UNIT-IV

P. Madhuravani

UNIT – IV

Pointer: Introduction, Pointers to Objects, this Pointer, Pointers to Derived Class, Virtual Functions, Pure Virtual Functions.

Streams and Files: C++ Streams, Stream Classes, Unformatted IO Operations, Formatted Console IO Operations, Managing Output with Manipulators.

Pointer

B. Madhuravani

Introduction

- Pointer variables, which are often just called pointers, are designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables.

- Working with memory locations that regular variables don't give you access to
- Working with strings and arrays
- Creating new variables in memory while the program is running
- Creating arbitrarily-sized lists of values in memory

Declaration & Initialization of a Pointer

```
data type* pointer variable;
pointer variable = value;
```

or

```
data type* pointer variable = value;
```

Example:

```
int *ptr;
int a;
ptr=&a;
```

Pointer Expression & Arithmetic

- C++ allows pointers to perform the following arithmetic operations.
- A pointer can be incremented or decremented
- Any integer can be added or subtracted from a pointer
- One pointer can be subtracted from another of it is a array.

Pointer To Objects

- item * it_ptr ; where item is a class and it_ptr is a pointer of type item.
- Object pointers are useful in creates objects at run time.
- An object pointer can be used to access the public members of an object.

Pointer To Objects

```
class item
{
     int code;
     float  price;
   public:
     void getdata( int a, float b )
      { code =a; price = b;}
     void show( void )
     { cout << "Code :" << code<<endl;
        << "Price :" << price << endl; }
};
item x;
item *ptr = &x;
```

- We can refer to the member functions of item in two ways:
  - Using dot operator and object.
    - x.getdata(100,75.50);
    - x.show( );
  - Using arrow operator and object pointer.
    - ptr -> getdata(100, 75.50);
    - ptr -> show( );
  - Since *ptr is an alias of x
    - (*ptr).show( );

Pointer To Objects

We can also create the objects using pointers and new operator as:

item * ptr = new item ;

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to ptr.

We can also create an array of objects using pointers

Item *ptr = new item[10];

Creates memory space for an array of 10 objects of item.

this Pointer

C++ uses a unique keyword called this to represent an object that invokes a member function.

This unique pointer is automatically passed to a member function when it is called.

The pointer this acts as an implicit argument to all the member functions.

One important application of the pointer this is to return the object it points to. return *this;

Pointer to Derived Classes

- We can use pointers not only to the base objects but also to the objects of derived classes.
- Pointers to objects of a base class are type-compatible with pointers to objects of a derived class.
- Therefore, single pointer variable can be made to point to objects belonging to different classes.

Pointer to Derived Classes

- If B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be a pointer to D.

```
B * bptr ;
B b;
D d;
bptr = &b;
```

We can also make bptr = &d;

Pointer to Derived Classes

- Using bptr, we can access only those members which are inherited from B and not the members that originally belong to D.
- In case a member of D has the same name as one of the members of B, then any reference to that member by bptr will always access the base class member.

