# Classes

## vPtr and vtable

### 1. A Class and its Objects in the Memory

When a class is first-time accessed in a program, e.g. an object of this class is created or a static method is called, **the class definition including one unique copy of its methods is loaded into the memory**. Each method has its own memory address.

Then, every time when an instance of this class is created, a copy of the data members of this class is created and put in a block of memory. Data members are placed in a fixed order one next to another. The address of an object is the address of the first byte of its memory block.

### 2. Linking of a Method Call

When compiler sees a method call such as

MyClass \*p = new MyClass;

p->**method** ();

it needs to link the individual data copy of the object to the unique implementation of the method. If the method is non-virtual, this task is done at compile time, while if the method is virtual, it is done at run time.

### 3. Memory Footprint of a Non-polymorphic Object

Suppose class **Derived** inherits from **Base1**, **Base2**, **Base3**, and none of them has virtual functions:

#include <stdio.h>

class **Base1** {

public:

char **a1**[100];

void **Hi1**(){ printf("Hi from Base1!\n"); }

};

class **Base2** {

public:

char **a2**[100];

void **Hi2**(){ printf("Hi from Base2!\n"); }

};

class **Base3** {

public:

char **a3**[100];

void **Hi3**(){ printf("Hi from Base3!\n"); }

};

class **Derived** : public Base1, public Base2, public Base3 {

public:

char **a**[100];

void **Hi**(){ printf("Hi from Derived!\n"); }

};

int main()

{

Derived \* **pDerived** = new Derived;

Base1 \* **pBase1** = (Base1 \*)pDerived;

pBase1->Hi1();

Base2 \* **pBase2** = (Base2 \*)pDerived;

pBase2->Hi2();

Base3 \* **pBase3** = (Base3 \*)pDerived;

pBase3->Hi3();

delete pDerived;

}

The memory footprint of an object of class **Derived** and the addresses of the pointers will be like:

**pDerived**   
pBase1 pBase2 pBase3  
↓ ↓ ↓

|  |  |  |  |
| --- | --- | --- | --- |
| a1 | a2 | a3 | a |

0 100 200 300 400

Fig 1. Memory footprint of a non-polymorphic type

From this footprint you can see some important facts:

1. An object of a non-polymorphic class doesn’t need to carry any information about addresses of the methods, because the linking of non-virtual functions is already done by compiler at compile time. So the object contains purely data members. The references such as pBase1, pBase2, pBase3, etc. are only used to access the data members, not the methods.
2. The memory of base classes is allocated in front of the derived class, which conforms to the sequence of object construction.
3. The address of the FIRST base class is the same as the derived class.
4. When a pointer of the derived class is casted to the base class, it is pointing to the base-class part of the memory.

### 4. Non-virtual Function Call

When compiler sees a call to a non-virtual method,

because only the class which declares the method can implement it, compiler will directly link the call to the address of the specific method at compile time.

### 5. Virtual Function Call Using vPtr and vtable

When compiler sees a call to a virtual method:

void Go(Vehicle \* pv)

{

pv->**StartEngine**();

pv->**Accelerate**();

}

it has no idea on the addresses of the virtual methods, because when method **Go** is called at run time, the parameter **Vehicle \* pv** can be passed a pointer to an object of any derived class, such as **FamilyCar**, **4WD**, **PickUpTruck**, **Van**, etc., each with its own implementations of **StartEngine** and **Accelerate** at different memory locations.

Therefore, there must be a mechanism for a program to figure out the locations of the virtual methods at run time.

Now consider the following virtual function call:

pBase2->Hi2();

A late-binding process involves the following activities:

1. Compiler adds a hidden **vPtr** member to the class, and generates one unique **vtable** for the class.

At compilation time, when compiler sees the definition of a class with virtual methods, it will **build a virtual table** (vtable) for the class, which is an array of function pointers to the implementations of all the virtual methods, and **add a hidden data member vPtr** to the class definition as the FIRST data member.

Now suppose the methods of classes in Fig. 1 (Hi, Hi1, Hi2, Hi3) are all virtual functions. The memory footprint of an object of class **Derived** becomes:

pDerived   
pBase1 pBase2 pBase3  
↓ ↓ ↓

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| vPtr | **a1** | vPtr2 | **a2** | vPtr3 | **a3** | **a** |

0 4 104 108 208 212 312 412

**a1** are data members of base class 1, **a2** are of base class2, …, **a** are of base class 1.

Fig. 2. Memory footprint of a polymorphic type object

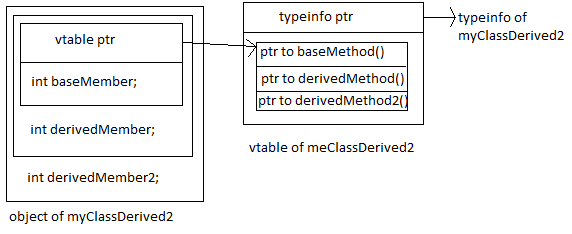
As shown above in the memory footprint, if you use a **Base2** pointer to receive a **Derived** object, for example, this pointer will point to memory offset 104 as **pBase2** does.

Note that each **Derive** object will have its own memory footprint, with the same structure but in different memory locations. However, the **vPtr**s will all be pointing to the same method implementations, in other words, the **vPtr2** of two instances will contain the same address.

The derived-class and the first base class shares the same **vPtr**, which points to their shared merged **vtable** (see following section “Inheritance of Base-class vPtrs” for details). The rest of the base classes have their own **vPtr**s.

Note that no matter how complicated the inheritance hierarchy is, a function pointer in the **vtable** always points to the latest/lowest implementation of the virtual function in the inheritance hierarchy.

An example memory layout is below, if class hierarchy is base->derived->derived2



1. Compiler generates code to do dynamic binding using the **vtable**.

At compilation time, when compiler sees a call to a virtual method through a pointer (pBase2->Hi2( )), it knows that the address of the function is only known at run time, so it will not try to find the implementation of the function. Instead, it knows that the pointer (pBase2) will be pointing to a **vPtr** at run time. So it generates code to go through the **vPtr** to find the **vtable** (whose composition is already know from the type of the pointer), and go to a certain entry of that **vtable**, fetch that function pointer, and make the call.

