# Association, Aggregation, Composition, Abstraction, Generalization, Realization, Dependency

These terms signify the relationships between classes. These are the building blocks of object oriented programming and very basic stuff. But still for some, these terms look like Latin and Greek. Just wanted to refresh these terms and explain in simpler terms.

## **Association**

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

http://javapapers.com/wp-content/uploads/2010/06/association.jpg

**Example:**A Student and a Faculty are having an association.

## **Aggregation**

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

## **Composition**

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

**Example:**A class contains students. A student cannot exist without a class. There exists composition between class and students.

### Difference between aggregation and composition

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

***Example:***A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

## **Abstraction**

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

***Example:***A wire frame model of a car.

## **Generalization**

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/generalization.jpg

***Example:*** Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

## **Realization**

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

http://javapapers.com/wp-content/uploads/2010/06/realization.jpg

***Example:*** A particular model of a car ‘GTB Fiorano’ that implements the blueprint of a car realizes the abstraction.

## **Dependency**

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

http://javapapers.com/wp-content/uploads/2010/06/dependency.jpg

***Example:***Relationship between shape and circle is dependency.

----------------------------------------------------------------

# Template Specialization in C++

[Template](http://en.wikipedia.org/wiki/Template_(C++))is a great feature in C++. We write code once and use it for any data type including user defined data types. For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.  
What if we want a different code for a particular data type? Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?  
It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

|  |
| --- |
| // A generic sort function  template <class T>  void sort(T arr[], int size)  {      // code to implement Quick Sort  }    // Template Specialization: A function specialized for char data type  template <>  void sort<char>(char arr[], int size)  {      // code to implement counting sort  } |

Another example could be a class *Set*that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

***An Example Program for function template specialization***  
For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

|  |
| --- |
| #include <iostream>  using namespace std;    template <class T>  void fun(T a)  {     cout << "The main template fun(): " << a << endl;  }    template<>  void fun(int a)  {      cout << "Specialized Template for int type: " << a << endl;  }    int main()  {      fun<char>('a');      fun<int>(10);      fun<float>(10.14);  } |

Output:

The main template fun(): a

Specialized Template for int type: 10

The main template fun(): 10.14

***An Example Program for class template specialization***  
In the following program, a specialized version of class Test is written for int data type.

|  |
| --- |
| #include <iostream>  using namespace std;    template <class T> //Normal template  class Test  {    // Data memnbers of test  public:     Test()     {         // Initializstion of data memnbers         cout << "General template object \n";     }     // Other methods of Test  };    template <> //Note template specialization syntax here it blanks and in class they put <int>  class Test <int>  {  public:     Test()     {         // Initializstion of data memnbers         cout << "Specialized template object\n";     }  };    int main()  {      Test<int> a;      Test<char> b;      Test<float> c;      return 0;  } |

Output:

Specialized template object

General template object

General template object

***How does template specialization work?***  
When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).  
If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Template Metaprogramming in C++

#include <iostream>

using namespace std;

template<int n> struct funStruct

{

    enum { val = 2\*funStruct<n-1>::val };

};

template<> struct funStruct<0>

{

    enum { val = 1 };

};

int main()

{

    cout << funStruct<8>::val << endl;

    return 0;

}

256

The program calculates “2 raise to the power 8 (or 2^8)”. In fact, the structure *funStruct*can be used to calculate 2^n for any known n (or constant n). The special thing about above program is: **calculation is done at compile time**. So, it is compiler that calculates 2^8. To understand how compiler does this, let us consider the following facts about templates and enums:

1) We can pass nontype parameters (parameters that are not data types) to class/function templates.  
2) Like other const expressions, values of enumaration constants are evaluated at compile time.  
3) When compiler sees a new argument to a template, compiler creates a new instance of the template.

Let us take a closer look at the original program. When compiler sees *funStruct<8>::val*, it tries to create an instance of *funStruct*with parameter as 8, it turns out that *funStruct<7>* must also be created as enumaration constant *val* must be evaluated at compile time. For *funStruct<7>*, compiler need *funStruct<6>* and so on. Finally, compiler uses *funStruct<1>::val* and compile time recursion terminates. So, using templates, we can write programs that do computation at compile time, such programs are called [template metaprograms](http://en.wikipedia.org/wiki/Template_metaprogramming). Template metaprogramming is in fact [Turing-complete](http://en.wikipedia.org/wiki/Turing_completeness), meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram. Template Metaprogramming is generally not used in practical programs, it is an interesting conecpt though.

# Understanding constexper specifier

constexpr is a feature added in C++ 11. The main idea is performance improvement of programs by doing computations at compile time rather than run time. Note that once a program is compiled and finalized be developer, it is run multiple times by users. The idea is to spend time in compilation and save time at run time

constexpr specifies that the value of an object or a function can be evaluated at compile time and the expression can be used in other constant expressions. For example, in below code product() is evaluated at compile time.

|  |
| --- |
| // constexpr function for product of two numbers.  // By specifying constexpr, we suggest compiler to  // to evaluate value at compiler time  constexpr int product(int x, int y)  {      return (x \* y);  }    int main()  {      const int x = product(10, 20);      cout << x;      return 0;  } |

constexpr specifies that the value of an object or a function can be evaluated at compile time and the expression can be used in other constant expressions. For example, in below code product() is evaluated at compile time.

|  |
| --- |
| // constexpr function for product of two numbers.  // By specifying constexpr, we suggest compiler to  // to evaluate value at compiler time  constexpr int product(int x, int y)  {      return (x \* y);  }    int main()  {      const int x = product(10, 20);      cout << x;      return 0;  } |

They serve different purposes. constexpr is mainly for optimization while const is for practically const objects like value of Pi.  
Both of them can be applied to member methods. Member methods are made const to make sure that there are no accidental changes by the method. On the other hand, the idea of using constexpr is to compute expressions at compile time so that time can be saved when code is run.  
const can be only be used with non-static member function whereas constexpr can be used with with member and non-member functions, even with constructors but with condition that argument and return type must be of literal types.