Early Binding or Compile Time Polymorphism

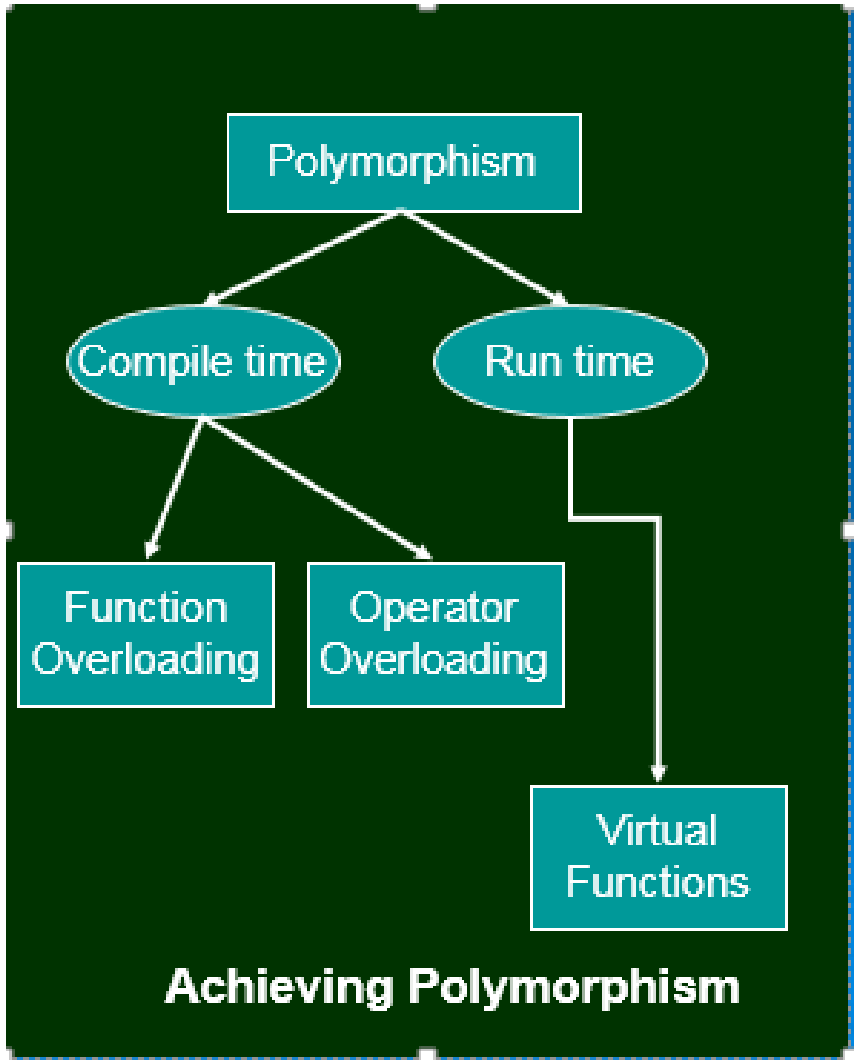
- The concept of polymorphism is implemented using overloaded functions and operators.
- The overloaded member functions are selected for invoking by matching arguments, both type and numbers.
- This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself.
- This is called early binding or static binding or static linking.

Late Binding or Run Time Polymorphism

```
class A
{
    int x;
  public:
    void show( ) {……}
};
class B : public A
{
    int y;
  public:
    void show( ) {……}
};
```

- Since the prototype of show( ) is the same in both the places, the function is not overloaded and therefore static binding does not apply.
- The class resolution operator is used to specify the class while invoking the functions with the derived class.
- Appropriate member function is selected while the program is running.

Late Binding or Run Time Polymorphism



- The appropriate version of function will be invoked at runtime.
- Since the function is linked with a particular class much later after the compilation, this process is termed as late binding.
- This also known as dynamic binding, since the selection of appropriate function is done dynamically at runtime.

Virtual Functions

- Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.
- An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.
- This necessitates the use of a single pointer variable to refer to the objects of different classes.

Virtual Functions

- We use pointer to base class to refer to all the derived objects.
- When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration.

- When a function made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.

Virtual Functions

- One important point to remember is that, we must access virtual functions through the use of a pointer declared as a pointer to the base class.
- Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Pure Virtual Functions

- A pure virtual function is a function declared in a base class that has no definition relative to the base class.
- A do-nothing function may be defined as follows:

  virtual void display( ) = 0;
- A class containing pure virtual functions cannot be used to declare any objects of its own. – abstract classes.

