Program 9.11 produces the following output:

```
bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400
```

note

We have used the statement

```
bptr -> show();
```

two times. First, when **bptr** points to the base object, and second when **bptr** is made to point to the derived object. But, both the times, it executed **BC::show()** function and displayed the content of the base object. However, the statements

```
dptr -> show();
((DC *) bptr) -> show();        // cast bptr to DC type
```

display the contents of the **derived** object. This shows that, although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.

9.6 Virtual Functions

As mentioned earlier, 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. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism?. It is achieved using what is known as 'virtual' functions.

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 is 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. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 9.12 illustrates this point.

VIRTUAL FUNCTIONS

```
#include <iostream>

using namespace std;

class Base
{
  public:
       void display() {cout << "\n Display base ";}
       virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
       void display() {cout << "\n Display derived";}
       void show() {cout << "\n show derived";}
};

int main()
{
       Base B;
       Derived D;
       Base *bptr;

       cout << "\n bptr points to Base \n";
       bptr = &B;
       bptr -> display();    // calls Base version
       bptr -> show();       // calls Base version

       cout << "\n\n bptr points to Derived\n";
       bptr = &D;
       bptr -> display();    // calls Base version
       bptr -> show();       // calls Derived version

       return 0;
}
```

PROGRAM 9.12

The output of Program 9.12 would be:

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

note

When **bptr** is made to point to the object **D**, the statement

```
bptr -> display();
```

calls only the function associated with the **Base** (i.e. **Base :: display**( )), whereas the statement

```
bptr -> show();
```

calls the **Derived** version of **show()**. This is because the function **display()** has not been made **virtual** in the **Base** class.

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. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions?. We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where **virtual** functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as **media** that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure 9.2 shows the class hierarchy for the book shop.

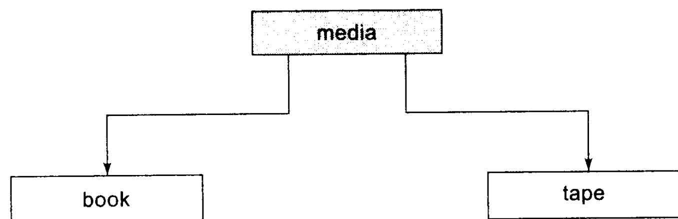

Fig. 9.2 ⇔ *The class hierarchy for the book shop*

```
        tape tapel(title, price, time);

        media* list[2];
        list[0] = &book1;
        list[1] = &tapel;

        cout << "\n MEDIA DETAILS";

        cout << "\n ......BOOK......";
        list[0] -> display(); // display book details

        cout << "\n ......TAPE......";
        list[1] -> display(); // display tape details

        result 0;
}
```

PROGRAM 9.13

The output of Program 9.13 would be:

```
ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
......BOOK......
Title:Programming_in_ANSI_C
Pages: 400
Price: 88

.....TAPE......
Title: Computing_Concepts
Play time: 55mins
Price: 90
```

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

9.7 Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder*. For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display() = 0;
```

Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. 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.

9.6 *What are the applications of **this** pointer?*

9.7 *What is a virtual function?*

9.8 *Why do we need virtual functions?*

9.9 *When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?*

9.10 *State which of the following statements are TRUE or FALSE.*

(a) *Virtual functions are used to create pointers to base classes.*

(b) *Virtual functions allow us to use the same function call to invoke member functions of objects of different classes.*

(c) *A pointer to a base class cannot be made to point to objects of derived class.*

(d) ***this** pointer points to the object that is currently used to invoke a function.*

(e) ***this** pointer can be used like any other pointer to access the members of the object it points to.*

(f) ***this** pointer can be made to point to any object by assigning the address of the object.*

(g) *Pure virtual functions force the programmer to redefine the virtual function inside the derived classes.*

Debugging Exercises

9.1 Identify the error in the following program.

