Chapter: 12

C++ I/O system

Formatted I/O, ***width()***, ***precision()***, ***fill()***,
Manipulator & User Defined manipulators, inserter, extractor, file I/O, Random access etc.

12.1 C++ I/O Stream

☑ ***C++ I/O stream:*** The C++ I/O system, like the C I/O system, operates through ***streams***. Some important points about streams are:

- ☞ A stream is a ***logical device*** that either produces or consumes information.
- ☞ A stream is linked to a physical device by the C++ I/O system.
- ☞ All streams behave in the same manner, even if the actual physical devices they are linked to differ. For example, the same function that you use to write to the screen can be used to write to a disk file or to the printer.

☑ ***Predefined streams of C++:*** when a C program begins execution, three predefined streams are automatically opened: ***stdin***, ***stdout***, and ***stderr***. Similarly when a C++ program begins, these four streams are automatically opened:

Stream	*Meaning*	*Default Devices*	*Stream*	*Meaning*	*Default Devices*
cin	Standard input	Keyboard	***cerr***	Standard error	Screen
cout	Standard output	Screen	***clog***	Buffered version of ***cerr***	Screen

- ☛ The streams ***cin***, ***cout***, and ***cerr*** correspond to C's ***stdin***, ***stdout***, and ***stderr***. ***clog*** is buffered version of ***cerr***.
- ☛ Standard C++ also opens **wide (16-bit) character** versions of these streams called ***wcin***, ***wcout***, ***wcerr***, and ***wclog***.
- ☛ By default, the standard streams are used to communicate with the ***console***. However, in proper environments these streams **can be redirected to other devices**.

☑ ***<iostream> and template classes:*** C++ provides support for its I/O system in the header file ***<iostream>***. In this file, a rather **complicated set of class hierarchies** is defined that supports I/O operations.

- ☞ The ***I/O classes*** begin with a ***system of template classes***.
  - ⇨ ***Template classes/ generic classes:*** Template classes, also called ***generic classes*** *(will be discussed in next Chapter)*. Briefly, a template class defines the ***form of a class without fully specifying the data*** upon which it will operate. Once a template class has been defined, specific instances of it can be created.
- ☞ Standard C++ creates 2 specific versions of the I/O template classes: one for ***8-bit characters*** and another for ***wide characters***.

☑ The C++ I/O system is build upon two related, but different, template class hierarchies.

- ☛ ***basic_streambuf:*** Derived from the ***low-level I/O class***. This class supplies the basic, ***low-level input and output operations*** and provides the underlying ***support for the entire C++ I/O system***. It is used in advanced I/O programming.
- ☛ ***basic_ios***: The class hierarchy that you will most commonly be working with is derived from ***basic_ios***. This is a ***high-level I/O class*** that provides: ***formatting, error-checking***, and ***status information*** related to stream I/O.
  - ⇨ ***basic_ios*** is used as a base for several derived classes, including :
    1: ***basic_istream*** 2: ***basic_ostream*** 3: ***basic_iostream***
    These classes are used to create ***streams*** capable of **input**, **output**, and **input/output**, respectively.

☑ The following character-based names will be used throughout the remainder of this note.

Template Class	*8-Bit Character-Based Class*	*Template Class*	*8-Bit Character-Based Class*
basic_ios	ios	***basic_streambuf***	streambuf
basic_istream	istream	***basic_fstream***	fstream
basic_ostream	ostream	***basic_ifstream***	ifstream
basic_iostream	iostream	***basic_ofstream***	ofstream

NOTE:

1: If you include ***<iostream>*** in you program, you will have access to ***ios*** class.
2: The ***ios*** class contains many member functions and variables that ***control or monitor the fundamental operation of a stream***.

12.2 Formatted I/O

It is possible to output information in a wide variety of forms using C++'s I/O system as we did before with C's printf() function

❒ Each stream has associated with it a ***set of format flags*** that control the way **information is formatted**. The ***ios*** class declares a **bitmask enumeration** called ***fmtflags***, in which the values are defined:

adjustfield	***dec***	***hex***	***oct***	***showbase***	***skipws***
basefield	***fixed***	***internal***	***right***	***showpoint***	***unitbuf***
boolalpha	***floatfield***	***left***	***scientific***	***showpos***	***uppercase***

- ☞ These values are used to ***set*** or ***clear*** the **format flags** and are defined within ***ios***.

- ❍ ***skipws:*** When the ***skipws*** flag is ***set***, whitespace characters **(spaces, tabs, and newlines) will be cleared** for new input. When ***skipws*** is ***cleared***, whitespace **characters are not discarded**.

- ❍ ***left, right, internal (for justified output):*** Default is ***right***.
  - ⇨ ***left*** and ***right*** flags make, **left and right justified** output.
  - ⇨ ***internal*** flag makes a **numeric value is padded** to fill a field by **inserting spaces** between **sign/base** character.

- ***Dec, oct, hex :*** Decimal output is default
  - ⇨ ***oct*** and ***hex*** flags produce **octal** and **hexadecimal** output respectively.
  - ⇨ To return output to **decimal**, set the ***dec*** flag.
- ***scientific, fixed:*** If the ***scientific*** flag produce floating-point values using scientific notation. And ***fixed*** flag makes scientific-notation disabled, and normal notation returned.
  - ⇨ When neither flag is set, the compiler chooses an appropriate method.
- ***unitbuf*** flushes the buffer after each **insertion operation**.
- ***showbase*** displays the base of numeric values. Eg. for **hexadecimal** conversion, ***1F*** will be displayed as ***0x1F***.
- By default, the **scientific** notation ***"e"*** and **hexadecimal** notation ***"x"*** is displayed in ***lowercase***, setting ***uppercase*** flag displays these characters in ***uppercase***.
- ***showpos*** flag displays **"+"** before **positive** values.
- ***showpoint*** flag display **".000000"** for all **floating-point** output-whether needed or not.
- Booleans can be input or output using the keywords true and false, when ***boolalpha*** is set.

  - ➢ ***basefield:*** the ***oct***, ***dec***, and ***hex*** fields can be collectively referred as ***basefield***.
  - ➢ ***adjustfield***: the ***left***, ***right***, and ***internal*** fields collectively referred as ***adjustfield***.
  - ➢ ***floatfield***: the ***scientific*** and ***fixed*** fields collectively referenced as ***floatfield***.

