Q1. FORMAT OF PROGRAME:

```
1  // conditional operator
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int a,b,c;
8
9    a=2;
10   b=7;
11   c = (a>b) ? a : b;
12
13   cout << c << '\n';
14 }
```

Q2. CLASS

Classes (I)

Classes are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using either keyword `class` or keyword `struct`, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

Classes have the same format as plain *data structures*, except that they can also include functions and have these new things called *access specifiers*. An *access specifier* is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights for the members that follow them:

- `private` members of a class are accessible only from within other members of the same class (or from their *"friends"*).
- `protected` members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
- Finally, `public` members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the `class` keyword have private access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically. For example:

```
1 class Rectangle {
2     int width, height;
3   public:
4     void set_values (int,int);
5     int area (void);
6 } rect;
```

Declares a class (i.e., a type) called `Rectangle` and an object (i.e., a variable) of this class, called `rect`. This class contains four members: two data members of type `int` (member `width` and member `height`) with *private access* (because private is the default access level) and two member functions with *public access*: the functions `set_values` and `area`, of which for now we have only included their declaration, but not their definition.

Notice the difference between the *class name* and the *object name*: In the previous example, `Rectangle` was the *class name* (i.e., the type), whereas `rect` was an object of type `Rectangle`. It is the same relationship `int` and `a` have in the following declaration:

```
  int a;
```

where `int` is the type name (the class) and `a` is the variable name (the object).

After the declarations of `Rectangle` and `rect`, any of the public members of object `rect` can be accessed as if they were normal functions or normal variables, by simply inserting a dot (`.`) between *object name* and *member name*. This follows the same syntax as accessing the members of plain data structures. For example:

```
1 rect.set_values (3,4);
2 myarea = rect.area();
```

The only members of `rect` that cannot be accessed from outside the class are `width` and `height`,

since they have private access and they can only be referred to from within other members of that same class.

Here is the complete example of class Rectangle:

```
1  // classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect;
19   rect.set_values (3,4);
20   cout << "area: " << rect.area();
21   return 0;
22 }
```

area: 12 Edit & Run

This example reintroduces the *scope operator* (`::`, two colons), seen in earlier chapters in relation to namespaces. Here it is used in the definition of function `set_values` to define a member of a class outside the class itself.

Notice that the definition of the member function `area` has been included directly within the definition of class `Rectangle` given its extreme simplicity. Conversely, `set_values` it is merely declared with its prototype within the class, but its definition is outside it. In this outside definition, the operator of scope (`::`) is used to specify that the function being defined is a member of the class `Rectangle` and not a regular non-member function.

The scope operator (`::`) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the function `set_values` in the previous example has access to the variables `width` and `height`, which are private members of class `Rectangle`, and thus only accessible from other members of the class, such as this.

The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that in the first case the function is automatically considered an *inline* member function by the compiler, while in the second it is

a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.

Members `width` and `height` have private access (remember that if nothing else is specified, all members of a class defined with keyword `class` have private access). By declaring them private, access from outside the class is not allowed. This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see how restricting access to these variables may be useful, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

The most important property of a class is that it is a type, and as such, we can declare multiple objects of it. For example, following with the previous example of class `Rectangle`, we could have declared the object `rectb` in addition to object `rect`:

```
1  // example: one class, two objects
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area () {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
14   height = y;
15 }
16
17 int main () {
18   Rectangle rect, rectb;
19   rect.set_values (3,4);
20   rectb.set_values (5,6);
21   cout << "rect area: " << rect.area() << endl;
22   cout << "rectb area: " << rectb.area() << endl;
23   return 0;
24 }
```

```
rect area: 12
rectb area: 30
```

Edit & Run

In this particular case, the class (type of the objects) is `Rectangle`, of which there are two instances (i.e., objects): `rect` and `rectb`. Each one of them has its own member variables and member functions.

Notice that the call to `rect.area()` does not give the same result as the call to `rectb.area()`. This is because each object of class `Rectangle` has its own variables `width` and `height`, as they -in some way- have also their own function members `set_value` and `area` that operate on the object's own

member variables.

Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called. Notice that no arguments were passed on the calls to `rect.area` or `rectb.area`. Those member functions directly used the data members of their respective objects `rect` and `rectb`.

Constructors

What would happen in the previous example if we called the member function `area` before having called `set_values`? An undetermined result, since the members `width` and `height` had never been assigned a value.

In order to avoid that, a class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even `void`.

The `Rectangle` class above can easily be improved by implementing a constructor:

```
1  // example: class constructor
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      Rectangle (int,int);
9      int area () {return (width*height);}
10 };
11
12 Rectangle::Rectangle (int a, int b) {
13   width = a;
14   height = b;
15 }
16
17 int main () {
18   Rectangle rect (3,4);
19   Rectangle rectb (5,6);
20   cout << "rect area: " << rect.area() << endl;
21   cout << "rectb area: " << rectb.area() << endl;
22   return 0;
23 }
```

```
rect area: 12
rectb area: 30
```

Edit & Run

The results of this example are identical to those of the previous example. But now, class `Rectangle` has no member function `set_values`, and has instead a constructor that performs a similar action: it initializes the values of `width` and `height` with the arguments passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
1 Rectangle rect (3,4);
2 Rectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.

Notice how neither the constructor prototype declaration (within the class) nor the latter constructor definition, have return values; not even `void`: Constructors never return values, they simply initialize the object.

Overloading constructors

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments:

```
1  // overloading class constructors
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      Rectangle ();
9      Rectangle (int,int);
10     int area (void) {return (width*height);}
11 };
12
13 Rectangle::Rectangle () {
14   width = 5;
15   height = 5;
16 }
17
18 Rectangle::Rectangle (int a, int b) {
19   width = a;
```

```
rect area: 12
rectb area: 25
```

Edit & Run

```
20   height = b;
21 }
22
23 int main () {
24   Rectangle rect (3,4);
25   Rectangle rectb;
26   cout << "rect area: " << rect.area() << endl;
27   cout << "rectb area: " << rectb.area() << endl;
28   return 0;
29 }
```

In the above example, two objects of class `Rectangle` are constructed: `rect` and `rectb`. `rect` is constructed with two arguments, like in the example before.

But this example also introduces a special kind constructor: the *default constructor*. The *default constructor* is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. In the example above, the *default constructor* is called for `rectb`. Note how `rectb` is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to call the default constructor:

```
1 Rectangle rectb;   // ok, default constructor called
2 Rectangle rectc(); // oops, default constructor NOT called
```

This is because the empty set of parentheses would make of `rectc` a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type `Rectangle`.

Uniform initialization

The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*. But constructors can also be called with other syntaxes:

First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):

```
class_name object_name = initialization_value;
```

More recently, C++ introduced the possibility of constructors to be called using *uniform initialization*, which essentially is the same as the functional form, but using braces (`{}`) instead of parentheses (`()`):

```
class_name object_name { value, value, value, ... }
```

Optionally, this last syntax can include an equal sign before the braces.