1. At run time, when an object is created out of this class definition, its **vPtr** member will be assigned the address of the class’s **vtable**.

### 6. A Language Needs Compiler and Type Information to Utilize vtable

To directly link the address of the implementation of a function at compile time (for a non-polymorphic object), or to generate code to find it out at run time (for a polymorphic one), the compiler needs to access the class definition of the pointer, and the type definition of the pointer is all it wants to know. Because of this, even if **pBase2** is actually pointing to an object of class **Derived**, it cannot access class **Derived**’s method **Hi**, because the code generated by the compiler only knows the **vtable** of class **Base2**. For the same reason, the following function call doesn’t work:

void \*pvoid = new Derived;

pvoid->Hi();

Scrip languages such as VBScript, JavaScript and early versions of VB do not have a compiler and do not do compilation. They explain and run the code line by line at run time. Therefore, they cannot generate code before run to access the **vtable**. So they cannot utilize **vtable** and cannot directly use polymorphism.

### 7. Polymorphic Base Classes are placed at the front

Now suppose only Base2 and Base3 have virtual methods while Base1 doesn’t. The memory footprint of Derived will become:

pDerived   
pBase2 pBase3 pBase1  
↓ ↓ ↓

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| vPtr | a2 | vPtr | a3 | a1 | a |

0 4 104 108 208 308 408

Fig. 3. Memory footprint of a polymorphically mixed type

You can see that the memory block of polymorphic base classes (Base2 and Base3) are moved to the front, while the non-polymorphic base classes (Base1) are moved to the back.

### 8. Inheritance of Base-class vPtrs

From the Fig.2 you can see, the **vPtr** member of the second and third base class was inherited, but the **vPtr** of the first base class is not. Suppose class D inherits only from C, C only from B, and B only from A. When compiler construct the **vtable** for D, it directly merges the **vtables** of A, B and C into D’s, so that class D has only one **vPtr** pointing to one **vtable**. The **vtable** contains function pointers to the lowest implementations of all virtual functions across the inheritance hierarchy, no matter where they are defined.

However, due to multiple inheritance, a class may indirectly inherit from many classes. If we decide to merge the **vtables** of all the base classes into one, the **vtable** may become very big. To avoid this, instead of discarding the **vPtrs** and **vtables** of all base classes and merging all **vtables** into one, the compiler only does it to all the FIRST base classes, and retains the **vPtrs** and **vtables** of all the subsequent base classes and their base classes.

In other words, in an object’s memory footprint, you can find the **vPtr**s of all its base classes all through the hierarchy, except for all the “first-borns”.

In Fig. 2, virtual function **Hi3** is defined in class **Base3**. When it is called:

pDerived->Hi3();

even if it is implemented in class **Derived**, the program will still go to the **vPtr** of **Base3** (offset 208 in the footprint), then to **Base3**’s **vtable**, then to **Hi3**’s function pointer, which points back to the implementation in class **Derived**. We can prove it by setting **Base3**’s **vPtr** in offset 208 to 0. Thus the address of **Base3**’s **vtable** is lost, and error will cause the program to be shut down:

pd->Hi3(); // virtual function Hi3 is defined in Base3. Works fine here

::memset( (Base3 \*)((DWORD)pd + 208), 0, 4);

pd->Hi3(); // Error happens!

In a typical case, the derived class inherits from a group of base classes, which can be interfaces (classes with only public pure virtual functions) or classes with some implementations. The derived class does not define any new virtual function, it only implements the virtual functions defined in the base classes, or simply group the services of the base classes together. In such a case, the **vtable** of the derived class is simply the **vtable** of the first base class. Therefore, the first **vPtr** in the derived-class object points to the first base class’s **vtable**, the second **vPtr** points to the second base class’s **vtable**, and so on. Everything is clean and neat.

### 9. De-referencing the vPtr

Comparing Fig.2 with Fig.1, because the binding of virtual methods are only done at run time, the object contains not only data members but also **vPtr**s telling the addresses of all the virtual functions. When you have a reference say **pBase2**, you know that the first four bytes from the given address is a **vPtr** of **Base2**, followed by the data members of **Base2**.

Base2 \* pBase2 = new Derived;

pBase2->Hi2(); // OK!

cout << pBase2->a2[3]; // OK!

pBase2->Hi3(); // Compile error: 'Hi2' : is not a member of 'Base3'

Because of this structure, if you only want to access virtual functions defined in Base2 and its own part of data members, not the virtual functions and data members of other classes, what you really need is the address of Base2’s **vPtr**.

## Virtual Functions

A virtual function is a ***member function that you expect to be redefined in derived classes*** (<https://docs.microsoft.com/en-us/cpp/cpp/virtual-functions?view=vs-2017>). It is a member function which is ***declared within base class*** and is ***re-defined (Overridden) by derived*** class (<https://www.geeksforgeeks.org/virtual-function-cpp/>). 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 are declared with a **virtual**keyword in base class.
* They are mainly used to achieve[runtime polymorphism](https://www.geeksforgeeks.org/polymorphism-in-c/)
* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* The resolving of function call is done at Run-time.

### ****Rules for Virtual Functions****

1. They Must be declared in public section of class.
2. Virtual functions cannot be static
3. Virtual functions cannot be a friend function of another class.
4. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
5. The prototype (signature) of virtual functions should be same in base as well as derived class.
6. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
7. A class may have [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.
8. Even the virtual destructor is pure, it still has to have a body. But you have to define the body outside the class because ***virtual ~base()=0* {};** is illegal

### No Virtual Constructor

Question: Can you have virtual constructor? explain your answer.  
<http://en.allexperts.com/q/C-1040/2009/5/VIRTUAL-Constructor-destructor.htm>

No, at that time, the **vtbl** is not constructed and the function call can't be dynamically bounded

In all the uses of polymorphism we have seen you require objects to start with - even if you then refer to them via some pointer or reference to a base type. When we create an object we have to know its exact type – how else would a compiler know what to create otherwise. Saying I want something that is either a base type or something derived from it is far too vague. In C++ if you ask for an instance of type T then an instance of type T is exactly what you get (unless for some reason such a request fails). When we create an object there is no existing object or reference/pointer to an object - or reference/pointer to a base object - we can use for a hint.