Pure Virtual Functions

- The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Streams and Files

B. Madhuravani

Managing Console I/O Operations

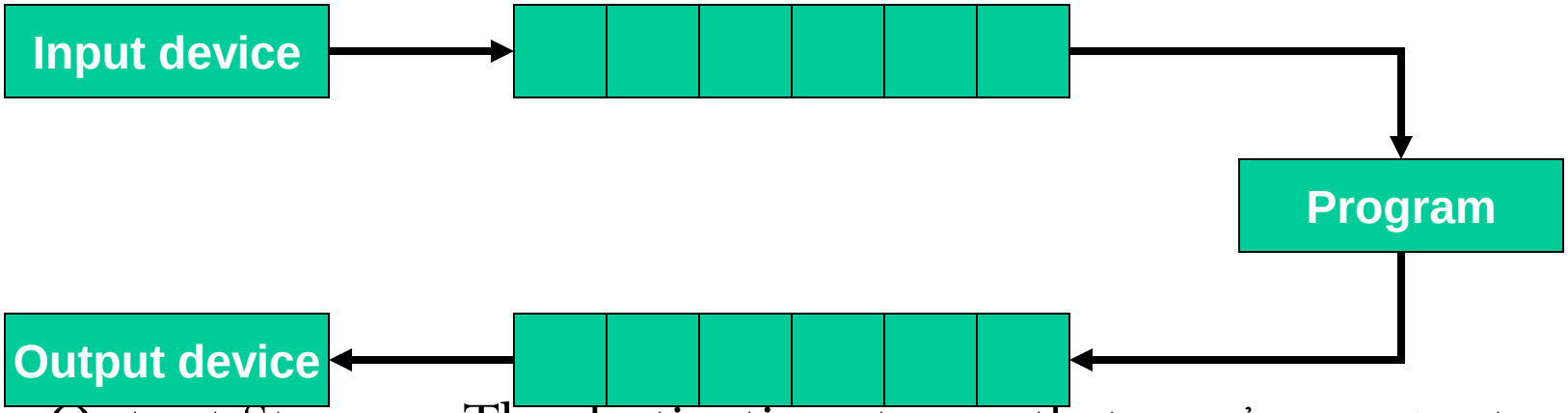
- C++ uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.
- C++ support all of C's rich set of I/O functions.

C ++ Stream

- Stream is an interface supplied by the I/O system of C++ between the programmer and the actual device being accessed.
- It will work with devices like terminals, disks and tape drives.
- A stream is a sequence of bytes.
- It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

C ++ Stream

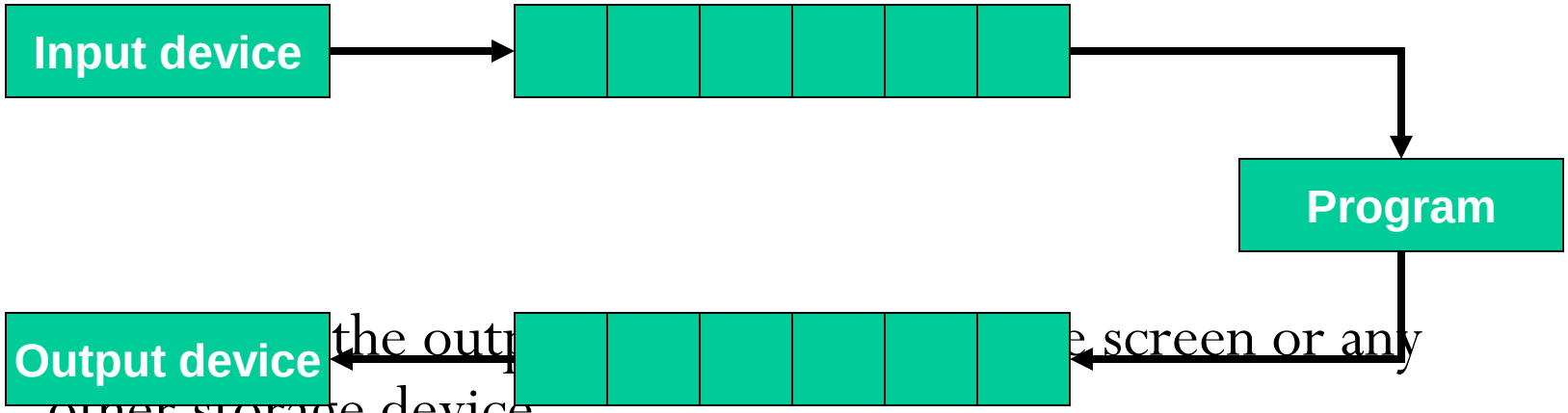
- Input Stream - The source stream that provides data to the program.



- Output Stream - The destination stream that receives output from the program.

C ++ Stream

- The data in the input stream can come from keyboard or any other storage device.



- the outp e screen or any other storage device.