```
#include <iostream.h>
class Info
{
    char *name;
    int number;
public:
    void getInfo()
    {
        cout << "Info::getInfo ";
        getName();
    }

    void getName()
    {
        cout << "Info::getName ";
    }
};
```

```
class Name: public Info
{
     char *name;
public:
     void getName()
     {
            cout << "Name::getName ";
     }
};

void main()
{
     Info *p;
     Name n;
     p = n;
     p->getInfo();
}
/*
```

9.2 Identify the error in the following program.

```
#include <iostream.h>
class Person
{
     int age;
public:
     Person()
     {
     }
     Person(int age)
     {
            this.age = age;
     }
     Person& operator < (Person &p)
     {
            return age < p.age ? p: *this;
     }
     int getAge()
     {
            return age;
```

```
        int a,b;
   public:
        A(int x = 0, int y)
        {
          a = x;
          b = y;
        }
        virtual void print();
};
class B: public A
{
  private:
     float p,q;
  public:
     B(int m, int n, float u, float v)
      {
          p = u;
          q = v;
      }
       B() {p = q = 0;}
       void input(float u, float v);
       virtual void print(float);
};
void A::print(void)
{
     cout << A values: << a <<""<< b <<"\n";
}
void B::print(float)
{
     cout <<B values:<< u <<""<< v <<"\n";
}
void B::input(float x, float y)
{
       p = x;
       q = y;
 }
main()
{
       A a1(10,20), *ptr;
       B b1;
       b1.input(7.5,3.142);

       ptr = &a1;
       ptr->print();

       ptr = &b1;
       ptr->print();
     }
```

Programming Exercises

9.1 *Create a base class called* ***shape****. Use this class to store two* ***double*** *type values that could be used to compute the area of figures. Derive two specific classes called* ***triangle*** *and* ***rectangle*** *from the base* ***shape****. Add to the base class, a member function* ***get_data()*** *to initialize base class data members and another member function* ***display_area()*** *to compute and display the area of figures. Make* ***display_area()*** *as a virtual function and redefine this function in the derived classes to suit their requirements.*

Using these three classes, design a program that will accept dimensions of a triangle or a rectangle interactively, and display the area.

Remember the two values given as input will be treated as lengths of two sides in the case of rectangles, and as base and height in the case of triangles, and used as follows:

```
Area of rectangle = x * y
Area of triangle = 1/2 * x * y
```

9.2 *Extend the above program to display the area of circles. This requires addition of a new derived class 'circle' that computes the area of a circle. Remember, for a circle we need only one value, its radius, but the get_data() function in the base class requires two values to be passed. (Hint: Make the second argument of get_data() function as a default one with zero value.)*

9.3 *Run the above program with the following modifications:*

(a) *Remove the definition of* ***display_area()*** *from one of the derived classes.*

(b) *In addition to the above change, declare the* ***display_area()*** *as* ***virtual*** *in the base class* ***shape****.*

Comment on the output in each case.

10.3 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*. Figure 10.2 shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file *iostream*. This file should be included in all the programs that communicate with the console unit.

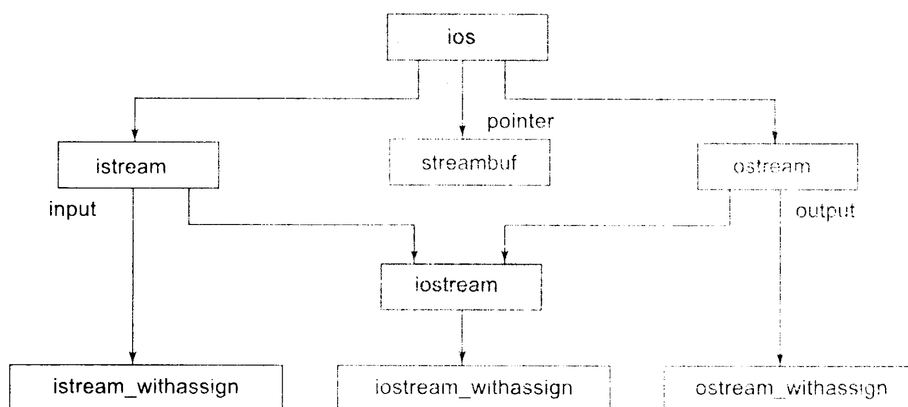

Fig. 10.2 ⇔ *Stream classes for console I/O operations*

As seen in the Fig. 10.2, **ios** is the base class for **istream** (input stream) and **ostream** (output stream) which are, in turn, base classes for **iostream** (input/output stream). The class **ios** is declared as the virtual base class so that only one copy of its members are inherited by the **iostream**.

The class **ios** provides the basic support for formatted and unformatted I/O operations. The class **istream** provides the facilities for formatted and unformatted input while the class **ostream** (through inheritance) provides the facilities for formatted output. The class **iostream** provides the facilities for handling both input and output streams. Three classes, namely, **istream_withassign, ostream_withassign**, and **iostream_withassign** add assignment operators to these classes. Table 10.1 gives the details of these classes.

10.4 Unformatted I/O Operations

Overloaded Operators >> and <<

We have used the objects **cin** and **cout** (pre-defined in the *iostream* file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types. The >> operator is overloaded in the

Table 10.1 *Stream classes for console operations*

Class name	*Contents*
ios (General input/output stream class)	▪ Contains basic facilities that are used by all other input and output classes ▪ Also contains a pointer to a buffer object **(streambuf** object) ▪ Declares constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	▪ Inherits the properties of **ios** ▪ Declares input functions such as **get(), getline()** and **read()** ▪ Contains overloaded extraction operator >>
ostream (output stream)	▪ Inherits the properties of ios ▪ Declares output functions **put()** and **write()** ▪ Contains overloaded insertion operator <<
iostream (input/output stream)	▪ Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions
streambuf	▪ Provides an interface to physical devices through buffers ▪ Acts as a base for **filebuf** class used ios files

istream class and << is overloaded in the **ostream** class. The following is the general format for reading data from the keyboard:

```
cin >> variable1 >> variable2 >> .... >> variableN
```

variable1, variable2, ... are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be:

```
data1 data2 ...... dataN
```

The input data are separated by white spaces and should match the type of variable in the **cin** list. Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For example, consider the following code:

```
int code;
cin >> code;
```

Suppose the following data is given as input:

```
4258D
```

The operator will read the characters upto 8 and the value 4258 is assigned to **code.** The character D remains in the input stream and will be input to the next **cin** statement. The general form for displaying data on the screen is:

```
cout << item1 << item2 << .... << itemN
```

The items *item1* through *itemN* may be variables or constants of any basic type. We have used such statements in a number of examples illustrated in previous chapters.

put() and get() Functions

The classes **istream** and **ostream** define two member functions **get()** and **put()** respectively to handle the single character input/output operations. There are two types of **get()** functions. We can use both **get(char *)** and **get(void)** prototypes to fetch a character including the blank space, tab and the newline character. The **get(char *)** version assigns the input character to its argument and the **get(void)** version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object.

Example:

```
char c;
cin.get(c);              // get a character from keyboard
                         // and assign it to c
while(c != '\n')
{
      cout << c;         // display the character on screen
      cin.get(c);        // get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character. The above **while** loop will not work properly if the statement

```
cin >> c;
```

is used in place of

```
cin.get(c);
```

note

Try using both of them and compare the results.

The **get(void)** version is used as follows:

```
.....
char c;
c = cin.get();    // cin.get(c); replaced
.....
.....
```

The value returned by the function **get**() is assigned to the variable **c**.

