CHAPTER-2 part-2

C++ Programming basics: Streams based I/O (Input with cin, Output using cout), Type bool, The setw manipulator, typecasting,Type conversions, strict type checking, name space, scope resolution operator (::)

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as output operation.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as input operation.

I/O Library Header Files

Let us see the common header files used in C++ programming are:

Header File	Function and Description
<iostream>	It is used to define the cout, cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw.
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The cout is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

```
#include <iostream>
using namespace std;
int main( ) {
   char ary[] = "Welcome to C++ tutorial";
   cout << "Value of ary is: " << ary << endl;
}
```

Output:

```
Value of ary is: Welcome to C++ tutorial
```

Standard input stream (cin)

The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
#include <iostream>
using namespace std;
int main( ) {
  int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Your age is: " << age << endl;
}
```

Output:

Enter your age: 22

Your age is: 22

Standard end line (endl)

The endl is a predefined object of ostream class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
#include <iostream>
using namespace std;
int main( ) {
cout << "C++ Tutorial";
cout << " Hello World"<<endl;
cout << "End of line"<<endl;
}
```

Output:

```
C++ Tutorial Hello World
End of line
```

Boolean Type in C++:

Boolean type: The boolean type, known in C++ as bool, can only represent one of two states, true or false.

The ISO/ANSI C++ Standard has added certain new data types to the original C++ specifications.They are provided to provide better control in certain situations as well as for providing conveniences to C++ programmers.

One of the new data type is: bool

Syntax:

```
bool b1 = true;      // declaring a boolean variable with true value
```

In C++, the data type bool has been introduced to hold a boolean value, true or false.The values true or false have been added as keywords in the C++ language.

Important Points:

1.The default numeric value of true is 1 and false is 0.

2.We can use bool type variables or values true and false in mathematical expressions also.

For example:

int x = false + true + 6;

is valid and the expression on right will evaluate to 7 as false has value 0 and true will have value 1.

It is also possible to convert implicitly the data type integers or floating point values to bool type.

For example, the statements

```
bool x = 0;  // false
bool y = 100;  // true
bool z = 15.75;  // true
```

// CPP program to illustrate bool data type in C++

```
#include<iostream>
using namespace std;
int main()
{
    int x1 = 10, x2 = 20, m = 2;
    bool b1, b2;
    b1 = x1 == x2; // false

    b2 = x1 < x2; // true
   cout << "b1 is = " << b1 << "\n";
    cout << "b2 is = " << b2 << "\n";
    bool b3 = true;
if (b3)
        cout << "Yes" << "\n";
    else
        cout << "No" << "\n";

    int x3 = false + 5 * m - b3;
    cout << x3;

return 0;
}
```

Output:

```
b1 is = 0
b2 is = 1
Yes
9
```

C++ Manipulators - endl, setw, setprecision, setf

Formatting output using manipulators

Formatted output is very important in development field for easily read and understand.

C++ offers the several input/output manipulators for formatting, commonly used manipulators are given below..

Manipulator	**Declaration in**
endl	iostream.h
setw	iomanip.h
setprecision	iomanip.h
setf	iomanip.h

setw() and setfill() manipulators

setw manipulator sets the width of the field assigned for the output.

The field width determines the minimum number of characters to be written in some output representations.

If the standard width of the representation is shorter than the field width, the representation is padded with fill characters (using setfill).

setfill character is used in output insertion operations to fill spaces when results have to be padded to the field width.

Syntax

```
setw([number_of_characters]);
setfill([character]);
```

```
#include <iostream.h>
   #include <iomanip.h>
   int main()
   {     cout<<"USING setw() .............\n";
         cout<< setw(10) <<11<<"\n";
         cout<< setw(10) <<2222<<"\n";
         cout<< setw(10) <<33333<<"\n";
         cout<< setw(10) <<4<<"\n";
         cout<<"USING setw() & setfill() [type- I]...\n";
         cout<< setfill('0');
         cout<< setw(10) <<11<<"\n";
         cout<< setw(10) <<2222<<"\n";
         cout<< setw(10) <<33333<<"\n";
         cout<< setw(10) <<4<<"\n";
         cout<<"USING setw() & setfill() [type-II]...\n";
         cout<< setfill('-')<< setw(10) <<11<<"\n";
         cout<< setfill('*')<< setw(10) <<2222<<"\n";
         cout<< setfill('@')<< setw(10) <<33333<<"\n";
         cout<< setfill('#')<< setw(10) <<4<<"\n";
         return 0;    }
```

USING setw() .....

```
          11
         2222
        33333
           4
```

USING setw() & setfill() [type- I]...

0000000011

0000002222

0000033333

0000000004

USING setw() & setfill() [type-II]...