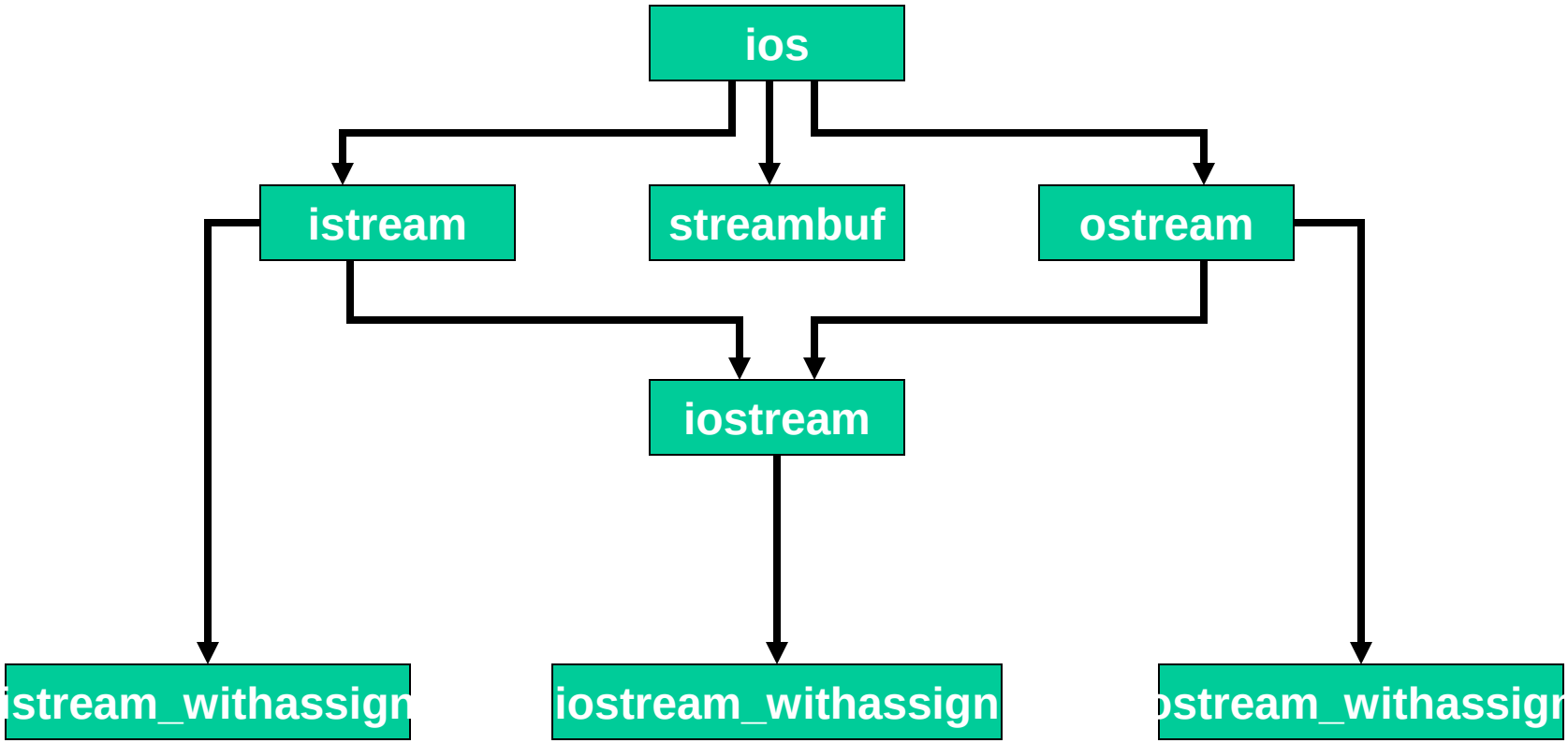
C ++ Stream Classes

- The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files.

