Write From/To files

To perform File I/o We include **<fstream.h>** in the program

OOPL

ifstream

Input file stream Class

open() is a member function of the class ifstream

Inherited functions of ifstream class, from the class istream are

- get()
- getline()
- read()
- seekg()
- tellg()

OOPL

ofstream

Output file stream Class

open() is a member function of the class ofstream

Inherited functions of ofstream class, from the class ostream are

- put()
- write()
- seekp()
- tellp()

OOP

File Handling Classes

```
ios
├── istream
│   ├── istrstream
│   ├── istream_withassign
│   └── ifstream
└── ostream
    ├── ofstream
    ├── ostream_withassign
    └── ostrstream

istream + ostream → iostream
                    ├── fstream
                    ├── strstream
                    └── stdiostream

streambuf
├── filebuf
├── strstreambuf
└── stdiobuf

Iostream_init
```

OOP

File Handling Classes

Opening a File

- Use **method "open()"**
- Or immediately in the **constructor** (the natural and preferred way).

copyrights © Elhanan Borenstein

OOP

Opening a File

- Before data can be written to or read from a file, the file must be opened.

ifstream inputFile;

inputFile.open("customer.dat");

Another Syntax

void open(const char* filename, int mode);

- filename – file to open (full path or local)
- mode – how to open (one or more of the following – using |)

OOP

File Handling Classes

- Modes can be
  - **ios::app – append**
  - **ios::ate – open with marker at the end of the file**
  - **ios::in / ios::out – (the defaults of ifstream and ofstream)**
  - **ios:nocreate / ios::noreplace – open only if the file exists / doesn’t exist**
  - **ios::trunc – open an empty file**
  - **ios::binary – open a binary file (default is textual)**

- Don’t forget to close the file using the method **“close()”**

Opening a File at Declaration

fstream f;

f.open(“names.dat”, ios::in | ios::out |ios:app);

Testing for Open Errors

```
dataFile.open("cust.dat", ios::in);
if (!dataFile)
{
   cout << "Error opening file.\n";
}
```

File Handling Classes

Querying a File

- **is_open()** – Checking whether the file was open correctly. (for compatibility with C, the operator ! was overloaded).
- **rd_state()** – returns a variable with one **or more** (check with AND) of the following options:
  - ios::goodbit – OK
  - ios::eofbit – marker on EOF
  - ios::failbit – illegal action, but alright to continue
  - ios:badbit – corrupted file, cannot be used.
- We can also access the bit we wish to check with **eof(), good(), fail(), bad().**
- **clear()** is used to clear the status bits (after they were checked).

copyrights © Elhanan Borenstein

Another way to Test for Open Errors

```
f.open(“cust.dat”, ios::in);
if (f.fail())
{
   cout << “Error opening file.\n”;
}
```

OOP

Detecting the End of a File

- The `eof()` member function reports when the end of a file has been encountered.

```
if (f.eof())
    f.close();
```

OOP

File Handling Classes

Moving within the File

- **seekg()** / **seekp()** – moving the reading (get) / writing (put) marker
  - two parameters: offset and anchor
- **tellg()** / **tellp()** – getting the position of the reading (get) / writing (put) marker

copyrights © Elhanan Borenstein

OOP

File Handling Classes

Reading /Writing from/to Textual Files

- **To write:**
  - **put() –** writing single character
  - **<< operator –** writing an object
- **To read:**
  - **get() –** reading a single character of a buffer
  - **getline() –** reading a single line
  - **>> operator –** reading a object

```
#include <fstream.h>
main()
{
        // Writing to file
        ofstream OutFile("my_file.txt");
        OutFile<<"Hello "<<5<<endl;
        OutFile.close();

        int number;
        char dummy[15];

        // Reading from file
        ifstream InFile("my_file.txt");
        InFile>>dummy>>number;

        InFile.seekg(0);
        InFile.getline(dummy, sizeof(dummy));
        InFile.close();
}
```

OOP

File Handling Classes

Reading /Writing from/to Binary Files

- **To write n bytes:**
  - **write (const unsigned char* buffer, int n);**
- **To read n bytes** (to a pre-allocated buffer):
  - **read (unsighed char* buffer, int num)**
  - Use: **int gcount()** to check how many byte where actually read (**WHY**)
  - Note: Unlike C, the buffers are of type unsigned char* (and not void*)

```
#include <fstream.h>
main()
{
        int array[] = {10,23,3,7,9,11,253};
        ofstream OutBinaryFile("my_b_file.txt“, ios::out | ios::binary);
        OutBinaryFile.write((char*) array, sizeof(array));
        OutBinaryFile.close();

}
```

OOP

Example

Example:

```
#include<iostream.h>
#include<fstream.h>
using namespace std;
int main( )
 {
    int a[ ] = {10,20,5,23,6}
    ofstream fob;        // creating object
    fob.open("output.txt");  // opening file using object
    while(fob)
        fob<<a[i]            //OR fob.write((char*) a, sizeof(a));
     fob.close();
     return 0;
}
```

OOP

Overloading >> & <<

- Overloading Insertion (<<) and Extraction Operator (>>) must be done by means of **friend functions**
- It is considered as binary operator where **one argument must be of ostream or istream compulsorily**

OOP

Overloading >>

- **Prototype:**

```
friend istream& operator >>(istream&, M&);
```

- **Example:**

```
istream& operator >>(istream& in, M & m)
{
    in >> m.data;
    return in;
}
```

```
void main()
{
    M  ob;
    cin>>ob;
}
```

