Traditional Procedural Programming
Data and functions are kept separate from the data they process. This has significant effect on the way a program handles data:
- The programmer must ensure that the data are initialized with suitable values before use and that suitable data are passed to a function when it is called
- If the data representation is changed, the corresponding functions must also be modified
Both of these points can lead to errors and neither support low program maintenance requirements.

Objects
OOP shifts the focus of attention to the objects, that is, to the aspects on which the problem is centered. OOP objects combine data (properties) and functions (capacities). A class defines a certain object type by defining both the properties and the capacities of the objects of that type. Object communicates by sending each other “messages”, which in turn activate another object’s capacities.

Advantages of OOP
OOP is centered around objects that encapsulate both data and the functions that operate on them. The advantages are:-
- Reduced susceptibility to errors: an object controls access to its own data. More specifically, an object can reject erroneous access attempts.
- Easy to re-use: object maintain themselves and can therefore be used as building blocks for other programs
- Low maintenance requirement: an object type can modify its own internal data representation without requiring changes to the application

C++ Class
A class is the collection of related data and function under a single name. A C++ program can have any number of classes. When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.

A Class is a blueprint for objects
When a class is defined, no memory is allocated. Only when an object is instantiated (created), memory is allocated.

Defining the Class in C++
Class is defined in C++ programming using keyword class followed by identifier (name of class). Body of class is defined inside curly brackets are terminated by semicolon at the end in similar way as structure.

```cpp
class class_name
{
    // some data (data members)
    int data1;
    float data2;
    // some functions (member function)
}
```

Example of Class in C++

```cpp
class temp
{
    private:
        int data1;
        float data2;

```
public:
    void func1()
    { data1=2; }
    float func2()
    { data2=3.5;
      return data;   } 
    
Explanation
As mentioned, definition of class starts with keyword class followed by name of class (temp) in this case. The body of that class is inside the curly brackets and terminated by semicolon at the end. There are two keywords: private and public mentioned inside the body of class.

Keywords: private and public
Keyword private makes data and functions private and keyword public makes data and functions public. Private data and functions are accessible inside that class only whereas, public data and functions are accessible both inside and outside the class. This feature in OOP is known as data hiding. If programmer mistakenly tries to access private data outside the class, compiler shows error which prevents the misuse of data. Generally, data are private and functions are public.

C++ Objects
When class is defined, only specification for the object is defined. Object has same relationship to class as variable has with the data type. Objects can be defined in similar way as structure is defined.

Syntax to Define Object in C++

```cpp
class_name variable name;
```

For the above defined class temp, objects for that class can be defined as:
    temp obj1,obj2;
Here, two objects (obj1 and obj2) of temp class are defined.

Data Member and Member Functions
The data within the class is known as data member. The function defined within the class is known as member function. These two technical terms are frequently used in explaining OOP. In the above class temp, data1 and data2 are data members and func1() and func2() are member functions.

Accessing Data Members and Member functions
Data members and member functions can be accessed in similar way the member of structure is accessed using member operator(.). For the class and object defined above, func1() for object obj2 can be called using code: obj2.func1();
Similarly, the data member can be accessed as: object_name.data_memeber;

Note: You cannot access the data member of the above class temp because both data members are private so it cannot be accessed outside that class.
Examples to Explain Working of Object and Class in C++ Programming

// Program example using class to calculate area of a rectangle
#include <iostream>
using namespace std;

class CRectangle {
private:
    int x, y;
public:
    void set_values (int a, int b) {
        x = a;
        y = b;
    }
    void CRectangle::set_values (int a, int b) {
        x = a;
        y = b;
    }
    int area () { return (x*y); }
};

void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

class temporary {
private:
    int x;
    float y;
public:
    temporary(): x(5), y(5.5) /* Constructor */
    {
        /* Body of constructor */
    }
};