❒ To set a ***format flag***, use the ***setf()*** function which is a member of ***ios***. Its form is: ***fmtflags setf (fmtflags flags);***

☞ This function ***returns the previous settings of the format flags*** and **turns on those flags** specified by ***flags***. (All other flags are unaffected.) For example, to turn on the **showpos flag**:

```
stream.setf ( ios :: showpos );
```

⌦ Here ***stream*** is the stream that you wish to affect.

⌦ Notice the use of the ***scope resolution operator*** (**::**). Because the ***format flags*** are defined within the ***ios*** class, you must access their values by using ***ios*** and the ***scope resolution operator***.

☞ ***setf()*** is a member function of the ***ios*** class and affects streams created by that class.

⇨ Therefore, any call to ***setf()*** is done **relative to a specific stream**.

⇨ ***setf()*** cannot be called by itself

⇨ There is no concept in C++ of **global format status**. ***Each stream maintains its own format status*** information individually.

☞ To set ***more than one flag*** in a single call to ***setf()***: use ***"OR"*** together the **values of the flags**. For example, this call sets the ***showbase*** and ***hex*** flags for ***cout***:

```
cout.setf( ios :: showbase | ios :: hex );
```

☞ NOTE: ***showpos***, ***showbase***, ***hex*** all are enumerated constants within the ***ios*** class. Therefore, it is necessary to tell the compiler this fact by preceding ***showpos/showbase/hex*** with the class name "***ios***" and the scope resolution operator "**::**". Otherwise ***showpos/showbase/hex*** will not be recognized. We must specify ***ios::showpos*** or ***ios::showbase*** or ***ios::hex***.

❒ The complement of ***setf()*** is ***unsetf()***.This member function of ***ios*** **clears one or more format flags**. Its prototype form is:

```
void unsetf ( fmtflags flags );
```

☞ The flags specified by ***flags*** are cleared. (All other flags are unaffected.)

❒ To know the ***current format*** settings **without altering**: Use the special member function of ***ios***, ***flags()***, which simply ***returns the current setting for each format flag***. Its prototype is: **fmtflags flags();**

☞ The **flags()** also allows to ***set/reset all format flags associated with a stream to those specified in the argument*** to **flags()**. The prototype for this version of **flags()** is:

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

⇨ For this version, the **bit pattern found in f** is copied to the **variable used to hold the format flags** associated with the stream, and ***overwrites all previous flag settings***. The function ***returns the previous settings***.

❒ ***Example 1:*** following program shows how to set several flags.

```
int main(){
    cout.unsetf(ios::dec); /* not required by all compilers */
    cout.setf (ios::hex | ios::scientific);
    cout<< 123.23 << "hello" << 100 <<'\n';
    cout.setf(ios::showpos );
    cout<< 10 <<' '<< -10 <<'\n';
    cout.setf(ios::showpoint | ios::fixed );
    cout<< 100.0;
    return 0; }
```

☞ This program displays: **1.232300e+02 hello 64 a fffffff6 +100.000000**

⇨ Here ***showpos*** flag affects only ***decimal*** output (i.e. **a fffffff6** is unaffected). It does not affect the value ***10*** when output in ***hexadecimal***.

⇨ Also notice the ***unsetf()*** call that ***turns off*** the ***dec*** flag (which is on by default). It is ***necessary to turn it off*** when ***turning on*** either ***hex*** or ***oct***. In general, it is better to set only the number base that you want to use and ***clear the others***.

❒ ***Example 2:*** The following program illustrates the effect of the ***uppercase*** flag. It first enable ***uppercase***, ***showbase***, and ***hex*** flags to output: ***99*** in ***hexadecimal***. Then disables the uppercase.

```
int main() { cout.unsetf(ios :: dec );
      cout.setf(ios::uppercase | ios::showbase | ios::hex);
      cout << 88 << '\n';
      cout.unsetf(ios::uppercase );
      cout << 88 << '\n';
      return 0; }
```

❒ ***Example3:*** The following illustrates the showflags() function. Displays which flag is on and which is off.

```
void showflags();                                   /* Declaration of the function */
int main(){  showflags();                           /* first shows default flag settings */
             cout.setf(ios::oct | ios::showbase | ios::fixed );   /* Changing flags */
             showflags();          /* shows changed flag settings */
                                                    return 0;}
```

```
void showflags(){
ios::fmtflags f;
f = cout.flags(); /* get flag settings */
if(f & ios :: skipws ) cout <<"skipws on \n";
else cout << " skipws off \n";

if(f & ios :: left ) cout << " left on\n";
else cout << " left off \n";

if(f & ios :: right ) cout << " right on\n";
else cout << " right off \n";

if(f& ios :: internal) cout<<"internal on\n";
else cout << " internal off \n";

if(f & ios :: dec ) cout << "dec on\n";
else cout << "dec off \n";

if(f & ios :: oct ) cout << "oct on\n";
else cout << "oct off \n";

if(f & ios :: hex ) cout << "hex on\n";
else cout << "hex off \n";

if(f&ios:: showbase ) cout<<"showbase on\n";
else cout << " showbase off \n";

if(f&ios::showpoint) cout<<"showpoint on\n";
else cout << " showpoint off \n";

if(f& ios :: showpos ) cout << " showpos on\n";
else cout << " showpos off \n";

if(f&ios::uppercase) cout<<"uppercase on\n";
else cout << " uppercase off \n";

if(f&ios::scientific) cout << " scientific on\n";
else cout << " scientific off \n";

if(f & ios :: fixed ) cout << " fixed on\n";
else cout << " fixed off \n";

if(f & ios :: unitbuf ) cout << " unitbuf on\n";
else cout << " unitbuf off \n";

if(f & ios :: boolalpha ) cout << " boolalpha on\n";
else cout << " boolalpha off \n";

cout << "\n"; }
```

☞ Inside ***showags()***, the ***local variable f*** is declared to be of type ***fmtflags***. If your compiler does not define ***fmtflags***, declare this variable as ***long*** instead.

12.3 width(), precision(), AND fill()

To set these format parameters: the ***field width***, the ***precision***, and the ***fill character***, there are ***three member functions*** defined by ***ios***. These are ***width()***, ***precision()*** and ***fill()***, respectively.