--------11

******2222

@@@@@33333

##########4

setprecision manipulator sets the total number of digits to be displayed, when floating point numbers are printed.

Syntax

```
setprecision([number_of_digits]);
cout<<setprecision(5)<<1234.537;
// output will be : 1234.5
```

On the default floating-point notation, the precision field specifies the maximum number of meaningful digits to display in total counting both those before and those after the decimal point.

Notice that it is not a minimum and therefore it does not pad the displayed number with trailing zeros if the number can be displayed with less digits than the precision.

In both the fixed and scientific notations, the precision field specifies exactly how many digits to display after the decimal point, even if this includes trailing decimal zeros. The number of digits before the decimal point does not matter in this case.

```
// setprecision example
#include <iostream>
#include <iomanip>
using namespace std;

int main () {
  double f1 =3.14159;
  cout << setprecision (5) << f1 << endl;
  cout << setprecision (9) << f1 << endl;
  return 0;
}
```

output:

The execution of this example shall display:

```
  3.1416
  3.14159
```

Example

In below example explains about setprecision function.

```
#include <iostream>
#include <iomanip>

int main () {
   double f2 =3.14159;
   std::cout << std::setprecision(5) << f2 << '\n';
   std::cout << std::setprecision(9) << f 2<< '\n';
   std::cout << std::fixed;
   std::cout << std::setprecision(5) << f 2<< '\n';
   std::cout << std::setprecision(9) << f 2<< '\n';
   return 0;
}
```

output:

```
3.1416
3.14159
3.14159
3.141590000
```

Setflag:

syntax:

```
setf([flag_value],[field bitmask]);
```

field bitmask	flag values
adjustfield	left, right or internal
basefield	dec, oct or hex
floatfield	scientific or fixed

Example

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
        cout<<"USING fixed .....................\n";
        cout.setf(ios::floatfield,ios::fixed);
        cout<< setprecision(5)<<1234.537<< endl;

        cout<<"USING scientific ..................\n";
        cout.setf(ios::floatfield,ios::scientific);
        cout<< setprecision(5)<<1234.537<< endl;
        return 0;
}
```

```
USING fixed .....................
    1234.53700
    USING scientific ..................
    1234.5
```

Example:

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
	int num=10;

	cout<<"Decimal value is :"<< num << endl;

	cout.setf(ios::basefield,ios::oct);
	cout<<"Octal value is :"<< num << endl;

	cout.setf(ios::basefield,ios::hex);
	cout<<"Hex value is :"<< num << endl;
	return 0;
}
```

Typecasting in C++

Typecasting is making a variable of one type, such as an int, act like another type, a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

Implicit conversion

Implicit conversions are automatically performed when a value is copied to a compatible type.

Example:

```
short a=2000;
int b;
b=a;// value of a is promoted from short to int.
```

Here, the value of a is promoted from short to int without the need of any explicit operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions.

Converting to int from some smaller integer type, or to double from float is known as promotion, and is guaranteed to produce the exact same value in the destination type. Other conversions between arithmetic types may not always be able to represent the same value exactly:

if a negative integer value is converted to an unsigned type, the resulting value corresponds to its 2's complement bitwise representation (i.e., -1 becomes the largest value representable by the type, -2 the second largest, ...).

The conversions from/to bool consider false equivalent to zero (for numeric types) and to null pointer (for pointer types); true is equivalent to all other values and is converted to the equivalent of 1.

If the conversion is from a floating-point type to an integer type, the value is truncated (the decimal part is removed). If the result lies outside the range of representable values by the type, the conversion causes undefined behavior.

Otherwise, if the conversion is between numeric types of the same kind (integer-to-integer or floating-to-floating), the conversion is valid, but the value is implementation-specific (and may not be portable).

Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

For non-fundamental types, arrays and functions implicitly convert to pointers, and pointers in general allow the following conversions:

Null pointers can be converted to pointers of any type

Pointers to any type can be converted to void pointers.

Pointer upcast: pointers to a derived class can be converted to a pointer of an accessible and unambiguous base class, without modifying its const or volatile qualification.

Explicit type conversion

Example:

```
#include <iostream>
using namespace std;
int main()
{
  cout<< (char)65 <<"\n";
  // The (char) is a typecast, telling the computer to interpret the 65 as a
  //  character, not as a number.  It is going to give the character output of
  //  the equivalent of the number 65 (It should be the letter A for ASCII).
  cin.get();
}
```

Output

```
A
```

One use for typecasting is when you want to use the ASCII characters. For example, what if you want to create your own chart of all 128 ASCII characters. To do this, you will need to use to typecast to allow you to print out the integer as its character equivalent.