// Program to illustrate working of Objects and Class in C++ Programming
#include <iostream>
using namespace std;
class temp[
private:
    int data1;
    float data2;
public:
    void int_data(int d) {
        data1=d;
        cout<<"Number: "<<data1;    }
    float float_data()
    {
        cout<<"\nEnter data: ";
        cin>>data2;
        return data2;
    }
];

main()
{
    temp obj1, obj2;
    obj1.int_data(12);
    cout<<"You entered "<<obj2.float_data();    }

CHECK

1. Write a program that uses a simple class that holds the radius of a circle as its data member. The program then sets the radius and print the area of the circle. Create two function members, one to set the radius and the other to calculate the area.

2. Write a program that uses a simple class, which calculates the voltage, given the values of current and resistance. Create two function members, one to set the values of current and resistance, the other to calculate the voltage.

Defining Member Function Outside the Class

A large program may contain many member functions. For the clarity of the code, member functions can be defined outside the class. To do so, member function should be declared inside the class (function prototype should be inside the class). Then, the function definition can be defined using scope resolution operator :: as illustrated above.

C++ Constructor

Constructors are the special type of member function that initializes the object automatically when it is created. Compiler identifies that the given member function is a constructor by its name and return type. Constructor has same name as that of class and it does not have any return type. This will avoid using an objects with garbage in it because no values has been assigned yet when the objects are created.

..... ...

class temporary{
private:
    int x;
    float y;
public:
    temporary(): x(5), y(5.5) /* Constructor */
    {
        /* Body of constructor */
    }
};
int main()
{ Temporary t1;
.... ....
}

Working of Constructor
In the above pseudo-code, temporary() is a constructor. When the object of class temporary is created, constructor is called and x is initialized to 5 and y is initialized to 5.5 automatically.
You can also initialize data member inside the constructor’s function body as below. But, this method is not preferred.

temporary(){
    x=5;
    y=5.5;
}

/* This method is not preferred. */

Use of Constructor in C++
Suppose you are working on 100’s of objects and the default value of a data member is 0. Initializing all objects manually will be very tedious. Instead, you can define a constructor which initializes that data member to 0. Then all you have to do is define object and constructor will initialize object automatically. This type of situation arises frequently while handling array of objects. Also, if you want to execute some codes immediately after object is created, you can place that code inside the body of constructor.

Constructor Example
Programs to demonstrate the working of constructor in C++ Programming.

CHECK
3. Write a program that uses a simple class that holds the radius of a circle as its data member. The program than sets the radius and print the area of the circle. Create a constructor to set the radius and a member function to calculate the area.
Constructor Overloading

Constructor can be overloaded in similar way as function overloading. Overloaded constructors have same name (name of the class) but different number of argument passed. Depending upon the number and type of argument passed, specific constructor is called. Since, constructor is called when object is created. Argument to the constructor also should be passed while creating object. Here is the modification of above program to demonstrate the working of overloaded constructors.

Programs to demonstrate the working of overloaded constructors.

```cpp
// overloading class constructors
#include <iostream>
#include <string>

using namespace std;

class CRectangle
{
public:
    CRectangle(int a, int b); // Constructor with two argument
    int area(); // Constructor with no argument

private:
    int width, height;
};

CRectangle::CRectangle(int a, int b) // Constructor with two argument
{
    width = a;
    height = b;
}

CRectangle::CRectangle() // Constructor with no argument
{
    width = 5;
    height = 5;
}

main()
{
    CRectangle rect(3,4);
    cout << "rect area: " << rect.area() << endl;
    CRectangle rectb;
    cout << "rectb area: " << rectb.area() << endl;
}
```

Default Copy Constructor

An object can be initialized with another object of same type. Let us suppose the above program. If you want to initialize an object A3 so that it contains same value as A2. Then, this can be performed as:

```cpp
int main()
{
    Area A1,A2(2,1);
    Area A3=A2;  /* Copies the content of A2 to A3 */
    OR,
    Area A3=A2;  /* Copies the content of A2 to A3 */
}
```

You might think you may need some constructor to perform this task. But, no additional constructor is needed. It is because this constructor is already built into all classes.
Inheritance in C++ Programming

Inheritance is one of the key feature of object-oriented programming including C++ which allows user to create a new class (derived class) from a existing class (base class). The derived class inherits all features from a base class and it can have additional features of its own.