Here is an example with four ways to construct objects of a class whose constructor takes a single parameter:

```
   // classes and uniform initialization
1  #include <iostream>
2  using namespace std;
3
4  class Circle {
5      double radius;
6    public:
7      Circle(double r) { radius = r; }
8      double circum() {return
9  2*radius*3.14159265;}
10 };                                                  foo's circumference:    Edit & Run
11                                                    62.8319
12 int main () {
13   Circle foo (10.0);   // functional form
14   Circle bar = 20.0;   // assignment init.
15   Circle baz {30.0};   // uniform init.
16   Circle qux = {40.0}; // POD-like
17
18   cout << "foo's circumference: " <<
19 foo.circum() << '\n';
20   return 0;
   }
```

An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors:

```
1 Rectangle rectb;   // default constructor called
2 Rectangle rectc(); // function declaration (default constructor NOT called)
3 Rectangle rectd{}; // default constructor called
```

The choice of syntax to call constructors is largely a matter of style. Most existing code currently uses functional form, and some newer style guides suggest to choose uniform initialization over the others, even though it also has its potential pitfalls for its preference of `initializer_list` as its type.

Member initialization in constructors

When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a

colon (:) and a list of initializations for class members. For example, consider a class with the following declaration:

```
1 class Rectangle {
2     int width,height;
3   public:
4     Rectangle(int,int);
5     int area() {return width*height;}
6 };
```

The constructor for this class could be defined, as usual, as:

```
  Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

But it could also be defined using *member initialization* as:

```
  Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

Or even:

```
  Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body.

For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.

Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list. For example:

```
1 // member initialization
2 #include <iostream>
3 using namespace std;
4
5 class Circle {
6     double radius;
7   public:
8     Circle(double r) : radius(r) { }
```

```
foo's volume:
6283.19
```

Edit & Run

```
9      double area() {return
10 radius*radius*3.14159265;}
11 };
12
13 class Cylinder {
14     Circle base;
15     double height;
16   public:
17     Cylinder(double r, double h) : base (r),
18 height(h) {}
19     double volume() {return base.area() *
20 height;}
21 };
22
23 int main () {
24   Cylinder foo (10,20);
25
     cout << "foo's volume: " << foo.volume() <<
   '\n';
     return 0;
   }
```

In this example, class `Cylinder` has a member object whose type is another class (`base`'s type is `Circle`). Because objects of class `Circle` can only be constructed with a parameter, `Cylinder`'s constructor needs to call `base`'s constructor, and the only way to do this is in the *member initializer list*.

These initializations can also use uniform initializer syntax, using braces `{}` instead of parentheses `()`:

```
Cylinder::Cylinder (double r, double h) : base{r}, height{h} { }
```

Pointers to classes

Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

```
Rectangle * prect;
```

is a pointer to an object of class `Rectangle`.

Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the arrow operator (`->`). Here is an example with some possible combinations:

```
1 // pointer to classes example
2 #include <iostream>
3 using namespace std;
4
5 class Rectangle {
6   int width, height;
7 public:
8   Rectangle(int x, int y) : width(x), height(y) {}
9   int area(void) { return width * height; }
10 };
11
12
13 int main() {
14   Rectangle obj (3, 4);
15   Rectangle * foo, * bar, * baz;
16   foo = &obj;
17   bar = new Rectangle (5, 6);
18   baz = new Rectangle[2] { {2,5}, {3,6} };
19   cout << "obj's area: " << obj.area() << '\n';
20   cout << "*foo's area: " << foo->area() << '\n';
21   cout << "*bar's area: " << bar->area() << '\n';
22   cout << "baz[0]'s area:" << baz[0].area() << '\n';
23   cout << "baz[1]'s area:" << baz[1].area() << '\n';
24   delete bar;
25   delete[] baz;
26   return 0;
27 }
```

Edit & Run

This example makes use of several operators to operate on objects and pointers (operators `*`, `&`, `.`, `->`, `[]`). They can be interpreted as:

expression	can be read as
`*x`	pointed to by `x`
`&x`	address of `x`
`x.y`	member `y` of object `x`
`x->y`	member `y` of object pointed to by `x`
`(*x).y`	member `y` of object pointed to by `x` (equivalent to the previous one)
`x[0]`	first object pointed to by `x`
`x[1]`	second object pointed to by `x`
`x[n]`	(`n`+1)th object pointed to by `x`

Most of these expressions have been introduced in earlier chapters. Most notably, the chapter about arrays introduced the offset operator (`[]`) and the chapter about plain data structures introduced the arrow operator (`->`).

Classes defined with struct and union

Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

The keyword `struct`, generally used to declared plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword `class`. The only difference between both is that members of classes declared with the keyword `struct` have `public` access by default, while members of classes declared with the keyword `class` have `private` access by default. For all other purposes both keywords are equivalent in this context.

Conversely, the concept of *unions* is different from that of classes declared with `struct` and `class`, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is `public`.

Q2B **CLASS 2**

Classes (II)

Overloading operators

Classes, essentially, define new types to be used in C++ code. And types in C++ not only interact with code by means of constructions and assignments. They also interact by means of operators. For example, take the following operation on fundamental types:

```
1 int a, b, c;
2 a = b + c;
```

Here, different variables of a fundamental type (`int`) are applied the addition operator, and then the assignment operator. For a fundamental arithmetic type, the meaning of such operations is generally obvious and unambiguous, but it may not be so for certain class types. For example:

```
1 struct myclass {
2   string product;
3   float price;
4 } a, b, c;
5 a = b + c;
```

Here, it is not obvious what the result of the addition operation on `b` and `c` does. In fact, this code alone would cause a compilation error, since the type `myclass` has no defined behavior for additions. However, C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes. Here is a list of all the operators that can be overloaded:

Overloadable operators

```
+     -     *     /     =     <     >     +=    -=    *=    /=    <<    >>
<<=   >>=   ==    !=    <=    >=    ++    --    %     &     ^     !     |
~     &=    ^=    |=    &&    ||    %=    []    ()    ,     ->*   ->    new
delete      new[]       delete[]
```

Operators are overloaded by means of `operator` functions, which are regular functions with special names: their name begins by the `operator` keyword followed by the *operator sign* that is overloaded. The syntax is:

```
type operator sign (parameters) { /*... body ...*/ }
```

For example, *cartesian vectors* are sets of two coordinates: `x` and `y`. The addition operation of two *cartesian vectors* is defined as the addition both `x` coordinates together, and both `y` coordinates

together. For example, adding the *cartesian vectors* `(3,1)` and `(1,2)` together would result in `(3+1,1+2) = (4,3)`. This could be implemented in C++ with the following code:

```
1  // overloading operators example
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {};
9      CVector (int a,int b) : x(a), y(b) {}
10     CVector operator + (const CVector&);
11 };
12
13 CVector CVector::operator+ (const CVector& param) {
14   CVector temp;                                          4,3 Edit & Run
15   temp.x = x + param.x;
16   temp.y = y + param.y;
17   return temp;
18 }
19
20 int main () {
21   CVector foo (3,1);
22   CVector bar (1,2);
23   CVector result;
24   result = foo + bar;
25   cout << result.x << ',' << result.y << '\n';
26   return 0;
27 }
```

If confused about so many appearances of `CVector`, consider that some of them refer to the class name (i.e., the type) `CVector` and some others are functions with that name (i.e., constructors, which must have the same name as the class). For example:

```
1 CVector (int, int) : x(a), y(b) {}  // function name CVector (constructor)
2 CVector operator+ (const CVector&); // function that returns a CVector
```

The function `operator+` of class `CVector` overloads the addition operator (+) for that type. Once declared, this function can be called either implicitly using the operator, or explicitly using its functional name:

```
1 c = a + b;
2 c = a.operator+ (b);
```

Both expressions are equivalent.

In an earlier chapter, the *copy assignment* function was introduced as one of the special member functions that are implicitly defined, even when not explicitly declared in the class. The behavior of this function by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
1 CVector d (2,3);
2 CVector e;
3 e = d;          // copy assignment operator
```

The *copy assignment* member is the only operator implicitly defined for all classes. Of course, it can be redefined to any other functionality, such as, for example, to copy only certain members or perform additional initialization operations.