☑ ***width():*** To specify a minimum field width we use the width() function. Its prototype is:

```
streamsize width( streamsize w);
```

- ➲ Here ***w*** becomes the ***field width***, and the previous field width is returned.
- ➲ The ***streamsizetype*** is defined by ***<iostream>*** as some form of ***integer***.
- ➲ It might be necessary to ***set the minimum field width before each output*** statement.
- ➲ When a value uses ***less than the specified width***, the field is ***padded with*** the current fill character (the ***space***, by default) so that the field width is reached.
- ➲ If the size of the output value ***exceeds the minimum field width***, the field will be ***overrun***. ***No*** values are ***truncated***.

☑ ***precision():*** By default, six digits of precision are used. You can set this number by using the ***precision()*** function. Its prototype:

```
streamsize precision( streamsize p);
```

- ➲ Here the ***precision*** is set to ***p*** and the old value is returned.

☑ ***fill():*** by default, when a field needs to be filled, it is filled with spaces. To specify the fill character use ***fill()*** function. Prototype:

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

- ➲ After a call to ***fill()***, ***ch*** becomes the ***new fill character***, and the old one is returned.

☑ ***Example 1:*** Following illustrates the basics of ***width***, ***precision*** and ***fill***.

```
int main(){ cout.width(10) ;          // set minimum filed width
            cout<< "hello"<<'\n';     // right - justify by default

            cout.fill ('%');          // set fill character
            cout.width(10) ;          // set width
            cout<< "hello" << '\n';   // right - justify default

            cout.setf(ios :: left );  // left - justify
            cout.width (10) ;         // set width
            cout<< "hello" <<'\n';    // output left justified

            cout.width(10) ;          // set width
            cout.precision (10) ;     // set 10 digits of precision
            cout<< 123.234567 << '\n';

            cout.width(10) ;          // set width
            cout.precision(6) ;       // set 6 digits of precision
            cout<< 123.234567 << '\n';

            return 0; }
```

☛ Notice that the ***field width is set before each output*** statement.

☑ ***Example 2:*** The following segment uses the C++ I/O format functions to create an aligned table of numbers:

```
int main(){ double x;
   cout.precision(4) ;
   cout<< "x  sqrt(x)  x^2  \n\n";

   for(x=2.0; x<=20.0; x++) { cout.width(7); cout<<x<<" ";
                              cout.width(7); cout<<sqrt(x)<<" ";
                              cout.width(7); cout<<x*x<< \n';}
   return 0; }
```

12.4 I/O MANIPULATORS

I/O manipulators are ***special I/O format functions*** that can occur **within an I/O statement**. (Where ***ios*** member functions stay separate from I/O statement). For example:

```
cout << oct << 100 << hex << 100;
cout << setw(10) << 100;
```

- ⇨ The first statement tells ***cout*** to display integers in ***octal*** and then outputs ***100*** in ***octal***. It then ***tells the stream*** to display integers in ***hexadecimal*** and then outputs ***100*** in ***hexadecimal*** format.
- ⇨ The second statement sets the ***field width*** to ***10*** and then displays ***100*** in ***hexadecimal format again*** (last base setting active).
- ⇨ Notice that ***when a manipulator does not take an argument***, such as oct in the example, ***it is not followed by parentheses***. This is because ***it is the address of the manipulator*** that is ***passed to the overloaded << operator***.

❒ The main ***advantages*** of using ***manipulatior*** over the ***ios member functions*** is that they are ***easier*** to use and allow ***compact coding***.

- ☞ Many of the ***I/O manipulators*** parallel **member functions** of the ***ios class***.
- ☞ An I/O manipulator affects only the stream of which the I/O expression is a part and doesn't affect all currently opened streams.
- ☞ To access manipulators that take parameters, such as setw(), you must include <iomanip> in you program. This is not necessary when you are using a manipulator that does not require an argument.

❒ ***Example 1:*** Following includes ***setfill()*** and ***setw()*** so we have to include ***<iomanip>***

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() { cout << hex << 100 << endl ;
             cout << setfill('X') << setw(10) ;
             cout << 100 << " hi " << endl ;
             return 0;}
```

❒ ***Boolalpha:*** ***boolalpha*** allows you to input and output ***Boolean*** values using the keywords ***true*** and ***false*** (normally you must enter ***1*** for ***true*** and ***0*** for ***false***).

☞ Must set the ***boolalpha*** flags for ***cin*** and ***cout separately***. Eg: ***cin >> boolalpha >> b;*** // enter true or false

☞ As with all format flags, setting ***boolalpha*** for one stream does not imply that it is also set for another. For ***Example 2:***

```
int main() { bool b;
cout << " Before boolalpha:";
b = true ;
cout << b <<" ";
b = false ;
cout << b << endl;
cout << " After boolalpha:";
b = true ;
cout << boolalpha << b << " ";
b = false ;
cout << b << endl;
return 0;}
```

OUTPUT:

```
Before boolalpha: 1  0
After boolalpha:  true false
```

❒ ***Set/Reset:***

☞ To set specific format flags manually by ***manipulator***, use ***setiosflags()*** which is equivalent to ***setf()***.

☞ To turn off flags use the ***resetiosflags()*** manipulator which is equivalent to ***unsetf()***.

❒ ***Table of Slandered C++ I/O Manipulators:***

Manipulator	*Purpose*	*I/O*	*Manipulator*	*Purpose*	*I/O*
boolalpha	Turns on ***boolalpha*** flag	I/O	oct	Turns on ***oct*** ag	I/O
dec	Turns on ***dec*** flag	I/O	right	Turns on ***right*** flag	Output
endl	***newline*** and ***flushes*** stream	Output	resetiosflags(fmtflags f)	Turns off the flags specified in ***f***	I/O
ends	Outputs a ***null***	Output	scientific	Turns on ***scientific*** flag	Output
fixed	Turns on ***fixed*** flag	Output	setbase(int base)	Sets the ***number base*** to base	I/O
flush	***Flushes*** a stream	Output	setfill(int ch)	Sets the ***fill character*** to ***ch***	I/O
hex	Turns on ***hex*** flag	I/O	setiosflags(fmtflags f)	Turns on the flags specified in ***f***	Output
internal	Turns on ***internal*** flag	Output	setprecision(int p)	Sets the number of ***digits*** of precision	Output
left	Turns on left	Output	setw(int w)	Sets the ***field*** width to ***w***	Output
noboolalpha	Turns off ***boolalpha*** flag	I/O	showbase	Turns on ***showbase*** flag	Output
noshowbase	Turns off ***showbase*** flag	Output	showpoint	Turns on ***showpoint*** flag	Output
noshowpoint	Turns off ***showpoint*** flag	Output	showpos	Turns on ***showpos*** flag	Output
noshowpos	Turns off ***showpos*** flag	Output	skipws	Turns on ***skipws*** flag	Input
noskipws	Turns off ***skipws*** flag	Input	unitbuf	Turns on ***unitbuf*** flag	Output
nounitbuf	Turns off ***unitbuf*** flag	Output	uppercase	Turns on ***uppercase*** flag	Output
nouppercase	Turns off ***uppercase*** flag	Output	ws	Skips leading ***white space***	Input

12.5 Inserters and Extractors

O ***Insertion and inserter:*** In C++, the output operation is called an ***insertion*** and the **<<** is called the ***insertion operator***. (The reason for this term: an output operator inserts information into a stream).

☞ Overloading the **<<** for output, creates an *inserter function*, or ***inserter*** for short. All inserted functions have this ***general form***:

```
ostream &operator<<(ostream &stream, class_name obj) {  /* body of inserter */
                                                        return stream;  }
```

⇨ The inserted function "**ostream &operator<<**" returns a ***reference to stream***, which is of type ***ostream***. This is required if the overloaded **<<** is going to be used in a series of I/O expressions, such as ***cout<< ob1 << ob2 << ob3 ;***

⇨ The first parameter is a ***reference to an object*** of type ***ostream***. This means that stream *must be an output* stream. (***ostream*** is derived form the ***ios class***.)

⇨ The second parameter receives the *object that will be output*. (This can also be a reference parameter).

O ***Extraction and extractor:*** In C++, the **>>** is referred to as the ***extraction operator***. The reason for this term is that the act of inputting information from a stream removes (that is, extracts) data from it.

☞ A function that overloads **>>** is called an ***extractor***. The ***general form*** of an extractor function is:

```
istream &operator>>(istream &stream, class_name &ob) { /* body of extractor */
                                                       return stream;  }
```

▶ Extractors return a ***reference to stream*** "**istream &operator>>**", which is of type ***istream*** an input stream.

▶ The first parameter must be a ***reference to an input stream***.

▶ The second parameter is a ***reference to the object*** that is receiving input.

O ***Inserter and extractor cannot be a member of a class:*** If an overloaded operator function is a *member of a class*, the left operand (which implicitly passed through this and also generates the call to the operator) must be an *object of that class*.

☞ When you create an ***inserter/extractor***, the left operand is a stream and the right operand is the object that you want to ***output/input***. Therefore, an ***inserter/extractor*** cannot be a member function.

- ***Inserter and extractor as `friend` of a class:*** ***Inserters/extractors*** can be ***friends*** of the class. In fact, in most programming situations you will encounter, an overloaded inserter will be a ***friend of the class*** for which it was created.
- ***Example 1:*** This program contains an inserter and an extractor for the ***coord*** class.

```
class coord {int x, y;
      public:
            coord() { x = 0; y = 0; }
            coord(int i, int j) { x = i; y = j; }
            friend ostream & operator <<( ostream &stream , coord ob);        /* inserter */
            friend istream & operator >>( istream &stream , coord &ob);       /* extractor */
            };
```

```
ostream &operator<<(ostream &stream, coord ob){
            stream << ob.x << ", " << ob.y << '\n';
            return stream ; }
istream &operator>>(istream &stream, coord &ob){
            cout << " Enter coordinates : ";
            stream >> ob.x >> ob.y;
            return stream ; }
```

```
int main() { coord a(1, 1) , b(10 , 23);
             cout << a << b;
             cin >> a;
             cout << a;
             return 0; }
```

- ***Make inserter/extractor as general as possible:*** In this case, the ***I/O*** statement inside the ***inserter/extractor*** outputs/inputs the values of ***x*** and ***y*** to "***stream***", which is *whatever stream* is passed to the function *("**`stream`**" is general for **`cin`**, **`cout`** and both **"<<"** & **">>"** can be used with it).* As you will see in the following chapter, when written correctly **the same inserter that outputs to the screen** ***can be used to output to any stream***.
  - ☞ However the following is *not for general streaming*. In this case, the output statement is ***hard-coded*** to display information on the *standard output device* linked to **`cout`**. This prevents the inserter from being used by other streams.

```
ostream &operator<<(ostream &stream, coord ob){
                    cout << ob.x << ", " << ob.y << '\n';    /* using "cout" instead of "stream" */
                    return stream ;}
```

- ***Non-Friend inserter/extractor:*** If inserter/extractor are not friends to any class, they **cannot use** the ***private*** members of any class. However all ***public*** members are **accessible**.

12.6 User Defined Manipulators

- Custom manipulators are important for *two main reasons*.
  - ☛ First, a manipulator can ***consolidate a sequence of several separate I/O operations***: in which the same sequence of I/O operations occurs frequently within a program.
  - ☛ Second, a custom manipulator can be important when you need ***to perform I/O operations on a nonstandard device***. For example, you could use a manipulator to send control codes to a special type of *printer or an optic recognition* system.
- ***Types of Manipulators:*** there are *two basic types of manipulators*: those that ***operate on input streams*** and those that ***operate on output streams***. There is a secondary division: those manipulators that take an argument and those that don't:
  [1] ***parameterized*** manipulator and [2] ***parameterless*** manipulator
  - ☞ ***parameterized manipulator:*** The procedures necessary to create a ***parameterized manipulator*** vary widely from compiler to compiler, and even between two different versions of the same compiler. For this reason, you must consult the documentation to your compiler for instructions on creating parameterized manipulators. Parameterized manipulator is out of scope of this note.
  - ☞ ***parameterless manipulators:*** However, the creation of ***parameterless manipulators*** is straightforward and the same for all compilers.
    - ⇨ ***Output functions:*** All *parameterless manipulator* ***output*** *functions* have this skeleton:

```
ostream &manip_name( ostream &stream ){   /* your code here */
                                          return stream ; }
```

      - ❖ Here ***`manip_name`*** is the name of the manipulator and
      - ❖ ***`stream`*** **" &stream "** is a ***reference to the invoking stream*** which must be returned by the manipulator ***" return stream; "***. This is necessary if a manipulator is used as part of a larger *I/O expression*.
      - ❖ Here the manipulator has a single argument "a reference to the stream upon which it is operating", but ***no argument will be used*** when the manipulator is called in an output operation.
    - ⇨ ***Input functions:*** All *parameterless* ***input*** *manipulator functions* have this skeleton:

```
istream &manip_name( istream &stream ){   /* your code here */
                                          return stream ; }
```

      - ❖ An input manipulator receives a ***reference to the stream*** on which it was ***invoked***. This stream must be returned by the manipulator.
- ***Example 1:*** Following creates a manipulator called ***`setup()`*** that sets field width to **10**, precision to **4**, and fill character to *****.