OOP

Overloading <<

- **Prototype:**

```
friend ostream& operator <<(ostream&, M &);
```

- **Example:**

```
ostream& operator <<(ostream& out, M & m)
{
   out<< m.data
   out << endl;
   return out;
 }
```

```
void main()
{
    M  ob;
   cout<<ob;
}
```

OOP

Overloading >> & << Example

```
#include<fstream.h>
#include<iostream.h>
#include<conio.h>
class Emp
{
   char *name;
   float balance;
 public : Emp() :name(" "), balance(0)  { }
friend istream& operator >>(istream&, Emp&);
friend ostream& operator <<(ostream&, Emp&);
}
```

OOP

Overloading >> & << Example

```
istream& operator >>(istream& in, Emp & eob)
 {
   cout<<"\nEnter name and balance of Employee : ";
   in>>ob.name>>ob.balance;
   return in;
}
ostream& operator <<(ostream& out, Emp & eob)
 {  cout<<"\nName and balance of Employee : ";
   out<<ob.name<<ob.balance;
   return out;
}
```

OOP

Overloading >> & << Example

```
void main()
{
   Emp e;
    fstream f;
     f.open(“A.txt”,ios::in|ios::out|ios::binary);
    if(!f)
        cerr<<“\nCould not open the file “;
    cin>>e;
    f.write((char * ) e, sizeof(e));
```

Overloading >> & << Example

OOP

```
f.seekg(ios::beg)    //Or can write f.seekg(0);
while(!f.eof())
{
    f.read((char *) e, sizeof(e) );
   cout<<e;
}
} //main ends
```

OOP

Memory as Stream Object

```
#include<strstream.h>
#include<iostream.h>
#include<iomanip.h>
int main()
{
  char str1 = "Kainjan";
   int a = 100;
  char buff[100];
  ostrstream ob(buff,100);
  ob<<"a= " <<a;  ob<<“str = “<<str;
cout<<buff;
  return 0; }
```

OOP

Command Line Arguments

OOP

C/C++ Command Line Arguments

- When executing a program in either C or C++ there is a way to pass command line arguments.
- Passed a character arrays.
- Each parameter separated by a space
- Comes into the program as two arguments
  - argc – Number of parameters
  - argv – Parameter list

Command Line Arguments

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  for (int i=0; i<argc; i++)
    cout << “This is Argument number #”
         << i
         << “->”
          << argv[i]
         << endl;
  return 0;
}
```

OOP

Sample Output

C:\>hello Each word should be a unique argument

- This is Argument number #0 -->hello
- This is Argument number #1 -->Each
- This is Argument number #2 -->word
- This is Argument number #3 -->should
- This is Argument number #4 -->be
- This is Argument number #5 -->a
- This is Argument number #6 -->unique
- This is Argument number #7 -->argument

OOP

Command Line Arguments

Conventional rules:

- Arguments are always passed to main( ).
- There must be two
  - first is an integer
  - second char pointer to an array
- First argument (argv[0]) will always be the name of the calling program.
- argc will always be at least 1
- The first argument is always argv[0]
- The last argument is always argv[argc-1]
- argv[argc] will always be a null pointer
- Arguments are always passed as character strings. Numbers must be converted from characters to integers, floats, doubles, etc.

OOP

Using Command-Line Arguments

- Upcoming example
  - Program to copy input file to output file
  - **`copy input output`**
  - Read a character from file **`input`** and write to file **`output`**
    - Stop when no more characters to read (EOF)

```
#include <iostream.h>
#include<fstream.h>

int main(int argc, char*argv[] )
{
 //check no. of command line arguments
  if(argc!=3)
   cout<<"Usage : copy infilename outfilename"<<endl;
  else
   {
     ifstream inFIle(argv[1], ios::in);

     //input file cannot be opened
    if(!inFile)
     {
        cout<<argv[1] <<"could not be opened" <<endl;
        return -1;
     }
```

```
ofstream outFIle(argv[2], ios::out);
      //output file cannot be opened
      if(!outFile)
       {
           cout<<argv[2] <<"could not be opened" <<endl;
           return -1;
      }
char c = inFIle.get();

while(inFile)
 {
    outFIle.put( c );
   c = inFIle.get();
 }
} //end else

return 0;
} //end main();
```

Printer Output

- Sending Data to Printer is similar to sending data to a disk file.
- DOS predefines special filename for hardware devices.
- In most systems the printer is connected to first parallel port , so filename for the printer should be “PRN or LPT1”

Printer Output

Name	Device
CON	Console
AUX or COM1	First Serial Port
COM2	Second Serial Port
PRN or LPT1	First Parallel printer
LPT2	Second Parallel Printer
LPT3	Third Parallel Printer
NUL	Dummy device

Printer Output

- The following program : OPRINT, prints the contents of a disk file , specified on the command line to the printer.

```
//oprint.cpp
#include<fstream.h>
#include<process.h> //for exit()
void main(int argc , char * argv[ ])
 {     if (argc!=2)
       {
         cerr<<" \n oprint : filename:";
         exit(-1);
       }
```

Printer Output

```
char ch;
Ifstream inf;
inf.open(argv[1]);
if (!inf)
      {
         cerr<<“ \nCan’t open:”;
         exit(-1);
       }
 ofstream outf;
outf.open(“PRN”);
while(if.get(ch) !=0)
     outf.put(ch);     }
```

OOP

ly Binding Vs Late Binding

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Function Overloading, Operator Overloading	Virtual function in C++