The operator overload functions are just regular functions which can have any behavior; there is actually no requirement that the operation performed by that overload bears a relation to the mathematical or usual meaning of the operator, although it is strongly recommended. For example, a class that overloads `operator+` to actually subtract or that overloads `operator==` to fill the object with zeros, is perfectly valid, although using such a class could be challenging.

The parameter expected for a member function overload for operations such as `operator+` is naturally the operand to the right hand side of the operator. This is common to all binary operators (those with an operand to its left and one operand to its right). But operators can come in diverse forms. Here you have a table with a summary of the parameters needed for each of the different operators than can be overloaded (please, replace `@` by the operator in each case):

Expression	Operator	Member function	Non-member function
`@a`	`+ - * & ! ~ ++ --`	`A::operator@()`	`operator@(A)`
`a@`	`++ --`	`A::operator@(int)`	`operator@(A,int)`
`a@b`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A,B)`
`a@b`	`= += -= *= /= %= ^= &= |= <<= >>= []`	`A::operator@(B)`	-
`a(b,c...)`	`()`	`A::operator()(B,C...)`	-
`a->b`	`->`	`A::operator->()`	-
`(TYPE) a`	`TYPE`	`A::operator TYPE()`	-

Where `a` is an object of class `A`, `b` is an object of class `B` and `c` is an object of class `C`. `TYPE` is just any type (that operators overloads the conversion to type `TYPE`).

Notice that some operators may be overloaded in two forms: either as a member function or as a non-member function: The first case has been used in the example above for `operator+`. But some operators can also be overloaded as non-member functions; In this case, the operator function takes an object of the proper class as first argument.

For example:

```
1  // non-member operator overloads
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {}
9      CVector (int a, int b) : x(a), y(b) {}
10 };
11
12
13 CVector operator+ (const CVector& lhs, const CVector& rhs) {
14   CVector temp;                                                  4,3 Edit & Run
15   temp.x = lhs.x + rhs.x;
16   temp.y = lhs.y + rhs.y;
17   return temp;
18 }
19
20 int main () {
21   CVector foo (3,1);
22   CVector bar (1,2);
23   CVector result;
24   result = foo + bar;
25   cout << result.x << ',' << result.y << '\n';
26   return 0;
27 }
```

The keyword this

The keyword `this` represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer to the object itself.

One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

```
1  // example on this                yes, &a is b
```

```
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6   public:
7     bool isitme (Dummy& param);
8 };
9
10 bool Dummy::isitme (Dummy& param)
11 {
12   if (&param == this) return true;
13   else return false;
14 }
15
16 int main () {
17   Dummy a;
18   Dummy* b = &a;
19   if ( b->isitme(a) )
20     cout << "yes, &a is b\n";
21   return 0;
22 }
```

It is also frequently used in `operator=` member functions that return objects by reference. Following with the examples on *cartesian vector* seen before, its `operator=` function could have been defined as:

```
1 CVector& CVector::operator= (const CVector& param)
2 {
3   x=param.x;
4   y=param.y;
5   return *this;
6 }
```

In fact, this function is very similar to the code that the compiler generates implicitly for this class for `operator=`.

Static members

A class can contain static members, either data or functions.

A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```
1 // static members in classes
2 #include <iostream>
3 using namespace std;
4
5 class Dummy {
6   public:
7     static int n;
8     Dummy () { n++; };
9     ~Dummy () { n--; };
10};
11                              7
12int Dummy::n=0;                6
13
14int main () {
15  Dummy a;
16  Dummy b[5];
17  Dummy * c = new Dummy;
18  cout << a.n << '\n';
19  delete c;
20  cout << Dummy::n << '\n';
21  return 0;
22}
```

In fact, static members have the same properties as non-member variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it. As in the previous example:

```
int Dummy::n=0;
```

Because it is a common variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
1 cout << a.n;
2 cout << Dummy::n;
```

These two calls above are referring to the same variable: the static variable `n` within class `Dummy` shared by all objects of this class.

Again, it is just like a non-member variable, but with a name that requires to be accessed like a member of a class (or an object).

Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, acting exactly as non-member functions but being accessed like members of the class. Because they are like non-member functions, they cannot access non-static

members of the class (neither member variables nor member functions). They neither can use the keyword `this`.

Const member functions

When an object of a class is qualified as a `const` object:

```
const MyClass myobject;
```

The access to its data members from outside the class is restricted to read-only, as if all its data members were `const` for those accessing them from outside the class. Note though, that the constructor is still called and is allowed to initialize and modify these data members:

```
1 // constructor on const object
2 #include <iostream>
3 using namespace std;
4
5 class MyClass {
6   public:
7     int x;
8     MyClass(int val) : x(val) {}
9     int get() {return x;}
10 };
11
12 int main() {
13   const MyClass foo(10);
14 // foo.x = 20;           // not valid: x cannot be modified
15   cout << foo.x << '\n';  // ok: data member x can be read
16   return 0;
17 }
```

The member functions of a `const` object can only be called if they are themselves specified as `const` members; in the example above, member `get` (which is not specified as `const`) cannot be called from `foo`. To specify that a member is a `const` member, the `const` keyword shall follow the function prototype, after the closing parenthesis for its parameters:

```
int get() const {return x;}
```

Note that `const` can be used to qualify the type returned by a member function. This `const` is not the same as the one which specifies a member as `const`. Both are independent and are located at different places in the function prototype:

```
int get() const {return x;}        // const member function
const int& get() {return x;}       // member function returning a const&
const int& get() const {return x;} // const member function returning a
const&
```

Member functions specified to be `const` cannot modify non-static data members nor call other non-`const` member functions. In essence, `const` members shall not modify the state of an object.

`const` objects are limited to access only members marked as `const`, but non-`const` objects are not restricted can access both `const` members and non-`const` members alike.

You may think that anyway you are seldom going to declare `const` objects, and thus marking all members that don't modify the object as const is not worth the effort, but const objects are actually very common. Most functions taking classes as parameters actually take them by `const` reference, and thus, these functions can only access their `const` members:

```
// const objects
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print (const MyClass& arg) {
  cout << arg.get() << '\n';
}

int main() {
  MyClass foo (10);
  print(foo);

  return 0;
}
```

10 Edit & Run

If in this example, `get` was not specified as a `const` member, the call to `arg.get()` in the `print` function would not be possible, because `const` objects only have access to `const` member functions.