```
ostream &setup(ostream &stream){  stream.width(10);
                                  stream.precision(4);
                                  stream.fill('*');
                                  return stream ;}
```

```
int main(){   cout <<setup<< 123.123456;
              return 0;}
```

❒ ***Example 2:*** Following creates the **getpass( ) input manipulator**, which rings the bell and then prompts for a password:

```
#include<cstring >
            /* A simple input manipulator */
istream &getpass(istream &stream){
            cout << '\a';        /* sound bell */
            cout << " Enter password : ";
            return stream ; }
```

```
int main(){   char pw[80];
                /* comparing password */
              do{ cin >> getpass >> pw; }
              while(strcmp(pw, "password"));
              cout << " Logon complete \n";
              return 0; }
```

12.7 File I/O

❒ The same class hierarchy that supports ***console I/O*** also supports ***file I/O***. To perform ***file I/O***, you must include the header ***<fstream>*** in your program. It defines several classes, including ***ifstream***, ***ofstream***, and ***fstream*** which are derived from ***istream*** & ***ostream***. And ***istream***, ***ostream*** are derived ***fron*** ios.

☞ So ***ifstream***, ***ofstream***, and ***fstream*** also have access *to all operations* defined by ***ios***.

❒ In C++, a file is opened by *linking it to a stream*. Before you can open a file, you must first obtain a **stream**. There are *three types* of streams:

[1] ***input:*** To create an **input** stream, declare an object of type ***ifstream***.
[2] ***output:*** To create an **output** stream, declare an object of type ***ofstream***.
[3] ***input/output:*** Streams that will be performing both **input and output** operations must be declared as objects of type ***fstream***.

For example, this fragment creates an *input* stream, an *output* stream, and one stream capable of *both input and output*.

```
ifstream in;        /* input */
ofstream out;       /* output */
fstream io;         /* input and output */
```

❒ ***Associate stream with a file:*** Use the function ***open( )*** to associate a stream with a file. This function is a member of each ***ifstream***, ***ofstream***, and ***fstream***. The prototype for each:

```
void ifstream :: open(const char *filename, openmode mode = ios::in);
void ofstream :: open(const char *filename, openmode mode = ios::out | ios::trunc );
void fstream :: open(const char *filename, openmode mode = ios::in | ios::out);
```

☞ Here ***filename*** is the name of the file, which can include a ***path specifier***.

☞ The value of ***mode*** determines how the file is opened. It must be a value of ***type openmode***, which is an ***enumeration*** defined by ***ios*** that contains the following values:

1. ***ios::app*** causes all output to that file to be ***appended*** to the end. This value can be used only with files capable of output.
2. ***ios::ate*** causes a seek to the ***end-of-file*** to occur when the file is opened (***I/O*** can still occur anywhere within the file).
3. ***ios::binary*** value causes a file to be opened in ***binary mode*** (text is default mode). In binary mode *no character translations* (carriage return/linefeed sequences) will occur.
4. ***ios::in*** value specifies that the file is ***capable of input***.
5. ***ios::out*** value specifies that the file is ***capable of output***.
6. ***ios::trunc*** value causes the contents of a *preexisting file by the same name* to be ***destroyed*** and the file to be ***truncated to zero length***.
   ✓ When output stream using ***ofstream*** created, any *preexisting file with the same name is automatically truncated*.

☞ These six values ***can be combined*** using ***OR***.

❒ ***Example 1:*** The following fragment opens an *output file* called ***test***:

```
ofstream mystream;
mystream.open(" test ");
```

☞ Since the ***mode parameter*** to ***open( )*** defaults to a *value appropriate to the type of stream* being opened, there is no need to specify its value in the preceding example.

❒ ***Confirmation test:*** If ***open( ) fails***, the stream will *evaluate to false when used in a Boolean* expression. Which can be used in a confirmation test (consider Example 1) :

```
if(!mystream) {     cout << "Cannot open file. \n";
                      /* handle error */  }
```

☞ Always check the result of a call to ***open( )*** before attempting to access the file.

☞ Use the ***is_open( )*** function to see if a ***file successfully opened***. ***is_open( )*** is a member of ***fstream***, ***ifstream***, and ***ofstream***. It has this prototype: **bool** is_open( );

⇨ It returns ***true*** if the *stream is linked to an open file* and ***false otherwise***. For example, the following checks if ***mystream*** is **currently open**:

```
if( !mystream.is_open( ) ){ cout << " File is not open .\n";
                            . . . . . }
```

❒ ***Bypass the open() function:*** Most of the times we don't need to use the function ***open( )*** because the ***ifstream***, ***ofstream***, and ***fstream*** classes have *constructor functions that automatically open the file*. And those constructor functions have the *same parameters and defaults* as the ***open( )***. Therefore, the most common way to open a file is:

```
ifstream mystream("myfile");          /* open file for input */
```

☞ If the file cannot be opened, the stream variable will evaluate as false when used in a ***conditional/Boolean*** statement.

☞ Therefore, in this case we also need the ***confirmation test*** as stated above.

❒ ***Closing a file:*** To close a file, use the member function **`close()`**. For example, to close the file linked to a stream called ***mystream***:

```
mystream.close();
```

☞ The **`close()`** function takes *no parameters* and returns *no value*.

❍ ***The eof() function:*** Use the **`eof()`** member function of ***ios*** to detect when the *end-of-input-file* has been reach. It has this prototype:

```
bool eof();
```

☞ It *returns* ***true*** when the ***end-of-file*** has been encountered and ***false*** otherwise.

❍ ***Read/Write textual data:*** to read/write textual data from/to an *opened file* we simply use **<<** and **>>** operators (more like C's **`fprintf()`** and **`fscanf()`**).

☞ A file ***produced*** by using **<<** is a *formatted text* file. A file ***read*** by **>>** must be a *formatted text* file.

☞ Typically, ***formatted text files*** are operated through the **>>** and **<<** operators. They are *not for binary mode*. ***Binary mode*** is best used on ***unformatted files***.

❍ ***Example 2:*** Following creates an output file, write information to it, closes the file and opens it again as an input file, and reads in the information:

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){ ofstream f_out("test");      // create output file
if(!f_out){   cout << "Cannot open output file .\n";
              return 1;}                 //notice !f_out Boolean!!!
f_out << "Hello !\n";
f_out << 100 << ' ' << hex << 100 << endl ;
f_out.close();                           //closing the created file
```

```
ifstream f_in("test");       //open input file
if(!f_in ){   cout << "Cannot open input file .\n";
              return 1; }
char str[80];
int i;
f_in >> str >> i;
cout << str << ' ' << i << endl ;
f_in.close();                //closing the opened file
return 0; }
```

☞ When the **<<** and **>>** operators are used **to perform file I/O**, information is formatted exactly as it would appear on the screen.

12.8 UNFORMATTED I/O & BINARY I/O

Unformatted files contain the same *binary representation* of the data *as that used internally by* your *program* (rather than text data which is translated into by the **<<** and **>>**). Thus, unformatted functions give you detailed control over how files are written and read.

❍ ***Lowest-level unformatted I/O:*** The lowest-level unformatted ***I/O*** functions are **`get()`** and **`put()`**. **`get()`** is used to ***read a byte*** and **`put()`** is used ***to write a byte***. These are members of all ***I*** & ***O*** stream classes respectively. Common version of **`get()`** & **`put()`**:

```
istream &get(char &ch);
ostream &put(char &ch);
```

☞ ***`get()`*** reads ***a single character*** from the *associated stream* and puts that value in ***`ch`***.

⇨ It *returns a reference* to the ***stream***.

⇨ If a read is attempted at ***end-of-file***, on return the invoking stream will evaluate to *false in Boolean expression*.

☞ ***`put()`*** writes ***`ch`*** to the stream and *returns a reference* to the ***stream***.

❍ ***Overloading `get()`:*** There are several different ways in which the ***`get()`*** function is overloaded. The prototypes for the three most commonly used overloaded forms are:

```
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
int get();
```

☞ The ***first form*** reads characters into the *array pointed to* by ***`buf`*** until either ***`num-1`*** characters have been read, *a newline is found*, or the ***end-of-file*** has been encountered.

⇨ They *array pointed to* by ***`buf`*** will be ***null terminated*** by ***`get()`***.

⇨ If the *newline character* is encountered in the ***input stream***, it is **not** ***extracted*** (***inputted***). Instead, *it remains in the stream until the next input operation*.

☞ The ***second form*** reads characters into the *array pointed to* by ***`buf`*** until either ***`num-1`*** characters have been read, *the character specified by* ***delim*** *has been found*, or the ***end-of-file*** has been encountered.

⇨ The *array pointed to* by ***`buf`*** will be ***null terminated*** by ***`get()`***.

⇨ If the *delimiter character* is encountered in the ***input stream***, it is **not** ***extracted*** (***inputted***). Instead, *it remains in the stream until the next input operation*.

▣ NOTE ***(Delimiter character):*** A ***delimiter*** is one or more *characters that separate text strings*. Common delimiters are **`commas(,)`**, **`semicolon(;)`**, **`quotes(",')`**, **`braces({})`**, **`pipes(|)`**, or **`slashes(/ \)`**. ***Newline character*** is also a ***delimiter***.

☞ The ***third form*** of ***`get()`*** ***returns*** the *next character from the stream*. It ***returns EOF*** if the ***end-of-file*** is encountered. This form of ***`get()`*** is similar to C's ***`getc()`*** function.

❍ **`getline()`** ***with overloaded form:*** ***`getline()`*** is another *input function*. It is a member of each input stream class. Its prototypes:

```
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char delim);
```

☞ The ***first form*** reads characters into the *array pointed to* by ***`buf`*** until either ***`num-1`*** characters have been read, *a newline is found*, or the ***end-of-file*** has been encountered.

⇨ They *array pointed to* by ***`buf`*** will be ***null terminated*** by ***`getline()`***.

⇨ If the *newline character* is encountered in the ***input stream***, it is ***extracted*** (***inputted***), but it is not put into ***`buf`***.

☞ The ***second form*** reads characters into the *array pointed to* by ***buf*** until either ***num-1*** **characters** have been read, *the character specified by* ***delim*** *has been found*, or the ***end-of-file*** has been encountered.

⇨ The *array pointed to* by ***buf*** will be ***null terminated*** by ***getline()***.

⇨ If the *delimiter character* is encountered in the ***input stream***, it is ***extracted*** (***inputted***), but it is not put into ***buf***.

- ***Comparison between*** **get() *and*** **getline():** The two versions of ***getline()*** are virtually identical to the ***get(buf, num)*** and ***get(buf, num, delim)*** versions of ***get()***.

☞ The difference between ***get()*** and ***getline()*** is that ***getline()*** ***reads and removes the delimiter*** from the input stream; ***get()*** ***does not***.

- ***Data blocks I/O:*** To *read* and *write* ***blocks of data***, use the ***read()*** and ***write()*** functions, which are also members of the *I/O stream classes*, respectively. Their prototypes are:

```
istream &read (char *buf, streamsize num);
ostream &write (const char *buf, streamsize num);
```

☛ ***read()*** reads ***num*** bytes from the ***invoking stream*** and puts them in the ***buffer*** pointed to by ***buf***.

☛ ***write()*** writes ***num*** bytes to the ***associated stream*** from the ***buffer*** pointed to by ***buf***.

☛ ***streamsize*** type is some *form of integer*. An object of type ***streamsize*** is capable of *holding the largest number of bytes* that will be transferred in any one ***I/O*** operation.

☛ If the ***end-of-file*** is reached before ***num*** characters have been read, ***read()*** simply stops, and the ***buffer*** contains *as many characters as were available*.

- ***gcount():*** You can find out ***how many characters have been read*** by using the member function ***gcount()***. The prototype is:

```
streamsize gcount();
```

☞ It *returns the number of characters* read by the ***last unformatted input*** operations.