# Namespace in C++

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

* Namespace is a feature added in C++ and not present in C.
* A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
* Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

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

namespace namespace\_name

{

int x, y; // code declarations where

// x and y are declared in

// namespace\_name's scope

}

* Namespace declarations appear only at global scope.
* Namespace declarations can be nested within another namespace.
* Namespace declarations don’t have access specifiers. (Public or private)
* No need to give semicolon after the closing brace of definition of namespace.
* We can split the definition of namespace over several units.

# Some interesting facts about static member functions in C++

1. static member functions do not have [this pointer](http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr035.htm).
2. A static member function cannot be virtual
3. Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

#include<iostream>

class Test {

    static void fun() {}

    void fun() {} // compiler error

};

int main()

{

    getchar();

     return 0;

}

1. A static member function can not be declared const, volatile, or const volatile.

# When do we use Initializer List in C++?

* 1. **For initialization of non-static const data members:**
  2. **For initialization of reference members:**
  3. **For initialization of member objects which do not have default constructor**
  4. **For initialization of base class members**
  5. **When constructor’s parameter name is same as data member**
  6. **For Performance reasons:**

# When is copy constructor called?

* When an object of the class is returned by value.
* When an object of the class is passed (to a function) by value as an argument.
* When an object is constructed based on another object of the same class.
* When compiler generates a temporary object.

**References vs Pointers**

References are less powerful than pointers  
1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.  
2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.  
3) A reference must be initialized when declared. There is no such restriction with pointers

References are safer and easier to use:  
1) Safer: Since references must be initialized, wild references like [wild pointers](http://www.geeksforgeeks.org/archives/4979) are unlikely to exist. It is still possible to have references that don’t refer to a valid location (See questions 5 and 6 in the below exercise )  
2) Easier to use: References don’t need dereferencing operator to access the value. They can be used like normal variables. ‘&’ operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed to access members.

# Virtual Functions

# A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as [virtual](https://msdn.microsoft.com/en-us/library/by37c6et.aspx) and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the PrintBalance function and two derived classes

## Choosing composition vs. inheritance

Both composition and inheritance place subobjects inside your new class. Both use the constructor initializer list to [construct](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter14_012.html) these subobjects. You may now be wondering what the difference is between the two, and when to choose one over the other.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you [embed](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter14_012.html) an object to implement features of your new class, but the user of your new class sees the interface you’ve defined rather than the interface from the original class. To do this, you follow the typical path of embedding **private** objects of existing classes inside your new class.

Occasionally, however, it makes sense to allow the class user to directly access the composition of your new class, that is, to make the member objects **public**. The member objects use [access control](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter14_012.html) themselves, so this is a safe thing to do and when the user knows you’re assembling a bunch of parts, it makes the interface easier to understand. A **Car** class is a good example:

//: C14:Car.cpp

// Public composition

class Engine {

public:

void start() const {}

void rev() const {}

void stop() const {}

};

class Wheel {

public:

void inflate(int psi) const {}

};

class Window {

public:

void rollup() const {}

void rolldown() const {}

};

class Door {

public:

Window window;

void open() const {}

void close() const {}

};

class Car {

public:

Engine engine;

Wheel wheel[4];

Door left, right; // 2-door

};

int main() {

Car car;

car.left.window.rollup();

car.wheel[0].inflate(72);

} ///:~

Because the composition of a **Car** is part of the analysis of the problem (and not simply part of the underlying design), making the members **public** assists the client programmer’s understanding of how to use the class and requires less code complexity for the creator of the class.

With a little thought, you’ll also see that it would make no sense to compose a **Car** using a “vehicle” object – a car doesn’t contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

### Object slicing

There is a distinct difference between passing the addresses of objects and passing objects by value when using polymorphism. All the examples you’ve seen here, and virtually all the examples you should see, pass addresses and not values. This is because addresses all have the same size[[58]](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_030.html), so passing the address of an object of a derived type (which is usually a bigger object) is the same as passing the address of an object of the base type (which is usually a smaller object). As explained before, this is the goal when using polymorphism – code that manipulates a base type can transparently manipulate derived-type objects as well.

If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast. In the following example you can see what happens when an object is sliced:

//: C15:ObjectSlicing.cpp

#include <iostream>

#include <string>

using namespace std;

class Pet {

string pname;

public:

Pet(const string& name) : pname(name) {}

virtual string name() const { return pname; }

virtual string description() const {

return "This is " + pname;

}

};

class Dog : public Pet {

string favoriteActivity;

public:

Dog(const string& name, const string& activity)

: Pet(name), favoriteActivity(activity) {}

string description() const {

return Pet::name() + " likes to " +

favoriteActivity;

}

};

void describe(Pet p) { // Slices the object

cout << p.description() << endl;

}

int main() {

Pet p("Alfred");

Dog d("Fluffy", "sleep");

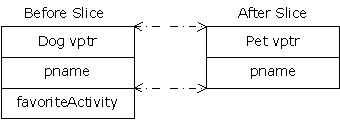
describe(p);

describe(d);

} ///:~

The function **describe( )** is passed an object of type **Pet** *by value*. It then calls the virtual function **description( )** for the **Pet** object. In **main( )**, you might expect the first call to produce “This is Alfred,” and the second to produce “Fluffy likes to sleep.” In fact, both calls use the base-class version of **description( )**.