So if you can define a virtual constructor, how would it work?

The polymorphic behavior that virtual member functions allow us to achieve work through base pointers or references to existing objects.

How then can you polymorphically directly create an object? Where is the base pointer or reference that refers to a specific object type instance that can be used?

Oops, we do not have one because we have not created the specific object type instance yet!

Constructors are always non-virtual and refer to the object type in which they are defined. By default, a derived class instance will call its base classes' default constructors before it executes to ensure they are all initialized before any derived initialization is done. In fact, **during construction the virtual function call mechanism is just initialized so calling virtual functions from a base class constructor will not call a derived override because that part of the object has not yet been initialized** (a similar case exists during destruction: objects are destroyed derived to base, in the reverse order of construction).

You might have noticed that I previously used the term "directly create an object" - meaning directly via a constructor. However, we can create "virtual constructor" like virtual member functions. These would work by creating new objects from existing objects. Two variations are obviously possible for all cases: a virtual create method that creates a new object using the default or other constructor (all classes in the hierarchy must have a compatible constructor) and a virtual clone method that creates new objects using the copy constructor (thereby making a clone of the object on which clone is called - all such types must be copyable of course).

Here is a simple example:

class AbstractBase{

public:

virtual ~AbstractBase() {}

virtual AbstractBase \* Create() = 0;

virtual AbstractBase \* Clone() = 0;

// ...

};

class SomeType : public AbstractBase{

public:

virtual SomeType \* Create();

virtual SomeType \* Clone();

// ...

};

SomeType \* SomeType::Create(){

return new SomeType;

}

SomeType \* SomeType::Clone(){

return new SomeType(\*this);

}

class SomeOtherType : public AbstractBase{

public:

virtual SomeOtherType \* Create();

virtual SomeOtherType \* Clone();

// ...

};

SomeOtherType \* SomeOtherType::Create(){

return new SomeOtherType;

}

SomeOtherType \* SomeOtherType::Clone(){

return new SomeOtherType(\*this);

}

### Calling (pure) virtual function from Constructor

During construction the virtual function call mechanism is just initialized, so calling virtual functions from a base class constructor will not call a derived override because that part of the object has not yet been initialized.

### Cannot Be Friend Function

One cannot ***declare*** a virtual function “friend” because “friend” is applied to non-member functions while virtual functions must be member functions.

However, you can declare a “friend” function in the base class which calls a virtual function, and thus achieving “almost virtual” effect – this is called “virtual friend idiom”. This way, you do not need to overload friend function for every class in the hierachy. (<https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Virtual_Friend_Function>)

A canonical example is to overload << operator, which is often a friend, to achieve seamless streaming capabilities.

**#include** <iostream>

**using** **namespace** std;

**class** **Base** {

**public**:

**friend** ostream& **operator** << (ostream& o, **const** Base& b);

**protected**:

**virtual** void print(ostream& out) **const**

{ out << "virtual base::print"; } { ... }

};

**class** derived : **public** base{

**public**:

**virtual** void print(ostream& out) **const**

{ out << "virtual derived::print"; }

};

std::ostream& **operator**<<(std::ostream& os, **const** base& b ){

b.print(os);

return os;

}

int main(){

derived d;

std::cout << d << std::endl;

return 0;

}

### abstract class

(<https://en.cppreference.com/w/cpp/language/abstract_class>) An *abstract class* is a class that either defines or inherits at least one function for which [the final overrider](https://en.cppreference.com/w/cpp/language/virtual) is *pure virtual*.

Abstract classes are used to represent general concepts (for example, Shape, Animal), which can be used as base classes for concrete classes (for example, Circle, Dog).

Abstract classes cannot be instantiated. No objects of an abstract class can be created.

Abstract types cannot be used as parameter types, as function return types, or as the type of an explicit conversion (note this is checked at the point of definition and function call, since at the point of function declaration parameter and return type may be incomplete). However, the references or pointers of abstract types can.

## Constructors and destructors

### Explicit Constructor

A constructor declared explicit cannot be instantiated by using “=”; and it cannot be used in implicit type conversions.

### Destructors and virtual

<http://www.codeguru.com/cpp/tic/tic0163.shtml>

|  |  |
| --- | --- |
| [Bruce Eckel's Thinking in C++, 2nd Ed](http://www.bruceeckel.com/ThinkingInCPP2e.html) | [Contents](http://www.codeguru.com/cpp/tic/tic_c.shtml) | [Prev](http://www.codeguru.com/cpp/tic/tic0162.shtml) | [Next](http://www.codeguru.com/cpp/tic/tic0164.shtml) |

Constructors cannot be made explicitly virtual (and the technique in Appendix B only simulates virtual constructors), but destructors can and often must be virtual.

The constructor has the special job of putting an object together piece-by-piece, first by calling the base constructor, then the more derived constructors in order of inheritance. Similarly, the destructor also has a special job – it must disassemble an object that may belong to a hierarchy of classes. To do this, the compiler generates code that calls all the destructors, but in the ***reverse*** order that they are called by the constructor. That is, the destructor starts at the most-derived class and works its way down to the base class. This is the safe and desirable thing to do: The current destructor always knows that the base-class members are alive and active because it knows what it is derived from. Thus, the destructor can perform its own cleanup, then call the next-down destructor, which will perform *its* own cleanup, knowing what it is derived from, but not what derives from it.

You should keep in mind that **constructors and destructors are the only places where this hierarchy of calls must happen** (and thus the proper hierarchy is automatically generated by the compiler). ***In all other functions, only that function will be called, whether it’s virtual or not***. The only way for base-class versions of the same function to be called in ordinary functions (virtual or not) is if you *explicitly* call that function.

Normally, the action of the destructor is quite adequate. But what happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? This is certainly a major objective in object-oriented programming. The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete**. Sound familiar? This is the same problem that virtual functions were created to solve for the general case. Fortunately, virtual functions work for destructors as they do for all other functions except constructors.