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

int main()
{
  for ( int x = 0; x < 128; x++ ) {
    cout<< x <<". "<< (char)x <<" ";
    //Note the use of the int version of x to
    // output a number and the use of (char) to
    // typecast the x into a character
    // which outputs the ASCII character that
    // corresponds to the current number
  }
  cin.get();
}
```

Type Conversion

Type Conversion is that which converts from one datatype into another. For example converting a int into float or converting a float into double. The Type Conversion is that which automatically converts one datatype into another. We can store a large datatype into the other.

For example we can't store a float into int because a float is greater than int.

Different situations of data conversion between in compatible types:-

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type

Difference between TypeConversion and TypeCasting:

When a user can convert the one datatype into antoher then it is called as the typecasting/ Remember the typeConversion is performed by the compiler but a casting is done by the user. When we use the TypeConversion then it is called the promotion. When we use the typecasting, it means converting a large datatype into another and is called as the demotion. When we use the typecasting, we can loss some data.

Strict type checking

C++ uses very strict typechecking. A prototype must be known for each function which is called, and the call must match the prototype.

```
int main()
{
printf("HelloWorld \n");
return(0);
}
```

The program does often compile under C, though with a warning that printf() is not a known function.

Many C++ compilers will fail to produce code in such a situation.

The error is of course the missing #include<stdio.h> directive.

NAMESPACE and USING Directive Topics

Consider a situation, when we have two persons with the same name, Zara, in the same class. Whenever we need to differentiate them definitely we would have to use some additional information along with their name, like either the area, if they live in different area or their mother's or father's name, etc.

Same situation can arise in your C++ applications. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Defining a Namespace

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name {
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend (::) the namespace name as follows:

```
name::code;  // code could be variable or function.
```

Let us see how namespace scope the entities including variable and functions

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space"
<< endl;
   }
}
// second name space
namespace second_space {
   void func() {
      cout << "Inside
second_space" << endl;
   } }
```

```
int main () {
   // Calls function from first name
space.
   first_space::func();

   // Calls function from second
name space.
   second_space::func();

   return 0;
}
```

output:

```
Inside first_space
Inside second_space
```

The using directive

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code.

```
#include <iostream>
using namespace std;
// first name space
namespace first_space {
   void func() {
cout << "Inside first_space" << endl;
   }
}
// second name space
namespace second_space {
   void func() {
       cout << "Inside second_space"
<< endl;
   }
}
```

```
using namespace first_space;
int main () {
   // This calls function from first
name space.
   func();

   return 0;
}
output:
Inside first_space
```

The ‘using’ directive can also be used to refer to a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows

```
using std::cout;
```

Subsequent code can refer to cout without prepending the namespace, but other items in the std namespace will still need to be explicit as follows

```
#include <iostream>
using std::cout;

int main () {
   cout << "std::endl is used with std!" << std::endl;
   return 0;
}
```

output:

```
std::endl is used with std!
```

Names introduced in a using directive obey normal scope rules. The name is visible from the point of the using directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Scope resolution operator in C++

In C++, scope resolution operator is :: It is used for following purposes.

1) To access a global variable when there is a local variable with same name:

```
// C++ program to show that we can access a global variable using scope resolution
//operator :: when there is a local variable with same name

#include<iostream>
using namespace std;
int x;  // Global x
int main()
{
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Output:

```
Value of global x is 0
Value of local x is 10
```

```
#include<iostream>
using namespace std;
class A
{
public:
   // Only declaration
   void fun();
};

// Definition outside class using ::
void A::fun()
{
   cout << "fun() called";
}
```

```
int main()
{
   A a;
   a.fun();
   return 0;
}
```

Output:

```
fun() called
```

```
#include<iostream>
using namespace std;
class Test
{
    static int x;
public:
    static int y;
void func(int x)
    {
/* We can access class's static
/variable even if there is a local
variable*/
        cout << "Value of static x is "
<< Test::x;
 cout << "\nValue of local x is " << x;
    }
};
```

```
   //In C++, static members must be
explicitly defined
// like this
int Test::x = 1;
int Test::y = 2;
  int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);
    cout << "\nTest::y = " << Test::y;
    return 0;
}
```

Output:

```
Value of static x is 1
Value of local x is 3
Test::y = 2;
```

```
// Use of scope resolution operator in multiple inheritance.
#include<iostream>
using namespace std;
 class A
{
protected:
    int x;
public:
    A() { x = 10; }
};
 class B
{
protected:
    int x;
public:
    B() { x = 20; }  };
```

```
class C: public A, public B
{
public:
   void fun()
   {
      cout << "A's x is " << A::x;
      cout << "\nB's x is " << B::x;
   }
};
int main()
{
    C c;
    c.fun();
    return 0;
}Output:
A's x is 10
B's x is 20
```