The function **put**(), a member of **ostream** class, can be used to output a line of text, character by character. For example,

```
cout.put('x');
```

displays the character **x** and

```
cout.put(ch);
```

displays the value of variable **ch.**

The variable **ch** must contain a character value. We can also use a number as an argument to the function **put**(). For example,

```
cout.put(68);
```

displays the character D. This statement will convert the **int** value 68 to a **char** value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get(c);             // read a character

while(c != '\n')
{
        cout.put(c);    // display the character on screen
        cin.get(c);
}
```

Program 10.1 illustrates the use of these two character handling functions.

CHARACTER I/O WITH get() AND put()

```
#include <iostream>

using namespace std;

int main()
{
  int count = 0;
  char c;

  cout << "INPUT TEXT\n";

  cin.get(c);

  while(c != '\n')
  {
        cout.put(c);
        count++;
        cin.get(c);
  }
  cout << "\nNumber of characters = " << count << "\n";

  return 0;
}
```

PROGRAM 10.1

Input

```
    Object Oriented Programming
```

Output

```
    Object Oriented Programming
    Number of characters = 27
```

note

When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement **cin.get(c);** and displays it using the statement **cout.put(c);**. The process is terminated when the newline character is encountered.

getline() and write() Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions **getline**() and **write**(). The **getline**() function reads a whole line of text that ends with a newline character (transmitted by the RETURN key). This function can be invoked by using the object **cin** as follows:

```
cin.getline (line, size);
```

This function call invokes the function **getline()** which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size-1 characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character. For example, consider the following code:

```
char  name[20];
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

```
Bjarne Stroustrup <press RETURN>
```

This input will be read correctly and assigned to the character array **name**. Let us suppose the input is as follows:

```
Object Oriented Programming  <press RETURN >
```

In this case, the input will be terminated after reading the following 19 characters:

```
Object Oriented Pro
```

Remember, the two blank spaces contained in the string are also taken into account.

We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember **cin** can read strings that do not contain white spaces. This means that **cin** can read just one word and not a series of words such as "Bjarne Stroustrup". But it can read the following string correctly:

```
Bjarne_Stroustrup
```

After reading the string, **cin** automatically adds the terminating null character to the character array.

Program 10.2 demonstrates the use of >> and

```
#include <iostream>

using namespace std;
```

(Contd)

```
int main()
{
    int size = 20;
    char city[20];

    cout << "Enter city name: \n";
    cin  >> city;
    cout << "City name:" << city << "\n\n";

    cout << "Enter city name again: \n";
    cin.getline(city, size);
    cout << "City name now: " << city << "\n\n";

    cout << "Enter another city name: \n";
    cin.getline(city, size);
    cout << "New city name: " << city << "\n\n";

    return 0;
}
```

PROGRAM 10.2

The output of Program 10.2 would be:

```
First run
     Enter city name:
     Delhi
     City name: Delhi

     Enter city name again:
     City name now:
     Enter another city name:
     Chennai
     New city name: Chennai

Second run
     Enter city name:
     New Delhi
     City name: New

     Enter city name again:
     City name now: Delhi

     Enter another city name:
     Greater Bombay
     New city name: Greater Bombay
```

note

During first run, the newline character '\n' at the end of "Delhi" which is waiting in the input queue is read by the **getline()** that follows immediately and therefore it does not wait for any response to the prompt 'Enter city name again:'. The character '\n' is read as an empty line. During the second run, the word "Delhi" (that was not read by cin) is read by the function **getline()** and, therefore, here again it does not wait for any input to the prompt 'Enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second **cin.getline(city,size);** statement.

The **write()** function displays an entire line and has the following form:

```
cout.write (line, size)
```

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display. Note that it does not stop displaying the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line. Program 10.3 illustrates how **write()** method displays a string.

DISPLAYING STRINGS WITH write()

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    char * string1 = "C++ ";
    char * string2 = "Programming";
    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";
    }
```

(Contd)

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

    cout << "\n";

    // crossing the boundary
    cout.write(string1,10);

    return 0;
}
```

PROGRAM 10.3

Look at the output of Program 10.3:

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr
```

The last line of the output indicates that the statement

```
cout.write(string1, 10);
```

displays more characters than what is contained in **string1.**

It is possible to concatenate two strings using the **write**() function. The statement

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