Two things are happening in this program. First, because **describe( )**accepts a **Pet** *object* (rather than a pointer or reference), any calls to **describe( )** will cause an object the size of **Pet** to be pushed on the stack and cleaned up after the call. This means that if an object of a class inherited from **Pet** is passed to **describe( )**, the [compiler](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_017.html) accepts it, but it copies only the **Pet** portion of the object. It *slices* the derived portion off of the object, like this:



Now you may wonder about the virtual [function call](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_017.html). **Dog::description( )**makes use of portions of both **Pet** (which still exists) and **Dog**, which no longer exists because it was sliced off! So what happens when the virtual function is called?

You’re saved from disaster because the object is being passed by value. Because of this, the compiler knows the precise type of the object because the derived object has been forced to become a base object. When passing by value, the copy-constructor for a **Pet**object is used, which initializes the VPTR to the **Pet** VTABLE and copies only the **Pet**parts of the object. There’s no explicit copy-constructor here, so the compiler synthesizes one. Under all interpretations, the object truly becomes a **Pet** during slicing.

Object slicing actually removes part of the existing object as it copies it into the [new object](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_017.html), rather than simply changing the meaning of an address as when using a pointer or reference. Because of this, upcasting into an object is not done often; in fact, it’s usually something to watch out for and prevent. Note that, in this example, if **description( )**were made into a pure virtual function in the base class (which is not unreasonable, since it doesn’t really do anything in the [base class](http://www.linuxtopia.org/online_books/programming_books/thinking_in_c++/Chapter15_017.html)), then the compiler would prevent object slicing because that wouldn’t allow you to “create” an object of the base type (which is what happens when you upcast by value). This could be the most important value of pure virtual functions: to prevent object slicing by generating a compile-time error message if someone tries to do it.

--------------------------------------------------------------------------------------------------------------------------------------

# Function Overloading in C++

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

|  |
| --- |
| #include <iostream>  using namespace std;    void print(int i) {    cout << " Here is int " << i << endl;  }  void print(double  f) {    cout << " Here is float " << f << endl;  }  void print(char\* c) {    cout << " Here is char\* " << c << endl;  }    int main() {    print(10);    print(10.10);    print("ten");    return 0;  } |

Run on IDE

Output:

Here is int 10

Here is float 10.1

Here is char\* ten

# ----------------------------------------------------------------

# Law of Three

The "law of three" is not really a law, but rather a guideline: if a class needs an explicitly declared copy constructor, copy assignment operator, or destructor, then it usually needs all three.

There are exceptions to this rule (or, to look at it another way, refinements). For example, sometimes a destructor is explicitly declared just in order to make it virtual; in that case there's not necessarily a need to declare or implement the copy constructor and copy assignment operator.

Most classes should not declare any of the "big three" operations; classes that manage resources generally need all three.

## **Rule of Three[[edit](https://en.wikipedia.org/w/index.php?title=Rule_of_three_(C%2B%2B_programming)&action=edit&section=1" \o "Edit section: Rule of Three)]**

The **rule of three** (also known as the Law of The Big Three or The Big Three) is a [rule of thumb](https://en.wikipedia.org/wiki/Rule_of_thumb) in [C++](https://en.wikipedia.org/wiki/C%2B%2B) (prior to [C++11](https://en.wikipedia.org/wiki/C%2B%2B11)) that claims that if a [class](https://en.wikipedia.org/wiki/Class_(computer_science)) [defines](https://en.wikipedia.org/wiki/Declaration_(computer_science)) one (or more) of the following it should probably explicitly define all three:[[1]](https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)#cite_note-stroustrup-1)

* [destructor](https://en.wikipedia.org/wiki/Destructor_(computer_science))
* [copy constructor](https://en.wikipedia.org/wiki/Copy_constructor)
* [copy assignment operator](https://en.wikipedia.org/wiki/Assignment_operator_in_C%2B%2B)

These three functions are [special member functions](https://en.wikipedia.org/wiki/Special_member_functions). If one of these functions is used without first being declared by the programmer it will be implicitly implemented by the compiler with the default semantics of performing the said operation on all the members of the class. The default semantics are:

* **Destructor** – Call the destructors of all the object's class-type members
* **Copy constructor** – Construct all the object's members from the corresponding members of the copy constructor's argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type (e.g., *int* or pointer) data members
* **Copy assignment operator** – Assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type (e.g. *int* or pointer) data members.

---------------------------------------------------------------------------------------------------------------------------------

# What happens when more restrictive access is given to a derived class method in C++?

We have discussed a similar topic in Java [here](http://www.geeksforgeeks.org/archives/9629). Unlike Java, C++ allows to give more restrictive access to derived class methods. For example the following program compiles fine.

|  |
| --- |
| #include<iostream>  using namespace std;   class Base {  public:      virtual int fun(int i) { }  };   class Derived: public Base {  private:      int fun(int x)   {  }  };   int main()  {  } |

In the above program, if we change main() to following, will get compiler error becuase fun() is private in derived class.

|  |
| --- |
| int main(){      Derived d;      d.fun(1);      return 0;  } |

What about the below program?

|  |
| --- |
| #include<iostream>  using namespace std;   class Base {  public:      virtual int fun(int i) { cout << "Base::fun(int i) called"; }  };  class Derived: public Base {  private:      int fun(int x)   { cout << "Derived::fun(int x) called"; }  };  int main(){      Base \*ptr = new Derived;      ptr->fun(10);      return 0;  } |

Run on IDE

Output:

Derived::fun(int x) called

In the above program, private function “Derived::fun(int )” is being called through a base class pointer, the program works fine because fun() is public in base class. Access specifiers are checked at compile time and fun() is public in base class. At run time, only the function corresponding to the pointed object is called and access specifier is not checked. So a private function of derived class is being called through a pointer of base class.