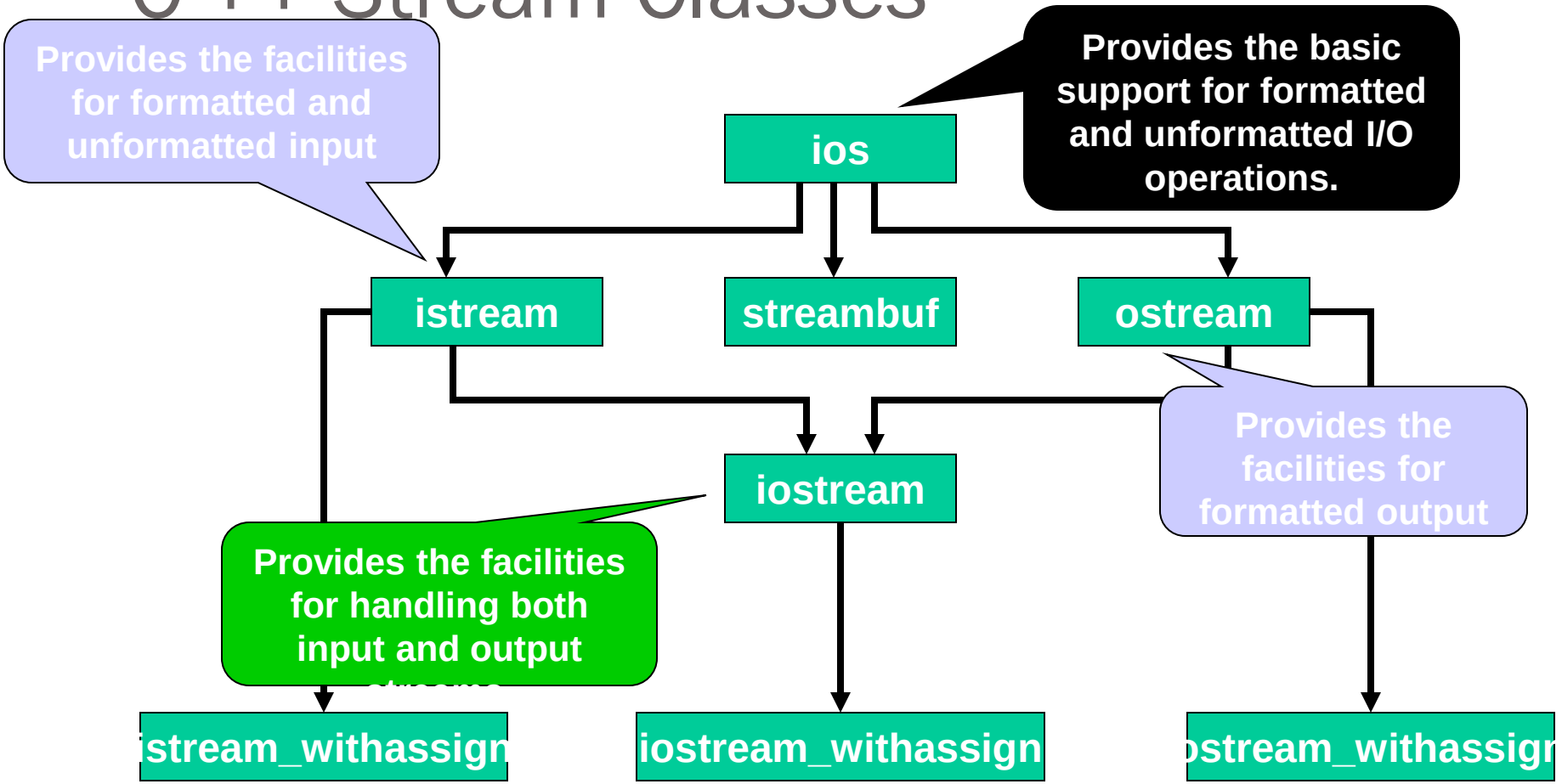
- These classes are called stream classes.

- These classes are declared in the header file iostream.

C ++ Stream Classes



C ++ Stream Classes



Unformatted I/O Operations

Overloaded Operators >> and <<

- The objects cin and cout are used for input and output of data of various types.
- By overloading the operators >> and <<.
- >> operator is overloaded in the istream class.
- << operator is overloaded in the ostream class.
- This is used for input data through keyboard.

Unformatted I/O Operations

Overloaded Operators >> and <<

cin >> variable1 >> varibale2 ... >> variableN

where variable1, variable2, …, variableN are valid C++ variable names.

cout << item1 << item2 << … << itemN

Where item1, item2, …,itemN may be variables or constants of an basic type.

Unformatted I/O Operations

put( ) and get( ) Functions

- get( ) and put( ) are member functions of istream and ostream classes.
- For single character input/output operations.
- There are two types of get( ) functions:
  - get(char*) → Assigns the input character to its argument.
  - get(void) → Returns the input character.
    - char c; cin.get(c) c = cin.get( );
  - put( ) → used to output a line of text, character by character.
    - char c; cout.put('x'); cout.put(c);

Unformatted I/O Operations

getline( ) and write( ) Functions

- getline( ) function reads a whole line of text that ends with a newline character.
- cin.getline(line, size);
- Reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read.
- write( ) function displays an entire line of text.
- cout.write(line, size);
- write( ) also used to concatenate strings.

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.

ios member functions

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.

unsetf( ) → to clear the flags specified.