Member functions can be overloaded on their constness: i.e., a class may have two member functions with identical signatures except that one is `const` and the other is not: in this case, the `const` version is called only when the object is itself const, and the non-`const` version is called when the object is itself non-`const`.

```
1 // overloading members on constness
2 #include <iostream>
3 using namespace std;
4
5 class MyClass {
6     int x;
7   public:
8     MyClass(int val) : x(val) {}
9     const int& get() const {return x;}
10    int& get() {return x;}
11 };                                                              15
12                                                                  20
13 int main() {
14   MyClass foo (10);
15   const MyClass bar (20);
16   foo.get() = 15;         // ok: get() returns int&
17 // bar.get() = 25;        // not valid: get() returns const int&
18   cout << foo.get() << '\n';
19   cout << bar.get() << '\n';
20
21   return 0;
22 }
```

Class templates

Just like we can create function templates, we can also create class templates, allowing classes to have members that use template parameters as types. For example:

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

This same class could also be used to create an object to store any other type, such as:

```
mypair<double> myfloats (3.0, 2.18);
```

The constructor is the only member function in the previous class template and it has been defined inline within the class definition itself. In case that a member function is defined outside the defintion of the class template, it shall be preceded with the `template <...>` prefix:

```
26 100
```

Notice the syntax of the definition of member function `getmax`:

```
1 template <class T>
2 T mypair<T>::getmax ()
```

Confused by so many `T`'s? There are three `T`'s in this declaration: The first one is the template parameter. The second `T` refers to the type returned by the function. And the third `T` (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

It is possible to define a different implementation for a template when a specific type is passed as template argument. This is called a *template specialization*.

For example, let's suppose that we have a very simple class called `mycontainer` that can store one element of any type and that has just one member function called `increase`, which increases its value. But we find that when it stores an element of type `char` it would be more convenient to have a completely different implementation with a function member `uppercase`, so we decide to declare a class template specialization for that type:

```
1
33 // 8
34    J
```

This is the syntax used for the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class name with `template<>` , including an empty parameter list. This is because all types are known and no template arguments are required for this specialization, but still, it is the specialization of a class template, and thus it requires to be noted as such.

But more important than this prefix, is the `<char>` specialization parameter after the class template