--------------------------------------------------------------------------------------------------------------------------------------

# STL

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of [template classes](http://www.geeksforgeeks.org/template-specialization-c/) is a prerequisite  for working with STL.

STL has four components – Algorithms, Containers, Functions, Iterators

Algorithms – They act on containers and provide means for [sorting](http://geeksquiz.com/sort-algorithms-the-c-standard-template-library-stl), [searching](http://geeksquiz.com/binary-search-algorithms-the-c-standard-template-library-stl),etc. of the contents of the containers.

Containers – Containers or container classes store objects and data. They are further sub divided into :

Simple Containers : [pair](http://geeksquiz.com/pair-simple-containers-the-c-standard-template-library-stl)

Sequence Containers (ordered collections) : [vector](http://geeksquiz.com/vector-sequence-containers-the-c-standard-template-library-stl-set-1), [list](http://geeksquiz.com/list-sequence-containers-the-c-standard-template-library-stl), [deque](http://geeksquiz.com/deque-sequence-containers-the-c-standard-template-library-stl)

Container Adaptors : [queue](http://geeksquiz.com/queue-container-adaptors-the-c-standard-template-library-stl), [priority queue](http://geeksquiz.com/priority-queue-container-adaptors-the-c-standard-template-library-stl), [stack](http://geeksquiz.com/stack-container-adaptors-the-c-standard-template-library-stl)

Associative Containers : [set](http://geeksquiz.com/set-associative-containers-the-c-standard-template-library-stl), [multiset](http://geeksquiz.com/multiset-associative-containers-the-c-standard-template-library-stl), [map](http://geeksquiz.com/map-associative-containers-the-c-standard-template-library-stl), [multimap](http://geeksquiz.com/multimap-associative-containers-the-c-standard-template-library-stl), hash\_set, hash\_multiset, hash\_map, hash\_multimap

Functions – The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

Iterators – As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

-------------------------------------------------------------------------------------------------------------------------------------

# Smart Pointers in C++

Consider the following simple C++ code with normal pointers.

|  |
| --- |
| MyClass \*ptr = new MyClass();  ptr->doSomething();  //  We must do delete(ptr) to avoid memory leak |

Using [smart pointers](http://en.wikipedia.org/wiki/Smart_pointer), we can make pointers to work in way that we don’t need to explicitly call delete. [Smart pointer](http://en.wikipedia.org/wiki/Smart_pointer) is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.  
The idea is to make a class with a pointer, destructor and [overloaded operators](http://geeksquiz.com/operator-overloading-c/) like \* and ->. Since destructor is automatically called when an object goes out of scope, the dynamically alloicated memory would automatically deleted (or reference count can be decremented). Consider the following simple smartPtr class.

|  |
| --- |
| #include<iostream>  using namespace std;    class SmartPtr  {     int \*ptr;  // Actual pointer  public:     // Constructor: Refer <http://www.geeksforgeeks.org/g-fact-93/>     // for use of explicit keyword     explicit SmartPtr(int \*p = NULL) { ptr = p; }       // Destructor     ~SmartPtr() { delete(ptr); }       // Overloading dereferencing operator     int &operator \*() {  return \*ptr; }  };    int main()  {      SmartPtr ptr(new int());      \*ptr = 20;      cout << \*ptr;        // We don't need to call delete ptr: when the object      // ptr goes out of scope, destructor for it is automatically      // called and destructor does delete ptr.        return 0;  } |

Output:

20

**Can we write one smart pointer class that works for all types?**  
Yes, we can use [templates](http://geeksquiz.com/templates-cpp/) to write a generic smart pointer class. Following C++ code demonstrates the same.

|  |
| --- |
| #include<iostream>  using namespace std;    // A generic smart pointer class  template <class T>  class SmartPtr  {     T \*ptr;  // Actual pointer  public:     // Constructor     explicit SmartPtr(T \*p = NULL) { ptr = p; }       // Destructor     ~SmartPtr() { delete(ptr); }       // Overloading dereferncing operator     T & operator \* () {  return \*ptr; }       // Overloding arrow operator so that members of T can be accessed     // like a pointer (useful if T represents a class or struct or     // union type)     T \* operator -> () { return ptr; }  };    int main()  {      SmartPtr<int> ptr(new int());      \*ptr = 20;      cout << \*ptr;      return 0;  } |

Output:

20

Smart pointers are also useful in management of resources, such as file handles or network sockets.

# auto\_ptr

**auto\_ptr** is a class [template](https://en.wikipedia.org/wiki/Template_(programming)) available in the [C++](https://en.wikipedia.org/wiki/C%2B%2B) [Standard Library](https://en.wikipedia.org/wiki/C%2B%2B_standard_library) (declared in the <memory> [header file](https://en.wikipedia.org/wiki/Header_file)) that provides some basic [RAII](https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) features for [C++ raw pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming)#C_and_C.2B.2B).