Even though the destructor, like the constructor, is an “exceptional” function, it is possible for the destructor to be virtual because the object already knows what type it is (whereas it doesn’t during construction). Once an object has been constructed, its VPTR is initialized, so virtual function calls can take place.

(*Looks like the following is incorrect. We can still define pure virtual destructor with its function body defined outside the class – this can be verified. The correct part is: virtual destructors, pure or not, must have a function body* ***because all destructors in a class hierarchy will be called****.*)

For a time, pure virtual destructors were legal and worked if you combined them with a function body, but in the final C++ standard function bodies combined with pure virtual functions were outlawed. This means that a virtual destructor cannot be pure, and must have a function body because (unlike ordinary functions) all destructors in a class hierarchy are always called. Here’s an example:

//: C15:Pvdest.cpp

// Pure virtual destructors

// require a function body.

#include <iostream>

using namespace std;

class Base {

public:

virtual ~Base() {

cout << "~Base()" << endl;

}

};

class Derived : public Base {

public:

~Derived() {

cout << "~Derived()" << endl;

}

};

int main() {

Base\* bp = new Derived; // Upcast

delete bp; // Virtual destructor call

} ///:~

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

### No-Virtual mechanism in destructor/constructor

There’s something that happens during destruction that you might not immediately expect. If you’re inside an ordinary member function and you call a virtual function, that function is called using the late-binding mechanism. This is not true with destructors, virtual or not. Inside a destructor, only the “local” version of the member function is called; the virtual mechanism is ignored.

Why is this? Suppose the virtual mechanism *were* used inside the destructor. Then it would be possible for the virtual call to resolve to a function that was “further out” (more derived) on the inheritance hierarchy than the current destructor. But destructors are called from the “outside in” (from the most-derived destructor down to the base destructor), so the actual function called would rely on portions of an object that has ***already been destroyed****!* Thus, the compiler resolves the calls at compile-time and calls only the “local” version of the function. Notice that the same is true for the constructor (as described earlier), but ***in the constructor’s case the information wasn’t available***, whereas in the destructor the information (that is, the VPTR) is there, but is isn’t reliable.

### Pure virtual destructors

<http://www.cplusplus.com/forum/general/12712/>

While pure virtual destructors are legal in Standard C++, there is an added constraint when using them:

***You must provide a function body for the pure virtual destructor.***

Otherwise you will get compiler error like “undefined reference to \*\*\*\*\*\*”

This seems counterintuitive; how can a virtual function be “pure” if it needs a function body?

But if you keep in mind that constructors and destructors are special operations it makes more sense, especially if you remember that all destructors in a class hierarchy are always called.

If you could leave off the definition for a pure virtual destructor, what function body would be called during destruction?

Thus, it’s absolutely necessary that the compiler and linker enforce the existence of a function body for a pure virtual destructor.

If it’s pure, but it has to have a function body, what’s the value of it?

The only difference you’ll see between the pure and non-pure virtual destructor is that the pure virtual destructor does cause the base class to be abstract, so you cannot create an object of the base class (although this would also be true if any other member function of the base class were pure virtual).

Things are a bit confusing, however, when you inherit a class from one that contains a pure virtual destructor. Unlike every other pure virtual function, you are not required to provide a definition of a pure virtual destructor in the derived class.

### Never call base class destructor

<https://isocpp.org/wiki/faq/dtors>

You never need to explicitly call a destructor (except with ***placement new***).

A derived class’s destructor (whether or not you explicitly define one) automatically invokes the destructors for base class subobjects. Base classes are destructed after member objects. In the event of multiple inheritance, direct base classes are destructed in the reverse order of their appearance in the inheritance list.

### exception in destructor

<https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw>

Write a message to a log-file. Terminate the process. Or call Aunt Tilda. But do not throw an exception!

Here’s why (buckle your seat-belts):

The C++ rule is that you must never throw an exception from a destructor that is being called during the “**stack unwinding**” process of another exception. For example, if someone says throw Foo(), the stack will be unwound so all the stack frames between the throw Foo() and the } catch (Foo e) { will get popped. This is called stack unwinding.

During stack unwinding, all the local objects in all those stack frames are destructed. If one of those destructors throws an exception (say it throws a Bar object), the C++ runtime system is in a no-win situation: should it ignore the Bar and end up in the } catch (Foo e) { where it was originally headed? Should it ignore the Foo and look for a } catch (Bar e) { handler? There is no good answer — either choice loses information.

So the C++ language guarantees that it will call terminate() at this point, and terminate() kills the process. Bang you’re dead.

The easy way to prevent this is never throw an exception from a destructor. But if you really want to be clever, you can say never throw an exception from a destructor while processing another exception. But in this second case, you’re in a difficult situation: the destructor itself needs code to handle both throwing an exception and doing “something else”, and the caller has no guarantees as to what might happen when the destructor detects an error (it might throw an exception, it might do “something else”). So the whole solution is harder to write. So the easy thing to do is always do “something else”. That is, never throw an exception from a destructor.

Of course the word never should be “in quotes” since there is always some situation somewhere where the rule won’t hold. But certainly at least 99% of the time this is a good rule of thumb.

### Resource clean-up in destructor

How should I handle resources if my constructors may throw exceptions?

Basically, every data member inside your object should clean up its own mess.

If a constructor throws an exception, the object’s destructor is not run. If your object has already done something that needs to be undone (such as allocating some memory, opening a file, or locking a semaphore), this “stuff that needs to be undone” must be remembered by a data member inside the object.

For example, rather than allocating memory into a raw Fred\* data member, put the allocated memory into a “smart pointer” member object, and the destructor of this smart pointer will delete the Fred object when the smart pointer dies. The template std::unique\_ptr is an example of such as “smart pointer.” You can also [write your own reference counting smart pointer](https://isocpp.org/wiki/faq/freestore-mgmt#ref-count-simple). You can also [use smart pointers to “point” to disk records or objects on other machines](https://isocpp.org/wiki/faq/operator-overloading#op-ov-examples).