Concept of Inheritance in OOP

Suppose, you want to calculate either area, perimeter or diagonal length of a rectangle by taking data (length and breadth) from user. You can create three different objects (Area, Perimeter and Diagonal) and asks user to enter length and breadth in each object and calculate corresponding data. But, the better approach would be to create an additional object Rectangle to store value of length and breadth from user and derive objects Area, Perimeter and Diagonal from Rectangle base class. It is because, all three objects Area, Perimeter and diagonal are related to object Rectangle and you don’t need to ask user the input data from these three derived objects as this feature is included in base class.

Implementation of Inheritance in C++ Programming

```cpp
class Rectangle
{
    .... ...
};

class Area : public Rectangle
{
    .... ...
};

class Perimeter : public Rectangle
{
    .... ...
};
```

In the above example, class Rectangle is a base class and classes Area and Perimeter are the derived from Rectangle. The derived class appears with the declaration of class followed by a colon, the keyword public and the name of base class from which it is derived. Since, Area and Perimeter are derived from Rectangle, all data member and member function of base class Rectangle can be accessible from derived class.
Note: Keywords private and protected can be used in place of public while defining derived class (will be discussed later).
This example calculates the area and perimeter a rectangle using the concept of inheritance.

/* C++ Program to calculate the area and perimeter of rectangles using concept of inheritance. */

#include <iostream>
using namespace std;

class Rectangle
{   protected:
    float length, breadth;

class Area : public Rectangle
{   public:
    float calc()
    { return length*breadth; } };

class Perimeter : public Rectangle
{   public:
    float calc()
    { return 2*(length+breadth); } };

int main()
{   cout<<"Enter data for first rectangle to find area.\n";
    Area a;
    cout<<"Area = " << a.calc() <<" square meter\n";
    cout<<"Enter data for second rectangle to find perimeter.\n";
    Perimeter p;
    cout<<"\nPerimeter = " << p.calc() <<" meter";
    return 0; }

Explanation of Program
In this program, classes Area and Perimeter are derived from class Rectangle. Thus, the object of derived class can access the public members of Rectangle. In this program, when objects of class Area and Perimeter are created, constructor in base class is automatically called. If there was public member function in base class then, those functions also would have been accessible for objects a and p.

Keyword protected
In this program, length and breadth in the base class are protected data members. These data members are accessible from the derived class but, not accessible from outside it. This maintains the feature of
data hiding in C++ programming. If you defined length and breadth as private members then, those two data are not accessible to derived class and if defined as public members, it can be accessible from both derived class and from main() function.

<table>
<thead>
<tr>
<th>Accessibility</th>
<th>Private</th>
<th>Protected</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessible from own class ?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Accessible from derived class ?</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Accessible outside derived class ?</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Member Function Overriding in Inheritance
Suppose base class and derived class has member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class.

Virtual Function in C++ Programming
If there are member function with same name in derived classes, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as polymorphism which is one of the important features of OOP.

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable. This can be demonstrated by an example.