The auto\_ptr template class describes an object that stores a pointer to a single allocated object that ensures that the object to which it points gets destroyed automatically when control leaves a scope.[[1]](https://en.wikipedia.org/wiki/Auto_ptr#cite_note-1)

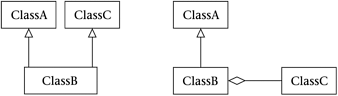
The [C++11](https://en.wikipedia.org/wiki/C%2B%2B11) standard made auto\_ptr deprecated, replacing it with the [unique\_ptr](https://en.wikipedia.org/wiki/Smart_pointer" \l "unique_ptr" \o "Smart pointer) class template.[[2]](https://en.wikipedia.org/wiki/Auto_ptr#cite_note-C.2B.2B11-2)[[3]](https://en.wikipedia.org/wiki/Auto_ptr#cite_note-3) The [shared\_ptr](https://en.wikipedia.org/wiki/Shared_ptr" \o "Shared ptr) template class defined in [C++11](https://en.wikipedia.org/wiki/C%2B%2B11), and available in the[Boost library](https://en.wikipedia.org/wiki/Boost_library), can be used as an alternative to auto\_ptr or unique\_ptr for collections with ownership semantics.[[4]](https://en.wikipedia.org/wiki/Auto_ptr#cite_note-4)

----------------------------------------------------------------------------------------------------------------------------------

**Why would you want to use composition in place of inheritance? There are several reasons.**

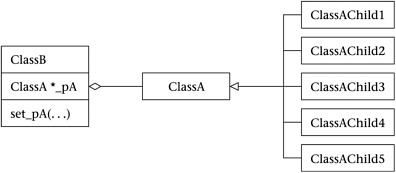
First, C++ code using multiple inheritance tends to be a bit difficult to maintain, and there are special MFC CRuntimeClass methods and macros that would need to be overridden if you want to use multiple inheritance. If you have a ClassB that you'd like to have inherit from both ClassA and ClassC, you can instead use composition for ClassA or ClassC. Figure 4.8 shows how this looks if we compose ClassB with ClassC.

##### **Figure 4.8. Use composition to avoid multiple inheritance**



Second, inheritance locks in a class's behavior at link time, while composition allows you to change the behavior of a class during runtime. This is illustrated in Figure 4.9. The ClassB has a set\_pA member method to delete the old \*\_pA and install a new one. In the Pop Framework, when you use the Player menu to change the player's controls, you are actually changing the kind of cListener \*\_plistener member which the playercCritter is composed with.

##### **Figure 4.9. Composition makes dynamic change possible**



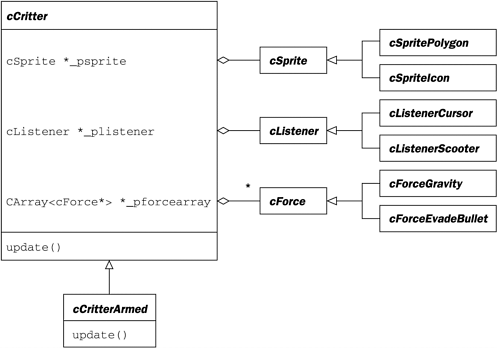
Third, inheritance is sometimes called 'white box' code reuse, because when you inherit from a class its internals are visible to you. Composition, on the other hand, is called 'black box' code reuse because (unless you've unwisely used a friend statement) the internals of the class you compose with are hidden. A practical advantage of black box code reuse is that you're less likely to break things that are used by classes other than your own. A useful mental model when using composition is that you're making a class by snapping together preexisting components.

A fourth and final reason why we often prefer composition to inheritance is that composition lets us avoid the 'combinatorial explosion' that we end up with if we try to separate out a class for every possible combination of the behaviors that we would otherwise delegate out to a composed member.

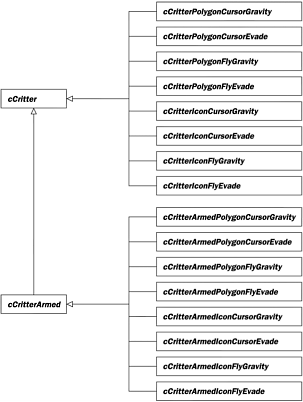
Of course there are still many situations where inheritance is the appropriate design method. Particularly if you're interested in having a polymorphic set of objects, it's good to have the objects inherit from a common base class. In the case of the Pop Framework, the cCritter base class plays this role. The individual cCritter child classes have constructors which compose specialized critters by 'snapping together' some component classes. And the individual cCritter update methods are usually overridden. We use inheritance so as to have a uniform list of cCritter child objects, and we use composition both to create new kinds of cCritter child classes and to possibly change the cCritters while the program is running.

Look, for instance, at the diagram of the critters and the classes they compose with (Figure 4.10). We see two kinds of critters, two kinds of sprites, two kinds of listeners, and two kinds of forces, eight classes in all. Now suppose that we wanted to avoid composition and put all of the behavior into the classes. Unless we use multiple inheritance, we'd end up with 18 classes: cCritter and cCritterArmed, with eight child classes each, one child for each of the eight ways of choosing polygon/icon, cursor/fly, or gravity/evade. This is illustrated in Figure 4.11. But if we can use composition to farm out the choices to helper classes, then we end up with a smaller number of classes in all.

##### **Figure 4.10. Critters and classes they are composed with**



##### **Figure 4.11. Combinatorial explosion of classes**

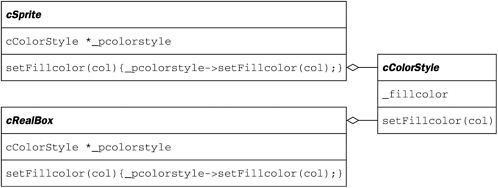


Now let's say a bit about the practicalities of composition and delegation. When you compose ClassB with a ClassA member \_mA (or with a \*\_pA member) the owner ClassB will need to use ClassA accessors and mutators to get at the ClassA object's data. You can get around this by having ClassA declare ClassB as a friend, but generally we try to avoid friend statements as they break encapsulation.