name. This specialization parameter itself identifies the type for which the template class is being specialized (`char`). Notice the differences between the generic class template and the specialization:

```
1 template <class T> class mycontainer { ... };
2 template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those identical to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Q3 MERITS & DEMERITS OF C

Inline function Advantages, Disadvantages, Performance and User Guidelines ?

Inline function is the optimization technique used by the compilers. One can simply prepend inline keyword to function prototype to make a function inline. Inline function instruct compiler to insert complete body of the function wherever that function got used in code.

Advantages :- 1) It does not require function calling overhead.
2) It also save overhead of variables push/pop on the stack, while function calling.
3) It also save overhead of return call from a function.
4) It increases locality of reference by utilizing instruction cache.
5) After in-lining compiler can also apply intraprocedural optmization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages :-
1) May increase function size so that it may not fit on the cache, causing lots of cahce miss.
2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if some body changes code inside inline function than all calling location will also be compiled.
4) If used in header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
6) Its not useful for embeded system where large binary size is not preferred at all due to memory size constraints.

Q4

advantage and disadvanges of c++

Advantages:

C++ is relatively-low level and is a systems programming language.
It has a large community.
It has a relatively clear and mature standard.

Disadvantages:

Unorthodox syntax is used in adoption of facilities like lamdas and templates (although still very usable).
It can be generally heavy if not careful.
C++ allows classes and thus functions with the same name (and overloaded functions) thus a symbol mangling system must be used. Can easily be wrapped in C functions though.

Q6

COMPARISON C / C++

When should we use C over C++ ?

C doesn't have as much overhead as C++, such as virtual tables, inheritance, and operator overloading. Since C cuts all of the latter out, there's less to go wrong and bumps up the efficiency which makes is some-what faster.

C++ is just as fast as C if you use the same features, so if you need the speed of C you have it without needing to use a C compiler.

C++ is no where near as portable as C code.

Advanced C++ features are at various levels of implementation in different compilers. This renders these advanced features non-portable.

One should remember C++ was written to solve specific problems with C. By throwing out C++ because some esoteric feature turns out not to be implemented in some vendor's implementation is no reason not to use the language.

Use C if you wish to write a program that is:
1. platform portable
2. programmer portable

1. C compilers exist for enormous number of platforms, while C++ compilers exist only for a limited number of some popular (mostly PC) platforms. For many exotic platforms (e.g. embedded or mainframes), if you are lucky, the only thing you get is some prehistoric build of GCC, that is far from standard compliant. So, probably you might be able to program C with classes, but not with STL and Boost. If you don't believe me, check Mozilla portable C++ programming guide - you will be shocked, which C++ constructs you are *not* allowed to use. Similar things at Google or IBM.

2. It is easy to find a C programmer, it is easy to understand C code. It is easy to maintain C code. You have a C project - just hire C programmers and you are done - they can immediately start hacking. You can learn C in a month or less. On the other hand, high level C++ programmers are incompatible with each other. They know (and use) different subsets of C++, and C++ is huge enough, that the number of useful subsets is enormous. Just ask a C++ programmer, whether constructors may throw exceptions or not, or whether you should use sharedptrs or not and you get a 40 post-long debate. They also tend to implement things in extremely compicated ways, just to get a virtual 5% performance improvement[1]. I pretty much understand why Linus Torvalds wanted to keep them away from Linux kernel.

There are boost libraries for C++ which does the same thing and even more, no need for C headers.

Q7B

One main difference between c and c++?

Best Answer

The main difference:
c is not object orientd where as c++ is object oriented.

10 major differences are:
1. C follows the procedural programming paradigm while C++ is a multi-paradigm language(procedural as well as object oriented)

In case of C, importance is given to the steps or procedure of the program while C++ focuses on the data rather than the process.
Also, it is easier to implement/edit the code in case of C++ for the same reason.

2. In case of C, the data is not secured while the data is secured(hidden) in C++

This difference is due to specific OOP features like Data Hiding which are not present in C.

3. C is a low-level language while C++ is a middle-level language

C is regarded as a low-level language(difficult interpretation & less user friendly) while C++ has features of both low-level(concentration on whats going on in the machine hardware) & high-level languages(concentration on the program itself) & hence is regarded as a middle-level language.

4. C uses the top-down approach while C++ uses the bottom-up approach

In case of C, the program is formulated step by step, each step is processed into detail while in C++, the base elements are first formulated which then are linked together to give rise to larger systems.

5. C is function-driven while C++ is object-driven

Functions are the building blocks of a C program while objects are building blocks of a C++ program.

6. C++ supports function overloading while C does not

Overloading means two functions having the same name in the same program. This can be done only in C++ with the help of Polymorphism(an OOP feature)

7. We can use functions inside structures in C++ but not in C.

In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

8. The NAMESPACE feature in C++ is absent in case of C

C++ uses NAMESPACE which avoid name collisions. For instance, two students enrolled in the same university cannot have the same roll number while two students in different universities might have the same roll number. The universities are two different namespace & hence contain the same roll number(identifier) but the same university(one namespace) cannot have two students with the same roll number(identifier)

9. The standard input & output functions differ in the two languages

C uses scanf & printf while C++ uses cin>> & cout<< as their respective input & output functions

10. C++ allows the use of reference variables while C does not

Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming.

C++, as the name suggests is a superset of C. As a matter of fact, C++ can run most of C code while C cannot run C++ code. Here are the 10 major differences between C++ & C...

Q7 23

History of C Language

The C programming language was designed by Dennies Ritchie in the early 1970s at Bell Laboratories. It was first used system implementation language for the nascent Unix operating system. The main reason to devised C was to overcome the limitations of B. It was Derived from the type-less language BCPL ((**Basic Combined Programming Language)**. C was was the evolution of B and BCPL by incorporating type checking. It was originally intended for use in writing compilers for other languages.

History of C++

C++ was devised by Bjarne Stroustrup in 1983 at Bell Laboratories. It is an extension of C by adding some enhancements to C language. Bjarne combined the simula's(a language designed for making simulations, created by Ole-Johan Dahl and Kristen Nygaard) features of object oriented and the efficiency of C. The new features added to language are templates, namespaces, exception handling and use of standary library.

Difference between c and c++

C++ is an extension of C language. This means that you can not only use the new features introduced with C++ but can also use the power and efficiency of C language. C and C++ are no more language for writing compilers and other languages, these general purpose languages are used worldwide in every field.
Here is a list of differences between c and c++.

The main difference between C and C++ is that C++ is object oriented while C is function or procedure oriented. Object oriented programming paradigm is focused on writing programs that are more readable and maintainable. It also helps the reuse of code by packaging a group of similar objects or using the concept of component programming model. It helps thinking in a logical way by using the concept of real world concepts of objects, inheritance and polymorphism. It should be noted that there are also some drawbacks of such features. For example using polymorphism in a program can slow down the performance of that program.

On the other hand, functional and procedural programming focus primarily on the actions and events, and the programming model focuses on the logical assertions that trigger execution of program code.