In the follwong example, a member variable throws exception in constructor:

**#include** <iostream>

**#include** <exception>

**using** **namespace** std;

**class** X{

**public**:

**X**(){ cout << "X" << endl; }

**~X**(){ cout << "~X" << endl; }

};

**class** Y{

**public**:

**Y**(){ cout << "Y" << endl; }

**~Y**(){ cout << "~Y" << endl; }

};

**class** Z{

**public**:

**Z**(){ **throw** 20; }

**~Z**(){ cout << "~Z" << endl; }

};

**class** A{

X x;

Z z; // The constructor throws exception which caused terminal of code

Y y;

};

**int** **main**()

{

A a; // If put this in try-catch, ~X() will be called

}

The output will be:

X

libc++abi.dylib: terminating with uncaught exception of type int

Abort trap: 6

### Data Member Initialization

#### Item 13: List members in an initialization list in the order in which they are declared

template<class T>

class Array {

public:

Array(int lowBound, int highBound);

private:

vector<T> data; // the array data is stored in a vector object

size\_t size; // # of elements in array

int lBound, hBound; // lower bound, higher bound

};

template<class T>

Array<T>::Array(int lowBound, int highBound)

: size(highBound - lowBound + 1),

lBound(lowBound), hBound(highBound),

data(size)

{}

An industrial-strength constructor would perform sanity checking on its parameters to ensure that highBound was at least as great as lowBound, but there is a much nastier error here: even with perfectly good values for the array's bounds, you have absolutely no idea how many elements data holds.

"How can that be?" I hear you cry. "I carefully initialized size before passing it to the vector constructor!" Unfortunately, you didn't — you just tried to. The rules of the game are that class members are initialized *in the order of their declaration in the class*; the order in which they are listed in a **member initialization** list makes not a whit of difference. In the classes generated by your Array template, data will always be initialized first, followed by size, lBound, and hBound. Always.

(*My note: this looks like having some problem. The above code does not give compile warning as expected and print correct size when I tested. Still not sure why*)

**#include** <cstddef>

**#include** <iostream>

**#include** <vector>

**using** **namespace** std;

**class** C {

**int** a1;

**int** a2;

**public**:

**C**() : a2(10), a1(a2) {} // Compile warning: a2 is uninitialized

**void** **print**(){

cout << "a1=" << a1 << ", a2=" << a2 << endl;

}

};

**void** **test\_C**(){

C c;

c.print();

}

//==================================================

**class** X{

**private**:

**int** x = 0;

**public**:

**X**(){ cout << "X()" << endl; } // default constructor

**X**(**int** a) : x(a) { cout << "X(int a)" << endl; }

**friend** **class** A;

};

**class** A{

**private**:

X x; // default initializer

**int** a;

**public**:

**A**();

**~A**() { cout << "~A(): " << x.x << endl; }

};

**A::A**() : a(5), x(a) {} // Compile warning: a is uninitialized

**void** **test\_A**(){ A a; }

//==================================================

**template**<**class** **T**>

**class** Array {

**private**:

vector<**T**> data; // the array data is stored in a vector object

**int** lBound, hBound; // lower bound, higher bound

**public**:

**Array**(**int** lowBound, **int** highBound);

**int** **size**() { **return** data.size(); }

};

**template**<**class** **T**>

**Array<T>::Array**(**int** lowBound, **int** highBound)

: lBound(lowBound), hBound(highBound),

data(highBound - lowBound + 1) // Why no compile warning here?

{}

**void** **test\_Array**(){

Array<**int**> a(10, 20);

cout << a.size() << endl; // This always prints "11"

}

**int** **main**(){ test\_Array(); }

#### constructor initializer supersedes the default

**class** Y{

**private**:

**int** y = 0;

**public**:

**Y**() { cout << "Y()" << endl; }

**Y**(**int** a) : y(a) {

cout << "Y(int a)" << endl;

}

**friend** **class** B;

};

**class** B{

**private**:

Y y; // default initializer will NOT be used if in I-list

**public**:

**B**();

**~B**() { cout << "~B(): " << y.y << endl; }

};

**B::B**() : y(5) {}

**void** **test\_Y**(){ B b; }

**int** **main**(){ test\_Y(); }

### Item 14:  Make sure base classes have virtual destructors

Sometimes it's convenient for a class to keep track of how many objects of its type exist. The straightforward way to do this is to create a static class member for counting the objects. The member is initialized to 0, is incremented in the class constructors, and is decremented in the class destructor. ([Item M26](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI26_FR.HTM#5350) shows how to package this approach so it's easy to add to any class, and [my article on counting objects](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MAGAZINE\CO_FRAME.HTM) describes additional refinements to the technique.) [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp2) [Item E14, P2](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp2)

You might envision a military application, in which a class representing enemy targets might look something like this: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp3) [Item E14, P3](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp3)

class EnemyTarget {

private:

static size\_t numTargets; // object counter

public:

EnemyTarget() { ++numTargets; }

EnemyTarget(const EnemyTarget&) { ++numTargets; }

~EnemyTarget() { --numTargets; }

static size\_t numberOfTargets()

{ return numTargets; }

virtual bool destroy(); // returns success of

// attempt to destroy

// EnemyTarget object

};

// class statics must be defined outside the class;

// initialization is to 0 by default

size\_t EnemyTarget::numTargets;

This class is unlikely to win you a government defense contract, but it will suffice for our purposes here, which are substantially less demanding than are those of the Department of Defense. Or so we may hope. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp4) [Item E14, P4](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp4)

Let us suppose that a particular kind of enemy target is an enemy tank, which you model, naturally enough (see [Item 35](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI35_FR.HTM#6914), but also see [Item M33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI33_FR.HTM#10947)), as a publicly derived class of EnemyTarget. Because you are interested in the total number of enemy tanks as well as the total number of enemy targets, you'll pull the same trick with the derived class that you did with the base class: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp5) [Item E14, P5](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp5)

class EnemyTank: public EnemyTarget {

private:

static size\_t numTanks; // object counter for tanks

public:

EnemyTank() { ++numTanks; }

EnemyTank(const EnemyTank& rhs)

: EnemyTarget(rhs)

{ ++numTanks; }

~EnemyTank() { --numTanks; }

static size\_t numberOfTanks()

{ return numTanks; }

virtual bool destroy();