When you use composition with delegation as illustrated in Figure 4.7, you need to explicitly declare and implement a ClassB function like foo() which is intended to pass off the call to the ClassA member method foo(). This is different from inheritance, where a child class automatically gets the methods of the parent class. In the case of composition, you of course don't have to give the ClassA method the same name as the ClassB method which it calls. In fact it is likely to make your code easier to understand if you give the ClassA method a name like 'feelfoo' or 'dofoo' or 'callfoo.'

A fairly trivial example of composition is that we give both our cSprite and our cRealBox classes a cColorStyle\*\_pcolorstyle member which holds things like the fill color to be used for the shape. The color-related mutators and accessors for cSprite and cRealBox pass the calls on to their \_pcolorstyle members. This is shown in Figure 4.12.

##### **Figure 4.12. Simple composition**



Why didn't we just make cSprite and cRealBox inherit from, say, a common cUsesColorStyle class? The reasons were that (a) other than being drawable with colors, the two classes really have nothing in common and, more importantly and (b) as we move from two-dimensional graphics to three-dimensional graphics, we'd like to allow the possibility of using richer and more complicated kinds of cColorStyle child classes to specify the colors and styles of our sprites and world boxes.

A less obvious point about delegation is that when ClassB delegates a method like foo(), you often want foo to be able to access and mutate the members of ClassB. If ClassB has a ClassA \*\_pA member, the correct way to delegate foo() so that it can access and mutate ClassB is the following.

ClassB::foo()

{

\_pA->foo(this);

}

ClassA::foo(ClassA \*powner)

{

/\* Use ClassA accessors and mutators to read and change the fields

of powner \*/

}

In the specific example of the cCritter and the cListener \*\_plistener that it's composed with, we have the following code.

void cCritter::listen(Real dt)

{

\_plistener->listen(this); /\* We pass the pointer "this" to the

listener so that it can change the fields of this calling

cCritter as required. \*/

}

void cListenerScooter::listen(cCritter \*pcritter)

{

cController \*pcontroller = pcritter->pgame()->pcontroller();/\*The

caller critter's pgame() holds the cController object that

stores all of the keys and mouse actions you need to possibly

listen to in here.\*/

//Translate

if (pcontroller->keyonplain(VK\_UP))

pcritter->setVelocity(pcritter->maxspeed()\*

pcritter->tangent());

/\* I want to move the critter position. But I don't just

use a moveTo because I want to have a correct \_velocity

inside the critter so I can use it to hit things and

bounce and so on. So I change the velocity. \*/

//Etcetera....

}

In other cases it may be that the foo call does some setup code before passing the call off to the composed object. This is the situation wherecCritter delegates some of its draw call to its cSprite \*\_psprite member. The matrix manipulations serve to translate and rotate the graphics frame of reference to match the critter's position and orientation.

void cCritter::draw(cGraphics \*pgraphics, int drawflags)

{

pgraphics->pushMatrix();

pgraphics->multMatrix(attitude());

\_psprite->draw(pgraphics, drawflags);

pgraphics->popMatrix();

}

**Use of Pure Virtual Destructor:**

To work correctly, classes with virtual methods must also have virtual destructors. Interestingly, virtual destructors can be declared pure, which can be useful in some cases.

Imagine you have a base class you want to make abstract. In this base class all methods have meaningful default implementations, and you want to allow the derived classes to inherit them as-is. However, to make a class abstract, at least one of its methods must be made pure virtual, which means the derived classes must override it. How do you make the class abstract in this case?

The answer is: declare the destructor pure virtual. This will make your class abstract without forcing you to declare any other method pure virtual.

// Abstract base class - can't be instantiated

//

**class** Base

{

**public**:

**virtual** ~Base() = 0;

**virtual** **void** method();

};

Base::~Base()

{

// Compulsory virtual destructor definition,

// even if it's empty

}

**void** Base::method()

{

// Default implementation.

// Derived classes can just inherit it, if needed

}

// We can now derive from Base, inheriting the

// implementation of method()

//

**class** Derived : **public** Base

{

**public**:

~Derived()

{}

};

While defining (providing an implementation) pure virtual methods is rarely useful, you *must* define a pure virtual destructor. This is because the destructor of a base class is always called when a derived object is destroyed. Failing to define it will cause a link error.

# Function overloading and const keyword

Predict the output of following C++ program.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  protected:      int x;  public:      Test (int i):x(i) { }      void fun() const      {          cout << "fun() const called " << endl;      }      void fun()      {          cout << "fun() called " << endl;      }  };    int main()  {      Test t1 (10);      const Test t2 (20);      t1.fun();      t2.fun();      return 0;  } |

Run on IDE

Output: The above program compiles and runs fine, and produces following output.

fun() called

fun() const called

The two methods ‘void fun() const’ and ‘void fun()’ have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, ‘const void fun()’ is called on const object and ‘void fun()’ is called on non-const object.  
C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function return reference or pointer. We can make one function const, that returns a const reference or const pointer, other non-const function, that returns non-const reference or pointer. See [this](http://www.parashift.com/c++-faq-lite/const-overloading.html" \t "_blank)for more details.