Q7 12

C vs. C++

This is an objective comparison of the applications, usage and language characteristics of **C** and **C++**. The origins and development trajectory of the two programming languages are also discussed.

Comparison chart

	C **User Rating (271):** • **3.99/5** • 1 • 2 • 3 • 4 • 5	C++ **User Rating (256):** • **4.07/5** • 1 • 2 • 3 • 4 • 5
Typing Discipline	Static, Weak	static, strong, unsafe, nominative
Paradigms	Imperative (procedural) systems implementation language	Multi-paradigm, object-oriented programming, generic programming, procedural programming, functional programming, metaprogramming
Designed by	Dennis Ritchie	Bjarne Stroustrup
Influenced	awk, csh, C++, C#, Objective-C, BitC, D, Concurrent C, Java, JavaScript, Limbo, Perl, PHP	Ada 95, C#, Java, PHP, D, Aikido
Influenced by	B (BCPL,CPL), ALGOL 68, Assembly	C, Simula, Ada 83, ALGOL 68, CLU, ML
Major Implementations	GCC, MSVC, Borland C, Watcom C	GNU Compiler Collection, Microsoft Visual C++, Borland C++ Builder, Intel C++ Compiler, LLVM/Clang
Appeared in	1972	1985
Garbage Collection	Manual; allows better management of memory.	No

	C	C++

	User Rating (271):
	• **3.99/5**
	  • 1
	  • 2
	  • 3
	  • 4
	  • 5
		User Rating (256):
		• **4.07/5**
		  • 1
		  • 2
		  • 3
		  • 4
		  • 5

Speed	C applications are faster to compile and execute than C++ applications	+-5% when compare with C if you know how to make a good use of C++.The performance of C++ and C are equal, since compilers are mature.
Usual filename extensions	.c	.cc, .cpp, .cxx, .h, .hh, .hpp
Programming-include	use #include<stdio.h>	include

Usage of C vs. C++

C proved very useful in running applications coded in assembly language because of its strengths like a simple compiler, lower access levels of memory, lower run time support and an efficient constructing language that was in sync with the hardware instructions. Another of its credits is that it is a highly portable (compatible with a variety of OS & Platforms) with very minimal source code changes required. Thus it has enabled remote operations & independence from the hardware. C is also compliant to a variety of standards, making it work with everything.

C++ is known as a mid-level language. Due to the fact that the C++ comprises of both high-level and low-level language features. Some of the adjectives used to describe C++ are static typed, free-form, multi-paradigm and supporting procedural programming.

Stroustrup, while programming for his Ph.D thesis, found that the Simula language had high level features helpful for large software development, but was too slow for practical use, while the BCPL (language) was fast, but too low-level and thus unsuitable for large software development. In Bell labs, he had to analyze the UNIX kernel with respect to distributed computing which created further problems and he set out to enhance C (due to its ultra portable nature) with features from the Simula. C++ was created in 1983 with additional features like virtual functions, function name and operator overloading, references, constants, user-controlled free-store memory, improved type checking and single-line comments with two forward slashes (//). The Cfront (commercial version) was released in 1985 with the class, derived class, strong type checking, inlining, and default argument features. 1985 also saw the release of the The C++ Programming Language, an important reference to the language in the absence of an official

standard. This was followed by the release of the C++ 2.0 in 1989 with features like multiple inheritance, abstract classes, static member functions, const member functions and protected members. Features like templates, exceptions, namespaces, new casts and Boolean type were added post 1990.

Along with the language, its library also evolved, with several additions like the stream I/O library, the Standard Template Library etc.

The first editions of the book K & R written by Dennis Ritchie & Brian Kernighan (original name: The C Programming Language) describes their version of C as the K & R C with full specifications, while the later editions include the ANSI (American National Standards Institute) C standards. Some of the salient features described are the introduction of various data types, removal of several semantic ambiguities, omission of other function declarations etc. Even after the introduction of the ANSI C, the K & R C continued to be the most portable programming language for programmers due to its wider compatibilities.

K&R function declarations did not include any information about function arguments leading to non-performance of function parameter type checks, although some compilers issued a warning message if a local function was called with the wrong number of arguments or if multiple calls to an external function used different numbers of arguments. Tools such as UNIX's lint utility were created for checking the consistency of functions used across multiple source files.

Language Characteristics

Characteristics of C

Some of the important characteristics of C are as follows:

1. Structured programming facilities
2. Confirming to the ALGOL traditions
3. Short circuit evaluation – usage of only one operand if the result can be determined with it alone
4. Static typing system for avoiding unintended operations
5. Value passed parameters with relevance to pointer value passing
6. Heterogeneous data combination & manipulation
7. Reserved keywords and free-format source text
8. Larger number of compound operators, such as +=, ++
9. Huge variable hiding capacity, though function definitions being non-nestable
10. Character – integer usage similar to assembly language
11. Low-level access to computer memory via machine addresses and typed pointers
12. Function pointers allow rudimentary forms of closures & polymorphic runtime
13. Pointer arithmetic defined Array indexing (secondary notion)
14. Standardized processor for defining macros, including source code files & conditional compilations
15. Complex Input/Output and mathematical functions with consistent delegation to library routines
16. Syntax same as "B" (C's predecessor) but different from ALGOL e.g.: { ... } replaced begin ... end, && and || replaced and & or, which
17. While B used & and | in both meanings, C made them syntactically distinct from the bit-wise operators
18. Similarities to Fortran e.g: the equal- sign for assignment (copying) & two consecutive equal-signs to test for equality (compare to EQ) or the equal-sign in BASIC)

Other unofficial features added with time were:

1. void functions
2. Functions returning struct or union types instead of pointers
3. Assignments enabled for struct data types
4. const qualifier to make an object read-only
5. Enumerated types
6. Creationg of tool to avoid the inherent problems of the language

Soon C became powerful enough to have the UNIX Kernel (written in a assembly language) re-written making it one of the first OS Kernels written in a language apart from the assembly languages.

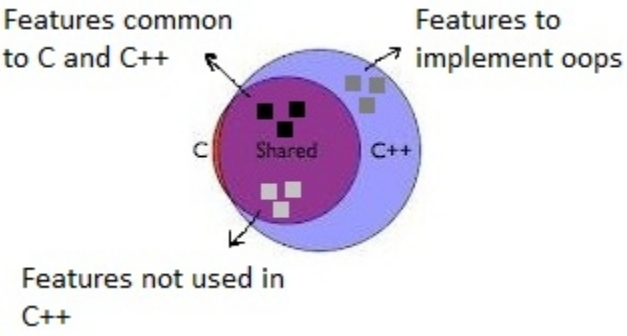
Characteristics of C++

1. C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
2. C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming)
3. C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
4. C++ is designed to be as compatible with C as possible, therefore providing a smooth transition from C
5. C++ avoids features that are platform specific or not general purpose
6. C++ does not incur overhead for features that are not used
7. C++ is designed to function without a sophisticated programming environment

Polymorphism, one of the prominent qualities of C++, enablesmany implementations with a single interphase and for objects to act according to circumstances. C++ supports both static (compile-time) and dynamic (run-time) polymorphisms. Compile-time polymorphism does not allow for certain run-time decisions, while run-time polymorphism typically incurs a performance penalty. C++, though considered a superset of C, there exist a few differences causing some valid C codes to be invalid in C++ or to behave differently in C++. Issues like the C++ defining new keywords namely new & class, that are used as identifiers in C. C and C++ codes can be intermixed by declaring any C code that is to be called from/used in C++ with C linkage & by placing it within an extern "C" { /* C code */ } block.

Q7 W

15 Most Important Differences Between C And C++



Basic Introduction:

- C++ is derived from C Language. It is a Superset of C.
- Earlier C++ was known as C with classes.
- In C++, the major change was the addition of classes and a mechanism for inheriting class objects into other classes.
- Most C Programs can be compiled in C++ compiler.
- C++ expressions are the same as C expressions.
- All C operators are valid in C++.

Following are the differences Between C and C++ :

C	C++
1. C is Procedural Language.	**1.** C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C	**2.** The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	**3.** The concept of polymorphism is used in C++. Polymorphism is the most Important Feature

	of OOPS.
4. Operator overloading is not possible in C.	**4.** Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	**5.** Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	**6.** Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	**7.** Multiple Declaration of global varioables are not allowed.