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

class B
{
public:
    void display()
    {
        cout<"Content of base class.\n";
    }
};

class D : public B
{
public:
    void display()
    {
        cout<"Content of derived class.\n";
    }
};

main()
{
    B *b;
    D d;
    b->display();
    b = &d;  /* Address of object d in pointer variable */
    b->display();
}
```

Note: An object (either normal or pointer) of derived class is type compatible with pointer to base class. So, b = &d; is allowed in above program.
Output:

Content of base class.
Content of base class.

In above program, even if the object of derived class d is put in pointer to base class, display( ) of the base class is executed( member function of the class that matches the type of pointer ).

Virtual Functions
If you want to execute the member function of derived class then, you can declare display( ) in the base class virtual which makes that function existing in appearance only but, you can’t call that function. In order to make a function virtual, you have to add keyword virtual in front of a function.

/* Example to demonstrate the working of virtual function in C++ programming. */

```
#include <iostream>
using namespace std;
class B {
    public:
        virtual void display() /* Virtual function */
        { cout<<"Content of base class.\n"; }
};
class D1 : public B {
    public:
        void display()
        { cout<<"Content of first derived class.\n"; }
};
class D2 : public B {
    public:
        void display()
        { cout<<"Content of second derived class.\n"; }
};

int main()
{
    B *b;
    D1 d1;
    D2 d2;
    /* b->display(); // You cannot use this code here because the function of base class is virtual. */
    b = &d1;
    b->display(); /* calls display() of class derived D1 */
    b = &d2;
    b->display(); /* calls display() of class derived D2 */
    return 0;
}
```

Output:

Content of first derived class.
Content of second derived class.
After the function of base class is made virtual, code b->display( ) will call the display( ) of the derived class depending upon the content of pointer.
In this program, display( ) function of two different classes are called with same code which is one of the example of polymorphism in C++ programming using virtual functions.

C++ Abstract class and Pure virtual Function
In C++ programming, sometimes inheritance is used only for the better visualization of data and you do not need to create any object of base class. For example: If you want to calculate area of different objects like: circle and square then, you can inherit these classes from a shape because it helps to visualize the problem but, you do not need to create any object of shape. In such case, you can declare shape as an abstract class. If you try to create object of an abstract class, compiler shows error.

Declaration of an Abstract Class
If expression =0 is added to a virtual function then, that function is becomes pure virtual function. Note that, adding =0 to virtual function does not assign value; it simply indicates the virtual function is a pure function. If a base class contains at least one virtual function then, that class is known as abstract class. Example to Demonstrate the Use of Abstract class

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

class Shape /* Abstract class */
{  protected:
    float l;

public:
    void get_data() /* Note: this function is not virtual. */
    {   cin>>l; }

    virtual float area() = 0; /* Pure virtual function */
};
class Square : public Shape
{  public:
    float area()
    {   return l*l; }
};
class Circle : public Shape
{   public:
    float area()
    {   return 3.14*l*l; }
};

main()
{   Square s;
    Circle c;
    cout<<"Enter length to calculate area of a square: ";
    s.get_data();
    cout<<"Area of square: "<<s.area();
    cout<<"\nEnter radius to calculate area of a circle: ";
```

c.get_data();
cout<<"Area of circle: "<<c.area();
}

In this program, pure virtual function virtual float area( ) = 0; is defined inside class Shape, so this class is an abstract class and you cannot create object of class Shape.

Summary
- A class is a specification or blueprint for a number of objects.
- Objects consist of both data and functions that operate on that data. In a class definition, the members—whether data or functions—can be private, meaning they can be accessed only by member functions of that class, or public, meaning they can be accessed by any function in the program.
- A member function is a function that is a member of a class. Member functions have access to an object’s private data, while non-member functions do not.
- A constructor is a member function, with the same name as its class that is executed every time an object of the class is created. A constructor has no return type but can take arguments. It is often used to give initial values to object data members. Constructors can be overloaded, so an object can be initialized in different ways.
- A destructor is a member function with the same name as its class but proceeded by a tilde (~). It is called when an object is destroyed. A destructor takes no arguments and has no return value.
- In the computer’s memory there is a separate copy of the data members for each object that is created from a class, but there is only one copy of a class’s member functions.
- You can restrict a data item to a single instance for all objects of a class by making it static.
- One reason to use OOP is the close correspondence between real-world objects and OOP classes. Deciding what objects and classes to use in a program can be complicated. For small programs, trial and error may be sufficient. For large programs, a more systematic approach is usually needed.