- ***peek()***: Use ***peek()*** to obtain the *next character* in the input stream ***without removing*** it from that stream . It is a member of the ***input stream classes*** and has this prototype:

```
int peek();
```

⌦ It *returns the next character* in the stream. ⌦ It returns **EOF** if the ***end-of-file*** is encountered.

- ***putback()***:Use ***putback()*** to return the ***last character read*** from a stream to that stream. It is a member of the ***input stream classes***. Its prototype is:

```
istream &putback(char c);
```

⌦ Where ***c*** is the last character read.

- ***flush():*** When output is performed, data is *not immediately written* to the *physical device linked to the stream*. Instead, information is ***stored in an internal buffer until the buffer is full***. Only then are the contents of that ***buffer*** written to disk.

☞ By calling ***flush()*** you can *force the information to be physically written* to disk before the *buffer is full*. ***flush()*** is a member of the ***output stream*** classes and has this prototype:

```
ostream &flush();
```

⌦ Calls to ***flush()*** might be warranted when a program is going to be used in adverse environments (in situations where *power outages occur frequently*, for example).

- ***ios :: binary***: For *unformatted file I/O* we always use ***binary operation*** (rather than text operations **>> <<**).

☛ specifying ***ios::binary*** prevents any ***character translations*** from occurring. This is important when the binary representations of data such as ***integers***, ***float***, and ***pointers*** are stored in the file.

☛ However, it is perfectly acceptable to use the ***unformatted functions*** on a file opened in ***text mode***, but remember, some *character translations may occur*.

- ***Example 1:*** Following uses write() to write a double and a string to a file called test:

```
#include<iostream>                                  double num = 100.45;
#include<fstream>                                   char str[] = "This is a test";
#include<cstring>                                   out.write((char *) &num, sizeof(double)) ;
using namespace std;                                out.write(str , strlen(str));
int main(){   ofstream out(" test ", ios :: out | ios :: binary );   out.close();
              if(!out ) { cout << " Cannot open output file .\n"; return 1; }   return 0; }
```

☞ The ***type cast*** to (***char ****) inside the call to ***write()*** is necessary when *outputting a buffer that is not defined as a character array*. Because of C++'s strong type checking, *a pointer of one type will not automatically be converted into a pointer of another type*.

- ***Example 2:*** This program uses ***read()*** to read the file created by the program in ***Example 1***:

```
#include <iostream>                                 double num;   char str[80];
#include <fstream>                                  in.read((char *) &num , sizeof (double)) ;
using namespace std;                                in.read(str , 14) ;
int main() {   ifstream in(" test ", ios :: in | ios :: binary );   str[14] = '\0 ';   /* null terminate str */
               if (!in) { cout << " Cannot open input file .\n"; return 1; }   cout << num << ' ' << str ;
                                                    in. close();   return 0;}
```

☞ As is the case with the program in the preceding example, the ***type cast*** (***char ****) inside ***read()*** is necessary because *C++ will not automatically convert a pointer of one type to another*.

Example 3: When you use **>>** to read a string, it *stops reading when the first whitespace character is encountered*. This makes it useless for reading a string containing spaces. **`getline()`** can resolve this problem:

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

int main(){   char str[80];
              cout << " Enter your name : ";
              cin.getline(str , 79);
              cout << str << '\n';
              return 0; }
```

☞ Here, the ***delimited*** used by **`getline()`** is the ***newline***. This makes **`getline()`** act like the standard **`gets()`** function.

Example 4: In real programming situations, the functions **`peek()`** and **`putback()`** are especially useful because they let you more easily handle situations in which you *do not know what type of information is being input at any point in time*. The following program gives the flavor of this. It reads either ***strings*** or ***integers*** from a file. *The strings and integers can occur in any order*.

```
#include <iostream >
#include <fstream >
#include <cctype >
#include <cstdlib >
using namespace std;

int main(){char ch;
    ofstream out("test", ios::out | ios::binary );
    if(!out ){ cout << "Cannot open output file .\n";
               return 1; }          /* confirmation */

    char str[80], *p;
    out << 123 << "this is a test" << 23;
    out << "Hello there !" << 99 << "sdf" << endl;
    out.close();                    /* closing 1st time */

ifstream in("test", ios::in | ios::binary);
if(!in) {  cout << "Cannot open input file .\n";
           return 1; } /* confirmation */

do{ p = str;
   ch = in.peek();                          /* see what type of char is next */
   if(isdigit(ch)){ while(isdigit( *p=in.get() ))  p++;     /* read integer */
                    in.putback(*p);                  /* return char to stream */
                    *p = '\0 ';                  /* null - terminate the string */
                    cout << " Integer : " << atoi(str ); }

   else if(isalpha(ch)){while(isalpha*p=in.get() ))  p++;   /* read a string */
                        in.putback (*p);
                        *p = '\0 ';
                        cout << " String : " << str ; }
   else  in.get();                                                  /* ignore */
   cout << '\n'; } while (! in.eof());
in.close();                                                  /* final file closing */
return 0; }
```

- ☑ The **`atoi()`** is one of C's standard library function, it returns the integer equivalent of the number represented by its string argument.
- ☑ The **`isalpba()`** function returns ***nonzero*** if **`ch`** is a ***letter*** of the *alphabet*; otherwise **`0`** is returned.
- ☑ The **`isdigit()`** function returns ***nonzero*** if **`ch`** is a ***digit*** (**`0 through 9`**); otherwise **`0`** is returned.