8. In C • scanf() Function used for Input. • printf() Function used for output.	**8.** In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	**9.** Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other Functions	**10.** In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	**11.** C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	**12.** Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	**13.**In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	**14.** It support both built-in and user define data types.

15. In C, Exception Handling is not present.	**15.** In C++, Exception Handling is done with Try and Catch block.

*mehul koli*27 April 2014 03:04

can any body tell me in which software c & c++ run or used or explaine....? plz tell me on this wall or mexxx1291@gmail.com

*Pramodh Kumar*31 May 2014 23:51

nice one and all the software professionals who are searching for s/w jobs now a days have to know must and differences between the c and c++ and java and .net languages

Q8

10 Major Differences Between C And C++

97950 Views August 14, 2009 343 Comments C++, Computing Rishabh Dev

C++, as the name suggests, is a superset of C. As a matter of fact, C++ can run most of C code while C cannot run C++ code. Here are the 10 major differences between C++ & C…

1. C follows the procedural programming paradigm while C++ is a multi-paradigm language(procedural as well as object oriented)

In case of C, importance is given to the steps or procedure of the program while C++ focuses on the data rather than the process.
Also, it is easier to implement/edit the code in case of C++ for the same reason.

2. In case of C, the data is not secured while the data is secured(hidden) in C++

This difference is due to specific OOP features like Data Hiding which are not present in C.

3. C is a low-level language while C++ is a middle-level language (Relatively, Please see the discussion at the end of the post)

C is regarded as a low-level language(difficult interpretation & less user friendly) while C++ has features of both low-level(concentration on whats going on in the machine hardware) & high-level languages(concentration on the program itself) & hence is regarded as a middle-level language.

Note: This is a relative difference. See updates at end of this post.

4. C uses the top-down approach while C++ uses the bottom-up approach

In case of C, the program is formulated step by step, each step is processed into detail while in C++, the base elements are first formulated which then are linked together to give rise to larger systems.

5. C is function-driven while C++ is object-driven

Functions are the building blocks of a C program while objects are building blocks of a C++ program.

6. C++ supports function overloading while C does not

Overloading means two functions having the same name in the same program. This can be done only in C++ with the help of Polymorphism(an OOP feature)

7. We can use functions inside structures in C++ but not in C.

In case of C++, functions can be used inside a structure while structures cannot contain functions in C.

8. The NAMESPACE feature in C++ is absent in case of C

C++ uses NAMESPACE which avoid name collisions. For instance, two students enrolled in the same university cannot have the same roll number while two students in different universities might have the same roll number. The universities are two different namespace & hence contain the same roll number(identifier) but the same university(one namespace) cannot have two students with the same roll number(identifier)

9. The standard input & output functions differ in the two languages

C uses scanf & printf while C++ uses cin>> & cout<< as their respective input & output functions

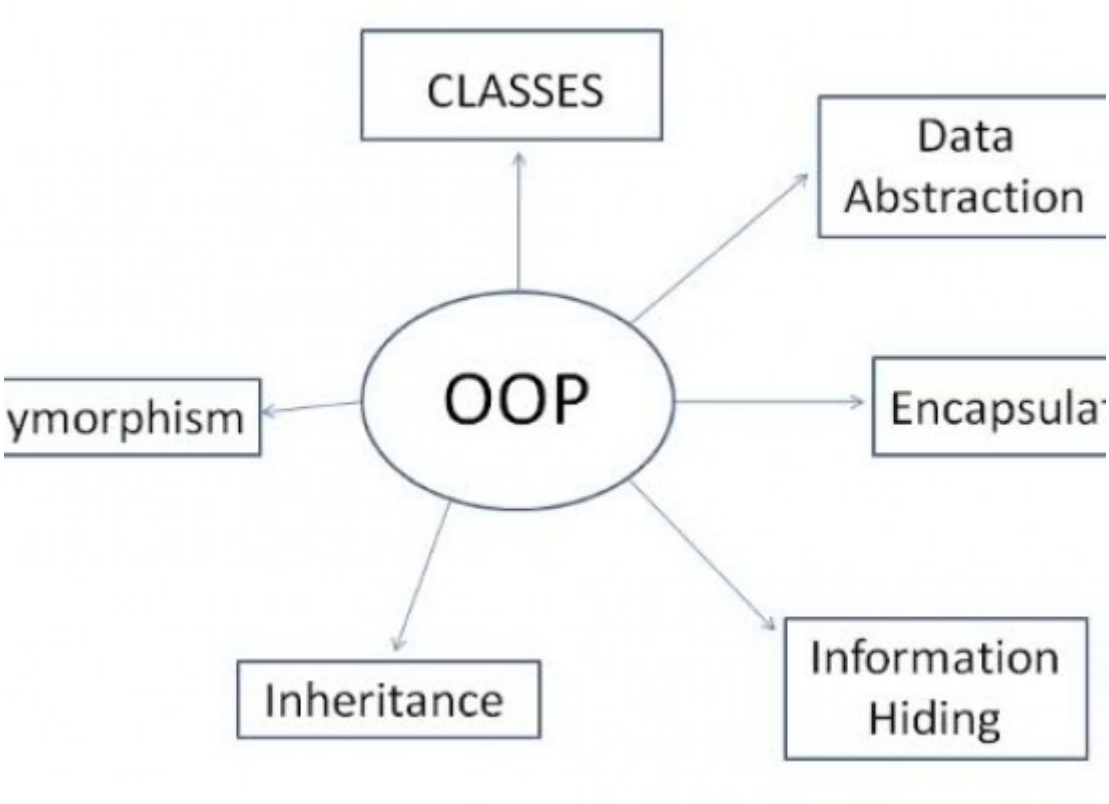
10. C++ allows the use of reference variables while C does not

Reference variables allow two variable names to point to the same memory location. We cannot use these variables in C programming.

About author

Rishabh Dev

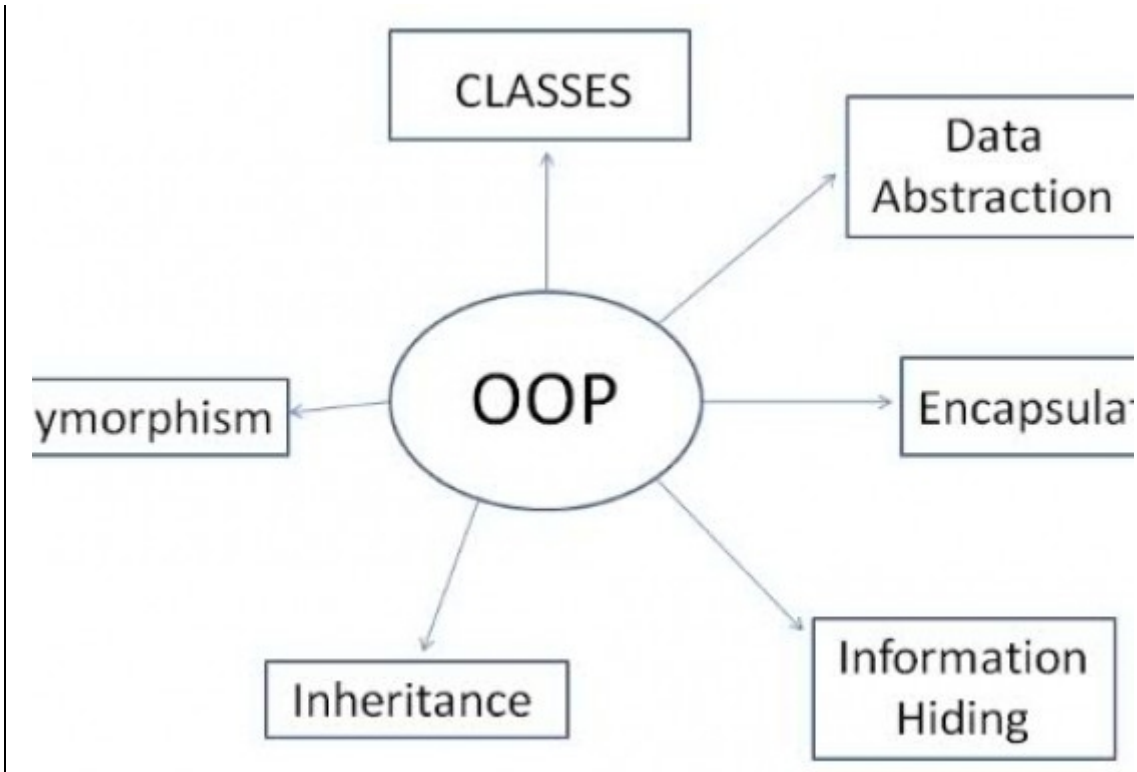
Rishabh Dev is the Founder, Author & Editor at Durofy.com. Start a conversation with Rishabh at dev@durofy.com to know more.



August 11, 2009

The Basics Of Object Oriented Programming

C


CLASSES
Data Abstraction
OOP
ymorphism
Encapsula
Inheritance
Information Hiding

Q 8 w

Advantages and Disadvantages of Object-Oriented Approach

Oracle Tips by Burleson Consulting

Benefits of Object-Oriented Approach

Object-oriented databases make the promise of reduced maintenance, code reusability, real world modeling, and improved reliability and flexibility. However, these are just promises and in the real world some users find that the object-oriented benefits are not as compelling as they originally believed. For example, what is code reusability? Some will say that they can reuse much of the object-oriented code that is created for a system, but many say there is no more code reusability in object-oriented systems than in traditional systems. Code reusability is a subjective thing, and depends heavily on how the system is defined. The object-oriented approach does give the ability to reduce some of the major expenses associated with systems, such as maintenance and development of programming code. Here are some of the benefits of the object-oriented approach:

Reduced Maintenance: The primary goal of object-oriented development is the assurance that the system will enjoy a longer life while having far smaller maintenance costs. Because most of the processes within the system are encapsulated, the behaviors may be reused and incorporated into new behaviors.

Real-World Modeling: Object-oriented system tend to model the real world in a more complete fashion than do traditional methods. Objects are organized into classes of objects, and objects are associated with behaviors. The model is based on objects, rather than on data and processing.

Improved Reliability and Flexibility: Object-oriented system promise to be far more reliable than traditional systems, primarily because new behaviors can be "built" from existing objects. Because objects can be dynamically called and accessed, new objects may be created at any time. The new objects may inherit data attributes from one, or many other objects. Behaviors may be inherited from super-classes, and novel behaviors may be added without effecting existing systems functions.

High Code Reusability: When a new object is created, it will automatically inherit the data attributes and characteristics of the class from which it was spawned. The new object will also inherit the data and behaviors from all superclasses in which it participates. When a user creates a new type of a widget, the new object behaves "wigitty", while having new behaviors which are defined to the system.

The downside of the Object Technology

There are several major misconceptions which must be addressed when considering the use of an object-oriented method:

Object-oriented Development is not a panacea - Object-oriented Development is best suited for dynamic, interactive environments, as evidenced by its widespread acceptance in CAD/CAM and engineering design systems. Wide-scale object-oriented corporate systems are still unproved, and many bread-and-butter information systems applications (i.e. payroll, accounting), may not benefit from the object-oriented approach.

Object-oriented Development is not a technology - Although many advocates are religious in their fervor for object-oriented systems, remember that all the "HOOPLA" is directed at the object-oriented approach to problem solving, and not to any specific technology.

Object-oriented Development is not yet completely accepted by major vendors - Object-oriented Development has gained some market respectability, and vendors have gone from catering to a "lunatic fringe" to a respected market. Still, there are major reservations as to whether Object-oriented development will become a major force, or fade into history, as in the 1980's when Decision Support Systems made great promises, only to fade into obscurity.

Cannot find qualified programmers and DBA?s

When one investigates the general acceptance of object-oriented systems in the commercial marketplace, you generally find that most managers would like to see an object technology approach, but they do not have the time to train their staffs in object-oriented methods. Other will say that the object-oriented method is only for graphical workstation systems, and that there is no pressing need for object-oriented system within mainstream business systems.

Even though commercial object-oriented programming languages have been on the market for several years, systems written with object-oriented languages comprise less than 1% of systems today.

Once a major vendor begins conforming to a standard, it can become impossible to retrofit their standard to conform to another standard. When the American Standards Committee came out with a standard character set for computers (ASCII), IBM disregarded the standard and proceeded with their own character set, called the Extended Binary Character Data Interchange Code (EBCDIC). Even thirty years later, there has still been no resolution between ASCII and EBCDIC, and data transfers between ASCII and EBCDIC machines continue to present problems. For example, the EBCDIC character set has no characters for "[" and "]", and ASCII has no character for the "cent" sign.