**What about parameters?**  
Rules related to const parameters are interesting. Let us first take a look at following two examples. The program 1 fails in compilation, but program 2 compiles and runs fine.

|  |
| --- |
| // PROGRAM 1 (Fails in compilation)  #include<iostream>  using namespace std;    void fun(const int i)  {      cout << "fun(const int) called ";  }  void fun(int i)  {      cout << "fun(int ) called " ;  }  int main()  {      const int i = 10;      fun(i);      return 0;  } |

Run on IDE

Output:

Compiler Error: redefinition of 'void fun(int)'

|  |
| --- |
| // PROGRAM 2 (Compiles and runs fine)  #include<iostream>  using namespace std;    void fun(char \*a)  {    cout << "non-const fun() " << a;  }    void fun(const char \*a)  {    cout << "const fun() " << a;  }    int main()  {    const char \*ptr = "GeeksforGeeks";    fun(ptr);    return 0;  } |

Run on IDE

Output:

const fun() GeeksforGeeks

C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer. That is why the program 1 failed in compilation, but the program 2 worked fine. This rule actually makes sense. In program 1, the parameter ‘i’ is passed by value, so ‘i’ in fun() is a copy of ‘i’ in main(). Hence fun() cannot modify ‘i’ of main(). Therefore, it doesn’t matter whether ‘i’ is received as a const parameter or normal parameter. When we pass by reference or pointer, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which can not.

As an exercise, predict the output of following program.

|  |
| --- |
| #include<iostream>  using namespace std;    void fun(const int &i)  {      cout << "fun(const int &) called ";  }  void fun(int &i)  {      cout << "fun(int &) called " ;  }  int main()  {      const int i = 10;      fun(i);      return 0;  } |

# Copy elision in C++

[Copy elision](http://en.wikipedia.org/wiki/Copy_elision) (or Copy omission) is a compiler optimization technique that avoids unnecessary copying of objects.Now a days, almost every compiler uses it. Let us understand it with the help of an example.

|  |
| --- |
| #include <iostream>  using namespace std;    class B  {  public:      B(const char\* str = "\0") //default constructor      {          cout << "Constructor called" << endl;      }        B(const B &b)  //copy constructor      {          cout << "Copy constructor called" << endl;      }  };    int main()  {      B ob = "copy me";      return 0;  } |

Run on IDE

The output of above program is:

Constructor called

**Why copy constructor is not called?**  
According to theory, when the object “ob” is being constructed, one argument constructor is used to convert “copy me” to a temporary object & that temporary object is copied to the object “ob”. So the statement

B ob = "copy me";

should be broken down by the compiler as

B ob = B("copy me");

However, most of the C++ compilers avoid such overheads of creating a temporary object & then copying it.

The modern compilers break down the statement

B ob = "copy me"; //copy initialization

as

B ob("copy me"); //direct initialization

and thus eliding call to copy constructor.

However, if we still want to ensure that the compiler doesn’t elide the call to copy constructor [disable the copy elision], we can compile the program using “-fno-elide-constructors” option with g++ and see the output as following:

aashish@aashish-ThinkPad-SL400:~$ g++ copy\_elision.cpp -fno-elide-constructors

aashish@aashish-ThinkPad-SL400:~$ ./a.out

Constructor called

Copy constructor called

If “-fno-elide-constructors” option is used, first default constructor is called to create a temporary object, then copy constructor is called to copy the temporary object to ob.

# Malloc Vs Calloc

malloc() allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block.

# void \* malloc( size\_t size );

malloc() doesn’t initialize the allocated memory.

calloc() allocates the memory and also initializes the allocates memory to zero.

|  |
| --- |
| void \* calloc( size\_t num, size\_t size ); |

Unlike malloc(), calloc() takes two arguments: 1) number of blocks to be allocated 2) size of each block.

We can achieve same functionality as calloc() by using malloc() followed by memset(),

|  |
| --- |
| ptr = malloc(size);  memset(ptr, 0, size); |

# Type Conversion

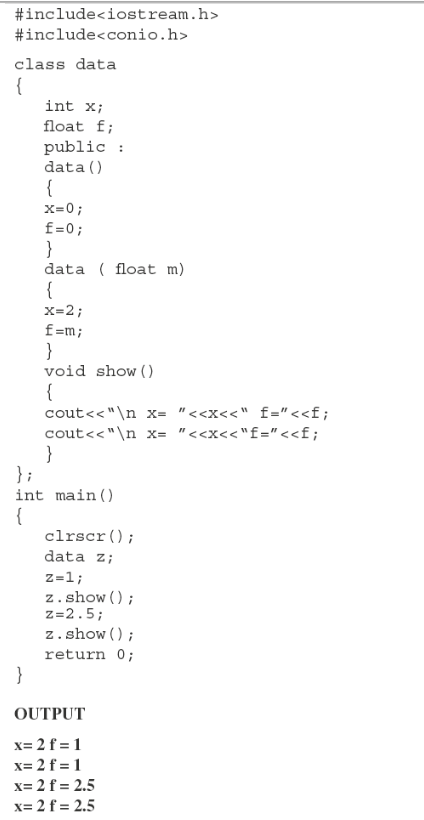
We learnt that when constants and variables of various data types are clubbed in a single expression, automatic type conversion takes place. This is so for basic data types. The compiler is unknown about the user-defined data type and about their conversion to other data types. The programmer should write the routines that convert basic data type to user-defined data type or vice versa. There are three possibilities of data conversion as given below:

1. Conversion from basic data type to user-defined data type (class type)
2. Conversion from class type to basic data type
3. Conversion from one class type to another class type

## Conversion from Basic to Class Type

The conversion from basic to class type is easily carried out. It is automatically done by the compiler with the help of in-built routines or by applying type casting. In this type, the left-hand operand of = sign is always class type and the right-hand operand is always basic type. The below-given program explains the conversion from basic to class type.