```
#include <ctype.h>
int isdigit(int ch);
int isalpha(int ch);
```

Eg:
```
if(isalpha(ch)) printf("%c is a letter\n", ch);
if(isdigit(ch)) printf("%c is a digit\n", ch);
```

12.9 Checking I/O Status

The current status of an ***I/O*** stream is described in an object of type ***iostate***, it is an enumeration defined by ***ios*** that includes members:

[1] ***goodbit*** *(Means-No errors occurred)* ***[2]*** ***eofbit*** *(Means-End-of-file has been encountered)* ***[3]*** ***failbit*** *(Means-A nonfatal I/O error has occurred)* ***[4]*** ***badbit*** *(Means-A fatal I/O error has occurred)*

❒ There are ***two ways*** in which you can obtain I/O status information.

☞ ***First***, you can call the **`rdstate()`** function, which is a member of ***ios***. It has this prototype: **`iostate rdstate();`**

⇨ It returns the current status of the error flags.

⇨ **`rdstate()`** returns **`goodbit`** when no error has occurred. Otherwise, an error flag is returned.

☞ ***Second*** way to determine whether an error has occurred is by using one or more of these ***ios*** member functions:

[1] **`bool eof();`**
The **`eof()`** was discussed earlier.

[2] **`bool bad();`**
The **`bad()`** returns ***true*** if **`badbit`** is set.

[3] **`bool fail();`**
The **`fail()`** returns ***true*** if **`failbit`** is set.

[4] **`bool good();`**
The **`good()`** returns ***true*** if there are **`no errors`**.

☑ Otherwise ***they return false***.

❒ ***clear():*** To *clear an error* before your program continues use the ***ios*** member function **`clear()`** whose prototype is:

```
void clear(iostate flags = ios::goodbit);
```

☛ If **`flags`** is **`goodbit`** (as it is by default), all *error flags* are ***cleared***. Otherwise, set **`flags`** to the settings you desire.

❒ ***Example 1:*** Following uses **`rdstate()`** to detect a *file error* for a file named "***in***":

```
void checkstatus(ifstream &in) {  ios :: iostate i;
                                  i = in.rdstate();
                                  if(i & ios::eofbit )       cout << "EOF encountered \n";
                                  else if(i & ios::failbit ) cout << "Non - Fatal I/O error \n";
                                  else if(i & ios::badbit )  cout << "Fatal I/O error \n"; }
```

❒ ***Example 2:*** Following uses **`good()`** to detect a *file error* for a file named "***in***":

```
if(!in.good() && !in.eof()) { cout << "I/O Error ... terminating \n"; return 1; }
```

12.10 Random Access

Use the ***seekg()*** and ***seekp()*** to perform random access, these are members of the ***I*** & ***O*** stream classes, respectively. Common forms:

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

- ***off_type*** is an *integer type* defined by ***ios*** that is *capable of containing the largest valid value* that ***offset*** can have.
- ***seekdir*** is an *enumeration* defined by ***ios*** that has these values:

[1] **ios::beg** (Means-Seek from beginning) ***[2]*** **ios::cur** (Means-Seek from current location) ***[3]*** **ios::end** (Means-Seek from end)

❒ C++ I/O system manages ***two pointers*** associated with a file. The appropriate pointer is automatically applied for each ***I/O*** operation.

☞ ***get pointer***, which specifies where in the file the *next* ***input*** *operation will occur*.

☞ ***put pointer***, which specifies where in the file the *next* ***output*** *operation will occur*.

❒ ***seekg()*** and ***seekp()*** can be used in *nonsequential* fashion.

☑ ***seekg()*** moves the associated file's current ***get*** pointer ***offset*** number of bytes from the specified origin.

☑ ***seekp()*** moves the associated file's current ***put*** pointer ***offset*** number of bytes from the specified origin.

☑ Files that will be accessed via ***seekg()*** and ***seekp()*** should be opened for *binary file operations*.

❒ Use following member functions to determine the ***current position of each file pointer***.

```
pos_type tellg();          pos_type tellp();
```

- ***pos_type*** is an *integer type* defined by ***ios*** that is *capable of holding the largest value* that defines a *file position*.

❒ ***Overloaded versions of*** **seekg()** ***and*** **seekp()*:*** There are overloaded versions of ***seekg()*** and ***seekp()*** that move the file pointers to the location specified by the **return values of** ***tellg()*** and ***tellp()***. Their prototypes are:

```
istream &seekg( pos_type position );
ostream &seekp( pos_type position );
```

❒ ***Example 1:*** The following program demonstrates the ***seekp()*** function. It *allows you to change a specific character in a file*. Specify a *file name* on the ***command line***, followed by the *number of the byte* in the file you want to change, followed by the new character. Notice that the file is opened for *read/write operations*.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main (int argc, char *argv[]) {
 if(argc !=4) { cout << " Usage : CHANGE <filename > <byte > <char >\n"; return 1;}
fstream out( argv[1] , ios::in | ios::out | ios::binary );
if (!out){cout << " Cannot open file .\n"; return 1; }
out.seekp( atoi(argv [2]), ios::beg);
out.put(*argv[3]) ;
out.close();
return 0;}
```

❒ ***Example 2:*** In the above program uses ***seekg()*** to position the get pointer into the *middle of a file named "in"* and then displays the contents of that file from that point. The name of the file and the location to begin reading from are specified on the command line.

```
in.seekg( atoi(argv[2]), ios::beg );
```

NOTE : ***argv[]*** and ***argc*** are used in ***main()***'s arguments. They are called the ***command line arguments***. (***Recall: 5.4***)

12.11 Customized I/O And Files

Overloaded inserters and ***extractors***, as well as ***I/O manipulators***, can be used with *any stream* as long as they are written in a general manner. Because all C++ streams are the same, for example, the *same overloaded inserter function* can be used to output to the ***screen*** or to a ***file*** with no changes whatsoever.

❒ If you "***hard-code***" a specific stream into an ***I/O*** function, its use is, of course, *limited to only that stream*. This is why you were urged to *generalize your I/O functions* whenever possible. ***(Recall 12.5 : Make inserter/extractor as general as possible)***

❒ ***Example 1:*** In the following program, the coord class overloads the << and >> operators. Notice that you can use the operator functions to write both to the screen and to a file.

```
#include <iostream >
#include <fstream >
using namespace std;
class coord { int x, y;
public :
coord (int i, int j) { x = i; y = j; }
friend ostream &operator <<( ostream &stream , coord ob);
friend istream &operator >>( istream &stream , coord &ob);
};

ostream &operator <<( ostream &stream , coord ob){
                        stream << ob.x << ' ' << ob.y <<
'\n';
                        return stream ;}
istream & operator >>( istream &stream , coord &ob){
                        stream >> ob.x >> ob.y;
                        return stream ;}

int main() {coord o1(1, 2) , o2(3, 4);
        ofstream out(" test ");
        if (!out ) { cout << " Cannot open output file .\n";
                return 1; }
        out << o1 << o2;
        out . close ();
        ifstream in(" test ");
        if (!in) {   cout << " Cannot open input file .\n";
                return 1; }
        coord o3(0, 0) , o4(0, 0);
        in >> o3 >> o4;
        cout << o3 << o4;
        in.close ();
return 0; }
```