Manipulators

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

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

- **setw( )**
- **setprecision( )**
- **setfill( )**
- **setiosflags( )**
- **resetiosflags( )**

width( ) → Defining Field Width

To define the width of a field necessary for the output of an item.

Since it is a member function, we have to use an object to invoke it.

cout.width(w)

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

The output will be printed in a field of w characters wide at the right end of the field.

width( ) → Defining Field Width

The width( ) function can specify the field width for only one item – item that follows immediately.

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


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

The field should be specified for each item separately.

precision( ) → Setting Precision

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

- cout.precision(d);
  - *Where d is the number of digits to the right of the decimal point.*

```
cout.precision(3);
cout << sqrt(2) << endl;
cout << 3.14159 << endl;
cout <<2.50032 << endl;
```

1.141 → truncated
3.142 → rounded
2.5 →o trailing zeros

precision( ) → Setting Precision

- Unlike width( ), precision ( ) retains the setting in effect until it is reset.

- We can also combine the field specification with the precision setting.

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

fill( ) → Filling and Padding

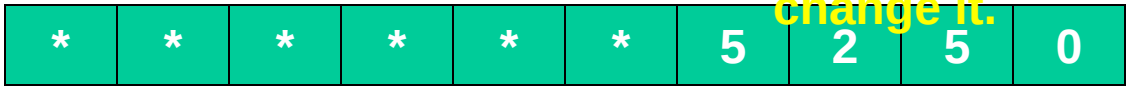
- When printing values with larger field width than required by the values, the unused positions of the field are filled with white spaces, by default.
- fill( ) function can be used to fill the unused positions by any desired character.

cout.fill(ch);

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

```
cout.fill(' * ');
cout.wdith(10);
cout << 5250 << endl;
```

Like precision( ), fill( ) stays in effect till we change it.



setf( ) → Formatting Flags, Bit-fields

- When the function width( ) is used, the value will be printed right-justified in the created width.
- setf( )function is used to print values in left-justified.

- cout.setf(arg1, arg2)

  eg:

  cout.fill(' * ');

  cout.setf(ios :: left, ios :: adjustfiel

  cout.wdith(10);

  cout << "TABLE 1" << endl;

arg1 is formatting flags defined in the class ios and arg2 is bit field constant in the ios class.

T	A	B	L	E		1	*	*	*

setf( ) → Formatting Flags, Bit-fields

- When the function width( ) is used, the value will be
  t-justifie
  on is use
  ified.

The formatting flag specifies the format action required for the

Bit field specifies the group to which the formatting flag belongs.

- cout.s arg1, arg2)

  eg:

  cout.fill(' * ');

  cout.setf(ios :: left, ios :: adjustfiel

  cout.wdith(10);

  cout << "TABLE 1" << endl;

arg1 is formatting flags defined in the class ios and arg2 is bit field constant in the ios class.

T	A	B	L	E		1	*	*	*

Managing Output with Manipulators

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

- Some manipulators are more convenient to use than the member functions and flags of ios.

- Two or more manipulators can be used as a chain in one statement.

```
cout << manip1 << manip2 << maip3 << item;
cout << manip1 << item1 << manip2 << item2;
```

Managing Output with Manipulators

- Manipulators and their meanings

Manipulators	Meaning	Equivalent
setw(int w)	Set the field width to w	width( )
setprecision(int d)	Set floating point precision to d	precision( )
setfill(int c)	Set the fill character to c	fill( )

```
cout << setw(5) << setprecision(2) << 1.2345
     << setw(10) << setprecision(4) << sqrt(2);
```

- We can jointly use the manipulators and the ios functions in a program.

Manipulators & ios Member Functions

- The ios member function return the previous format state which can be used later.

- But the manipulator does not return the previous format state.

```
cout.precision(2);  // previous state.
int p =cout.precision(4); // current state, p=2.
cout.precision(p); // change to previous state
```

Designing Our Own Manipulators

- We can design our own manipulators for certain special purposes.

```
ostream & manipulator ( ostream & output)
{
        ……  (code)
        return output;
}
ostream & unit (ostream & output)
{
        output << "inches";
        return output;
}
cout << 36 << unit;
```

→ will produce "36 inches".

Designing Our Own Manipulators

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

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

This function defines a manipulator called show that turns on the flags showpoint and showpos declared in the class ios and sets the field width to 10.