*Q) Write a program to define constructor with no argument and with float argument. Explain how compiler invokes constructor depending on data type.*

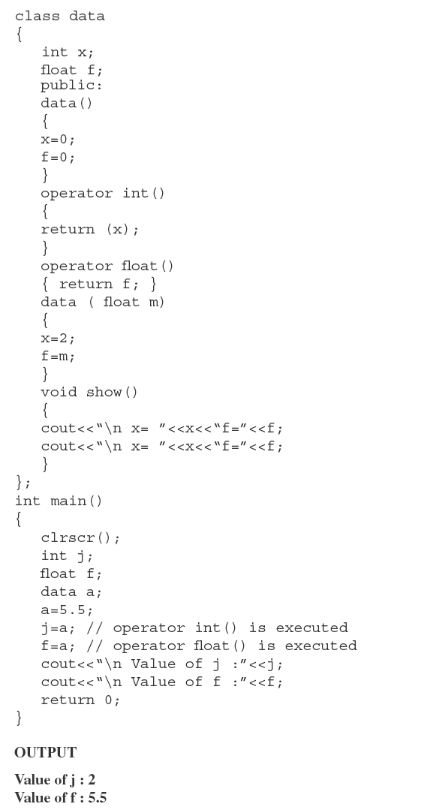


**Explanation:** In the above program, the class data has two member variables each of integer and float types respectively. It also has two constructors one with no argument and the other with float argument. The member function show() displays the contents of the data members. In function main(), z is an object of class data. When z is created, the constructor with no argument is called and data members are initialized to zero. When z is initialized to one, the constructor with float argument is invoked. The integer value is converted to float type and assigned member variable f. Again when z is assigned to 2.5, same process is repeated. Thus, the conversion from basic to class type is carried out.

## Conversion from Class Type to Basic Data Type

The conversion from basic to class type is easily carried out. It is automatically done by the compiler with the help of in-built routines or by applying type casting. In this type, the left-hand operand of = sign is always class type and the right-hand operand is always basic type. The below-given program explains the conversion from basic to class type.

*Q) Write a program to define constructor with no argument and with float argument. Explain how compiler invokes constructor depending on data type.*



**Explanation:** In the above program, the class data has two member variables each of integer and float data type. It also contains constructors as per described in the last example. In addition, it contains overloaded data types int and float. These functions are useful for conversion of data from class type to basic type. Consider the following statements:

(a) j=a;

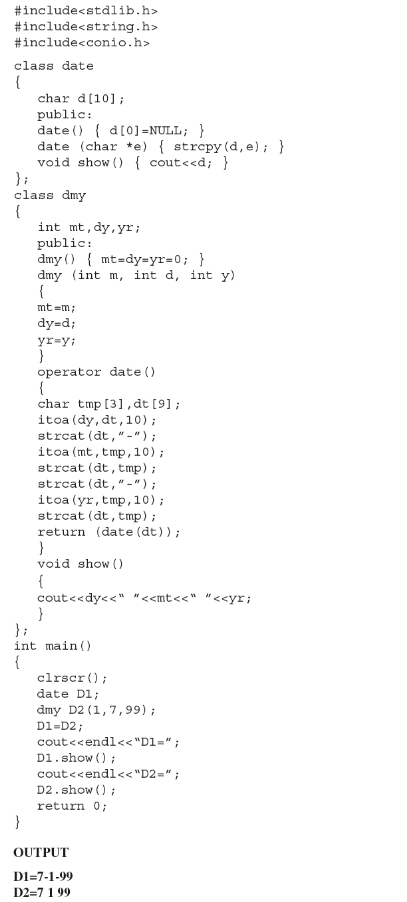
(b) f=a;

In the first statement object a is assigned to integer variable j. We know that class type data is a combination of one or more basic data types. The class contains two member functions operator int() and operator float(). Both these function are able to convert data types from class to basic. In statement (a), variable j is of integer type, the function operator int() is invoked and integer value data member is returned. In statement (b), f is of float type, the member function operator float() is invoked.

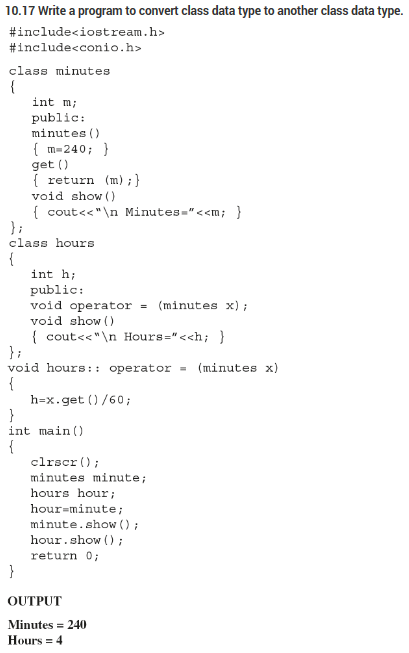
## Conversion from One Class Type to Another Class Type

We learnt how to convert basic data type to class type and vice versa. Now the third method is conversion from class type to another class type. When an object of one class is assigned to object of another class, it is necessary to give clear-cut instructions to the compiler about how to make conversion between these two user-defined data types. The method must be instructed to the compiler. There are two ways to convert object data type from one class to another. One is to define a conversion operator function in source class or a one-argument constructor in a destination class. Consider the following example: X=A; Here, X is an object of class XYZ and A is an object of class ABC. The class ABC data type is converted to class XYZ. The conversion happens from class ABC to XYZ. The ABC is a source class and XYZ is a destination class. We know the operator function operator data-type(). Here, data type may be built-in data type or userdefined data type. In the above declaration, the data type indicates target type of object. Here, the conversion takes place from class ABC (source class) to class XYZ (destination class).

*Q) Write a program to convert integer to date and vice versa using conversion function in source class.*







# Strategy pattern