};

Having now added this code to two different classes, you may be in a better position to appreciate [Item M26](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI26_FR.HTM#5350)'s general solution to the problem. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp6) [Item E14, P6](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp6)

Finally, let's assume that somewhere in your application, you dynamically create an EnemyTank object using new, which you later get rid of via delete: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp7) [Item E14, P7](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp7)

EnemyTarget \*targetPtr = new EnemyTank;

...

delete targetPtr;

Everything you've done so far seems completely kosher. Both classes undo in the destructor what they did in the constructor, and there's certainly nothing wrong with your application, in which you were careful to use delete after you were done with the object you conjured up with new. Nevertheless, there is something very troubling here. Your program's behavior is *undefined* — you have no way of knowing what will happen. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp8) [Item E14, P8](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp8)

The **°**[C++ language standard](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=cstandard) is unusually clear on this topic. ***When you try to delete a derived class object through a base class pointer and the base class has a nonvirtual destructor (as EnemyTarget does), the results are undefined***. That means compilers may generate code to do whatever they like: reformat your disk, send suggestive mail to your boss, fax source code to your competitors, whatever. (What often happens at runtime is that the derived class's destructor is never called. In this example, that would mean your count of EnemyTanks would not be adjusted when targetPtr was deleted. Your count of enemy tanks would thus be wrong, a rather disturbing prospect to combatants dependent on accurate battlefield information.) [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp9) [Item E14, P9](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp9)

To avoid this problem, you have only to make the EnemyTarget destructor *virtual*. Declaring the destructor virtual ensures well-defined behavior that does precisely what you want: both EnemyTank's and EnemyTarget's destructors will be called before the memory holding the object is deallocated. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp10) [Item E14, P10](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp10)

Now, the EnemyTarget class contains a virtual function, which is generally the case with base classes. After all, the purpose of virtual functions is to allow customization of behavior in derived classes (see [Item 36](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI36_FR.HTM#7007)), so almost all base classes contain virtual functions. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp11) [Item E14, P11](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp11)

If a class does *not* contain any virtual functions, that is often an indication that it is not meant to be used as a base class. When a class is not intended to be used as a base class, making the destructor virtual is usually a bad idea. Consider this example, based on a discussion in the ARM (see [Item 50](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI50_FR.HTM#8569)): [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp12) [Item E14, P12](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp12)

// class for representing 2D points

class Point {

public:

Point(short int xCoord, short int yCoord);

~Point();

private:

short int x, y;

};

If a short int occupies 16 bits, a Point object can fit into a 32-bit register. Furthermore, a Point object can be passed as a 32-bit quantity to functions written in other languages such as C or FORTRAN. If Point's destructor is made virtual, however, the situation changes. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp13) [Item E14, P13](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp13) (In summary: it breaks the portability)

The implementation of virtual functions requires that objects carry around with them some additional information that can be used at runtime to determine which virtual functions should be invoked on the object. In most compilers, this extra information takes the form of a pointer called a vptr ("virtual table pointer"). The vptr points to an array of function pointers called a vtbl ("virtual table"); each class with virtual functions has an associated vtbl. When a virtual function is invoked on an object, the actual function called is determined by following the object's vptr to a vtbl and then looking up the appropriate function pointer in the vtbl. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp14) [Item E14, P14](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp14)

The details of how virtual functions are implemented are unimportant (though, if you're curious, you can read about them in [Item M24](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI24_FR.HTM#41284)). What *is* important is that if the Point class contains a virtual function, objects of that type will implicitly *double* in size, from two 16-bit shorts to two 16-bit shorts plus a 32-bit vptr! No longer will Point objects fit in a 32-bit register. Furthermore, Point objects in C++ no longer look like the same structure declared in another language such as C, because their foreign language counterparts will lack the vptr. As a result, it is no longer possible to pass Points to and from functions written in other languages unless you explicitly compensate for the vptr, which is itself an implementation detail and hence unportable. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp15) [Item E14, P15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp15)

The bottom line is that gratuitously declaring all destructors virtual is just as wrong as never declaring them virtual. In fact, many people summarize the situation this way: **declare a virtual destructor in a class if and only if that class contains at least one virtual function**. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp16) [Item E14, P16](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp16)

This is a good rule, one that works most of the time, but unfortunately, it is possible to get bitten by the nonvirtual destructor problem even in the absence of virtual functions. For example, [Item 13](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI13_FR.HTM#2117) considers a class template for implementing arrays with client-defined bounds. Suppose you decide (in spite of the advice in [Item M33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI33_FR.HTM#10947)) to write a template for derived classes representing named arrays, i.e., classes where every array has a name: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp17) [Item E14, P17](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp17)

template<class T> // base class template

class Array { // (from [Item 13](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI13_FR.HTM#2117))

public:

Array(int lowBound, int highBound);

~Array();

private:

vector<T> data;

size\_t size;

int lBound, hBound;

};

template<class T>

class NamedArray: public Array<T> {

public:

NamedArray(int lowBound, int highBound, const string& name);

...

private:

string arrayName;

};

If anywhere in an application you somehow convert a pointer-to-NamedArray into a pointer-to-Array and you then use delete on the Array pointer, you are instantly transported to the realm of undefined behavior: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp18) [Item E14, P18](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_DIR.HTM#dingp18)

NamedArray<int> \*pna =

new NamedArray<int>(10, 20, "Impending Doom");

Array<int> \*pa;

...

pa = pna; // NamedArray<int>\* -> Array<int>\*

...

delete pa; // undefined! (Insert theme to

//**°**[Twilight Zone here](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=twilight)); in practice,

// pa->arrayName will often be leaked,

// because the NamedArray part of

// \*pa will never be destroyed

This situation can arise more frequently than you might imagine, because it's not uncommon to want to take an existing class that does something, Array in this case, and derive from it a class that does all the same things, plus more. NamedArray doesn't redefine any of the behavior of Array — it inherits all its functions without change — it just adds some additional capabilities. Yet the nonvirtual destructor problem persists. (As do others. See [Item M33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI33_FR.HTM#10947).)