Summary

When one strips away all of the confusing acronyms and jargon, the object technology approach is nothing more than a method, an approach to systems design which can be implemented without any changes to existing software technology.

Here is an actual example from the popular IDMS database:

Q8 g

Advantages and Disadvantages of OOP
by pooryorick ▲

Advantages of OOP
Object-Oriented Programming has the following advantages over conventional approaches:

- OOP provides a clear modular structure for programs which makes it good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer. This is particularly useful for developing graphical user interfaces.

Concepts of OOP:

- Objects
- Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism

Objects
Objects are the basic run-time entities in an object-oriented system. Programming problem is analyzed in terms of objects and nature of communication between them. When a program is executed, objects interact with each other by sending messages. Different objects can also interact with each other without knowing the details of their data or code.
Classes
A class is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class.
Data Abstraction and Encapsulation
Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes. Storing data and functions in a single unit (class) is encapsulation. Data cannot be accessible to the outside world and only those functions which are stored in the class can access it.
Inheritance
Inheritance is the process by which objects can acquire the properties of objects of other class. In OOP, inheritance provides reusability, like, adding additional features to an existing class without modifying it. This is achieved by deriving a new class from the existing one. The new class will have combined features of both the classes.
Polymorphism

Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation. Polymorphism is extensively used in implementing Inheritance.

Q8 ss

Object Oriented Programming: Advantages of OOP

Object Oriented Programming has great advantages over other programming styles:

- ***Code Reuse and Recycling***: Objects created for Object Oriented Programs can easily be reused in other programs.
- ***Encapsulation (part 1)***: Once an Object is created, knowledge of its implementation is not necessary for its use. In older programs, coders needed understand the details of a piece of code before using it (in this or another program).
- ***Encapsulation (part 2)***: Objects have the ability to hide certain parts of themselves from programmers. This prevents programmers from tampering with values they shouldn't. Additionally, the object controls how one interacts with it, preventing other kinds of errors. For example, a programmer (or another program) cannot set the width of a window to -400.
- ***Design Benefits***: Large programs are very difficult to write. Object Oriented Programs force designers to go through an extensive planning phase, which makes for better designs with less flaws. In addition, once a program reaches a certain size, Object Oriented Programs are actually *easier* to program than non-Object Oriented ones.
- ***Software Maintenance:*** Programs are not disposable. Legacy code must be dealt with on a daily basis, either to be improved upon (for a new version of an exist piece of software) or made to work with newer computers and software. An Object Oriented Program is much easier to modify and maintain than a non-Object Oriented Program. So although a lot of work is spent before the program is written, less work is needed to maintain it over time.

Q 8 dd

Object-Oriented Programming

Computer programs become ever more **complex**. Hello World in C# involves millions of lines of code. Developers need to reuse old code. They need ways to add new features without changing other code.

Intro

Most examples show Car objects and Employee classes. But what are the real benefits of object-oriented design, and why is it important? OOP has very little to do with Car objects or Employee classes.

OOP: It is normally called object-oriented programming, but it is also component-oriented programming.

Tip: Good OOP is bound to design patterns. These patterns are basically the blueprints of quality object design.

Class, Design Patterns

Syntax

Objects provide syntax clues. To make an object, you use new. Objects provide accessor methods that manage access to their internals. With native OOP languages, the compiler and build environments are designed for this.

NewConstructor

Sharing code. In a new project, you have three additional requirements. In old languages, you would need change the existing code. With C# and OOP, you can simply inherit the old code into a new class.

Inheritance

Tip: It improves reusability.
You haven't changed any of the old code.
You haven't introduced any new bugs in old code.

Old callers don't need changes. Your old, reliable code doesn't need to be cloned just to add some functionality to it. The existing callers can still function as is. No changes were made to them or the libraries they use.

So: It causes less breakage. In older languages, you can add functionality, but this breaks other parts of your project.

However: With OOP, changing the implementation of one part is safer, because it won't affect the way the code is called.

Errors

Exceptions improve error handling. These allow us to separate errors from what the main intent of the program is. When you write code to list objects in an array, checking for null or invalid numbers may not be its primary purpose.

So: With exception handling, we can separate the error handling from the really important thrust of the program.

Exception

Development

OOP can both make development faster, and your programs run faster. We are living in a world in which every person is rushed. As developers we are always learning and trying new things. OOP provides separation of various objects.

Tip: You can use the proxy design pattern to meter access to expensive parts of your code. Just call into the proxy to access a resource.

Protection Proxy

It allows single-instance code. In old languages, you can use global variables and initialize them when you need them. But the singleton design pattern offers a way to simplify the code that does that.

It has better IDEs. Object orientation can provide many clues to your development environment. A program like Visual Studio can make smarter guesses about what you intended to do, and what you may want to do next.

Visual Studio

So: It speeds development and execution. Almost always, these two are dependent on one another.

And: With a language like C#, we can spend more time on designing a better algorithm, rather than writing lower-level code.

Interfaces are a concession to the limitations of programmers. We can't remember every detail of how to use other developers' code. They provide organization and sanity to large projects. Interfaces help create a blueprint of a module.

Then: Others can help build it to specification. Interfaces don't provide anything beyond syntax or organization.

Practical

OOP is practical. It is really not a new paradigm or a revolutionary new thing. Everything on your brand new processor is still running machine code. Above that there is still assembly language.

Also: Object-orientation provides us with more organization and simpler syntax. Every part of this design is about practicality.

Special

C# is special because it eliminates the hard stuff. It has a compiler that basically rejects any code that looks like it could be buggy, whether it is or not. It takes several approaches to prevent programmers from making mistakes.

And: Code quality is improved. The C# language enforces that every method is in a class, which results in better organization.

Thus: It is a practical language based on organization and helping programmers write the code that they intend to write.

Criticism

Linus Torvalds wrote the original code for Linux. He is outspoken about C++ and newer object-oriented languages. However, the software industry that gets most stuff done is totally committed to OOP.

The difference between the two viewpoints is that objects are useful for getting programs written and working well fast. Linus didn't have a deadline to ship his operating system, but almost every commercial project does.

We are not perfect. We make mistakes. Almost every program has flaws. But we are people who have to work and support our families and pay rent. Object-orientation is a compromise between exacting design and practicality.

Therefore: If you are using a C# program that is slow, it is almost positively not the fault of the C# language.

Note: New languages like Java and C# are extremely fast and often faster than their C equivalents.

And: When you take a talented developer, he or she is free to focus on more important algorithmic problems.

Summary

If you want to program embedded systems or OS kernels, you probably won't want to work with C# or even OOP in general. For the rest of us, 90% of the software community, OOP is a practical way to write programs.

So:OOP enables those of us who are not computer science geniuses to write programs that are good and fill the needs of the market.