Polymorphism
Before getting into this section, it is recommended that you have a proper understanding of pointers and class inheritance. If any of the following statements seem strange to you, you should review the indicated sections:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Explained in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a::b(c) {};</td>
<td>Classes</td>
</tr>
<tr>
<td>a-&gt;b</td>
<td>Data Structures</td>
</tr>
<tr>
<td>class a: public b;</td>
<td>Friendship and inheritance</td>
</tr>
</tbody>
</table>

Pointers to base class
One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

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

// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
    protected:
        int width, height;
    public:
    void set_values (int a, int b) 
        { width=a; height=b; }
    
};

class CRectangle: public CPolygon {
    public:
        int area ()
            { return (width * height); }
    
};

class CTriangle: public CPolygon {
    public:
        int area ()
            { return (width * height / 2); }
    
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
}

Output:-
20
10

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations. The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:
Virtual members
A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

```cpp
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
    { width=a; height=b; }
    virtual int area ()
    { return (0); }
};

class CRectangle: public CPolygon {
public:
    int area ()
    { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area ()
    { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
}
```

Output:
20
10
0
Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0.

That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a polymorphic class.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.

**Abstract base classes**

Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all. This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```cpp
// abstract class CPolygon
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
        { width=a; height=b; }
        virtual int area () =0;
};
```

Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function.

This type of function is called a pure virtual function, and all classes that contain at least one pure virtual function are abstract base classes.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

```cpp
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:
CPolygon * ppoly1;
CPolygon * ppoly2;
would be perfectly valid.
This is so for as long as CPolygon includes a pure virtual function and therefore it’s an abstract base class.
However, pointers to this abstract base class can be used to point to objects of derived classes.
Here you have the complete example:

    // abstract base class
    #include <iostream>
    using namespace std;

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
            { width=a; height=b; }
        virtual int area (void) =0;
    }

class CRectangle: public CPolygon {
    public:
        int area (void)
            { return (width * height); }
    }

class CTriangle: public CPolygon {
    public:
        int area (void)
            { return (width * height / 2); }
    }

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
}

Output:-
20
10
If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

```
// pure virtual members can be called from the abstract base class
#include <iostream>
using namespace std;

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
    { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
    { cout << this->area() << endl; }
};

class CRectangle: public CPolygon {
public:
    int area (void)
    { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
    { return (width * height / 2); }
};

main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
}
```

Output:
20
10
Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let’s end with the same example again, but this time with objects that are dynamically allocated:

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

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
            { width=a; height=b; }
        virtual int area (void) =0;
        void printarea (void)
            { cout << this->area() << endl; }
};

class CRectangle: public CPolygon {
    public:
        int area (void)
            { return (width * height); }
};

class CTriangle: public CPolygon {
    public:
        int area (void)
            { return (width * height / 2); }
};

main () {
    CPolygon * ppoly1 = new CRectangle;
    CPolygon * ppoly2 = new CTriangle;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    delete ppoly1;
    delete ppoly2;
}

Output:
20
10
```
Notice that the ppoly pointers:
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.

**Introduction to Object-Oriented Analysis and Analysis**

The planning phase of any program’s development is extremely important. It is during this phase that the programmer carefully examines the program requirements and determines what the program must be able to do.

The process of object-oriented analysis includes the following steps:-

1. Identify the classes and objects to be used in the program.
2. Define the attributes for each class. A class’s attributes are the **data elements** used to describe an object instantiated from the class. They are all the values needed for the object to function properly in the program. A class’s attributes are its data members.
3. Define the behaviors for each class. The behaviors are the **activities** of each class must be capable of performing. A class’s behaviors are its **member functions**.
4. Define the relationships for each class. The possible relationships are access, ownership and inheritance.