Finally, it's worth mentioning that it can be convenient to declare pure virtual destructors in some classes. Recall that pure virtual functions result in *abstract* classes — classes that can't be instantiated (i.e., you can't create objects of that type). Sometimes, however, you have a class that you'd like to be abstract, but you don't happen to have any functions that are pure virtual. What to do? Well, because an abstract class is intended to be used as a base class, and because a base class should have a virtual destructor, and because a pure virtual function yields an abstract class, the solution is simple: declare a pure virtual destructor in the class you want to be abstract.

Here's an example:

class AWOV { // AWOV = "Abstract w/o

// Virtuals"

public:

virtual ~AWOV() = 0; // declare pure virtual

// destructor

};

This class has a pure virtual function, so it's abstract, and it has a virtual destructor, so you can rest assured that you won't have to worry about the destructor problem. **There is one twist, however: you must provide a *definition* for the pure virtual destructor**:

AWOV::~AWOV() {} // definition of pure

// virtual destructor

You need this definition, because the way virtual destructors work is that the most derived class's destructor is called first, then the destructor of each base class is called. That means that compilers will generate a call to ~AWOV even though the class is abstract, so you have to be sure to provide a body for the function. If you don't, the linker will complain about a missing symbol, and you'll have to go back and add one.

You can do anything you like in that function, but, as in the example above, it's not uncommon to have nothing to do. If that is the case, you'll probably be tempted to avoid paying the overhead cost of a call to an empty function by declaring your destructor inline. That's a perfectly sensible strategy, but there's a twist you should know about.

Because your destructor is virtual, its address must be entered into the class's vtbl (see [Item M24](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI24_FR.HTM#41284)). But inline functions aren't supposed to exist as freestanding functions (that's what inline means, right?), so special measures must be taken to get addresses for them. [Item 33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI33_FR.HTM#6729) tells the full story, but the bottom line is this: if you declare a virtual destructor inline, you're likely to avoid function call overhead when it's invoked, but your compiler will still have to generate an out-of-line copy of the function somewhere, too.

### Private Constructor

#### Why?

* 1. Prevent inheritance;
  2. Special way to control the life cycle of its objects

#### How to use?

(<https://www.geeksforgeeks.org/can-constructor-private-cpp/>)

1. User friend class to delegate creation of its objects. This technology can be used in factory pattern and singleton pattern.
2. Named constructor idiom. <https://isocpp.org/wiki/faq/ctors>

A technique that provides more intuitive and/or safer construction operations for users of your class.

The problem is that constructors always have the same name as the class. Therefore the only way to differentiate between the various constructors of a class is by the parameter list. But if there are lots of constructors, the differences between them become somewhat subtle and error prone.

With the Named Constructor Idiom, you declare all the class’s constructors in the private or protected sections, and you provide public static methods that return an object. These static methods are the so-called “Named Constructors.” In general there is one such static method for each different way to construct an object.

class Point {

public:

Point(float x, float y); *// Rectangular coordinates*

Point(float r, float a); *// Polar coordinates (radius and angle)*

*// ERROR: Overload is Ambiguous: Point::Point(float,float)*

};

int main()

{

Point p = Point(5.7, 1.2); *// Ambiguous: Which coordinate system?*

}

One way to solve this ambiguity is to use the Named Constructor Idiom:

*#include* <cmath> *// To get std::sin() and std::cos()*

class Point {

public:

static Point rectangular(float x, float y); *// Rectangular coord's*

static Point polar(float radius, float angle); *// Polar coordinates*

*// These static methods are the so-called "named constructors"*

*// ...*

private:

Point(float x, float y); *// Rectangular coordinates*

float x\_, y\_;

};

inline Point::Point(float x, float y)

: x\_(x), y\_(y) { }

inline Point Point::rectangular(float x, float y)

{ return Point(x, y); }

inline Point Point::polar(float radius, float angle)

{ return Point(radius\*std::cos(angle), radius\*std::sin(angle)); }

Now the users of Point have a clear and unambiguous syntax for creating Points in either coordinate system:

int main()

{

Point p1 = Point::rectangular(5.7, 1.2); *// Obviously rectangular*

Point p2 = Point::polar(5.7, 1.2); *// Obviously polar*

*// ...*

}

Make sure your constructors are in the protected section if you expect Point to have derived classes.

The Named Constructor Idiom can also be used to [make sure your objects are always created via new](https://isocpp.org/wiki/faq/freestore-mgmt#static-create-methods).

### Private Destructor

<http://stackoverflow.com/questions/631783/what-is-the-use-of-having-destructor-as-private> Basically, any time you want some other class to be responsible for the life cycle of your class' objects, or you have reason to prevent the destruction of an object, you can make the destructor private.

For instance, if you're doing some sort of reference counting thing, you can have the object (or manager that has been "friend"ed) responsible for counting the number of references to itself and delete it when the number hits zero. A private dtor would prevent anybody else from deleting it when there were still references to it.

For another instance, what if you have an object that has a manager (or itself) that may destroy it or may decline to destroy it depending on other conditions in the program, such as a database connection being open or a file being written. You could have a "request\_delete" method in the class or the manager that will check that condition and either delete or or decline it, and return a status telling you what it did. That's far more flexible that just calling "delete".

<http://blogs.msdn.com/b/larryosterman/archive/2005/07/01/434684.aspx> The CLR IDisposable pattern comes in quite handy here. That allows the caller to notify the object that it's done with the object. Of course, it's also responsible for dealing with the consequences...

The bottom line is that once you decide to use reference counted objects, you don't control the lifetime of the object, all you do is control the lifetime of a reference to the object. And declaring the destructor private forces you to recognise this.

<http://stackoverflow.com/questions/2559750/private-destructor-for-singleton-class> All classes have a destructor. If you don't create one the compiler will do so for you. So your question can be reworded to: Does the destructor for a singleton class have to private?

The simple answer is no, it doesn't have to be.

A more interesting question: Is it a good idea to make the destructor of a singleton class private?

~~Yes, in general, it is a good idea. If you make it private then your client code won't call the destructor by accident. Calling the destructor would cause the singleton to fail for all clients as the instance would become invalid~~.

No, and in general objects in C++ are not given private destructors. Keep in mind that Singleton means that there is only one instance, and so it is construction, not destruction, that needs to be controlled / prevented. Usually a singleton has a private constructor, a public destructor, a private static instance variable, and a public static singleton get / lazy construction function, although there are variations on that pattern.

### Constructor vs operator=

If both copy constructor and operator= are defined, which will be called?

Rule: when initializing the object, copy constructor is called; if object is already initialized, operator= is called

main()

{ myClass first(100);

myClass second(first);

myClass third = first; //initialize “third”, copy constructor

myClass \*ptr = new myClass(first);

myClass \*p;

second = third; // second already initialized: operator=

p = new myClass(200); //Note here assignment is not executed!!!!

delete ptr;

delete p;

}

### Return Object Compiler Optimization: Does return-by-value mean extra copies and extra overhead?

<http://www.parashift.com/c++-faq-lite/ctors.html>

Not necessarily.

All(?) commercial-grade compilers **optimize away the extra copy**.

To keep the example clean, let's strip things down to the bare essentials. Suppose function caller() calls rbv() ("rbv" stands for "return by value") which returns a Foo object by value:

class Foo { ... };  
Foo rbv();  
void caller(){  
 Foo x = rbv(); *← the return-value of rbv() goes into x*

*...*  
}

Now the question is, How many Foo objects will there be? Will rbv() create a temporary Foo object that gets copy-constructed into x? How many temporaries? Said another way, does return-by-value necessarily degrade performance?

The point of this FAQ is that the answer is No, commercial-grade C++ compilers implement return-by-value in a way that lets them eliminate the overhead, at least in simple cases like those shown in the previous FAQ. In particular, all(?) commercial-grade C++ compilers will optimize this case:

Foo rbv()  
{  
 *...*  
 return Foo(42, 73); *← suppose Foo has a ctor Foo::Foo(int a, int b)*  
}

Certainly the compiler is *allowed* to create a temporary, local Foo object, then copy-construct that temporary into variable x within caller(), then destruct the temporary. But all(?) commercial-grade C++ compilers won't do that: the return statement will directly construct x itself. Not a copy of x, not a pointer to x, not a reference to x, but x itself.

You can stop here if you don't want to genuinely understand the previous paragraph, but if you want to know the secret sauce (so you can, for example, reliably predict when the compiler can and cannot provide that optimization for you), the key is to know that compilers usually implement return-by-value using pass-by-pointer. When caller() calls rbv(), the compiler secretly passes a pointer to the location where rbv() is supposed to construct the "returned" object. It might look something like this (it's shown as a void\* rather than a Foo\* since the Foo object has not yet been constructed):

*// Pseudo-code*  
void rbv(void\* put\_result\_here) *← Original C++ code: Foo rbv()*  
{  
   ...    *← rbv() initializes (not assigns to) the variable pointed to by put\_result\_here*  
}  
void caller()  
{  
   *//Original C++ code: Foo x = rbv()*  
   struct Foo x; *← Note: x does not get initialized prior to calling rbv()*  
   rbv(&x);      *← Note: rbv() initializes a local variable defined in caller()*  
   *...*  
}

**So the first ingredient in the secret sauce is that the compiler (usually) transforms return-by-value into pass-by-pointer**. This means that commercial-grade compilers don't bother creating a temporary: they directly construct the returned object in the location pointed to by put\_result\_here.

**The second ingredient in the secret sauce is that compilers typically implement constructors using a similar technique**. This is compiler-dependent and somewhat idealized (I'm intentionally ignoring how to handle new and overloading), but compilers typically implement Foo::Foo(int a, int b) using something like this:

*// Pseudo-code*  
 void Foo\_ctor(Foo\* this, int a, int b) *← Original C++ code: Foo::Foo(int a, int b)*  
 {  
   *...*  
 }

Putting these together, the compiler might implement the return statement in rbv() by simply passing put\_result\_here as the constructor's this pointer:

*// Pseudo-code*  
 void rbv(void\* put\_result\_here) *← Original C++ code: Foo rbv()*  
 {  
   *...*  
   Foo\_ctor((Foo\*)put\_result\_here, 42, 73); *← Original C++ code: return Foo(42,73);*  
   return;  
 }

So caller() passes &x to rbv(), and rbv() in turn passes &x to the constructor (as the this pointer). That means constructor *directly* constructs x.

In the early 90s I did a seminar for IBM's compiler group in Toronto, and one of their engineers told me that they found this return-by-value optimization to be so fast that you get it even if you don't compile with optimization turned on. Because the return-by-value optimization causes the compiler to generate less code, it actually improves compile-times in addition to making your generated code smaller and faster. The point is that the return-by-value optimization is almost universally implemented, at least in code cases like those shown above.

Final thought: this discussion was limited to whether there will be any extra copies of the returned object in a return-by-value call. Don't confuse that with other things that could happen in caller(). For example, if you changed caller() from Foo x = rbv(); to Foo x; x = rbv(); (note the ; after the declaration), the compiler is required to use Foo's assignment operator, and unless the compiler can prove that Foo's default constructor followed by assignment operator is exactly the same as its copy constructor, the compiler is required by the language to put the returned object into an unnamed temporary within caller(), use the assignment operator to copy the temporary into x, then destruct the temporary. The return-by-value optimization still plays its part since there will be only one temporary, but by changing Foo x = rbv(); to Foo x; x = rbv();, you have prevented the compiler from eliminating that last temporary.

### Deletaging constructors

In C++98, if you want two constructors to do the same thing, repeat yourself or call “an init() function.” For example:

class X {

int a;

void validate(int x) { if (0<x && x<=max) a=x; else throw bad\_X(x); }

public:

X(int x) { validate(x); }

X() { validate(42); }

X(string s) { int x = lexical\_cast<int>(s); validate(x); }

*// ...*

};

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. So, in C++11, we can define one constructor in terms of another:

class X {

int a;

public:

X(int x) { if (0<x && x<=max) a=x; else throw bad\_X(x); }

X() :X{42} { }

X(string s) :X{lexical\_cast<int>(s)} { }

*// ...*

};

### Empty class

Size of an emptuy class is not zero. It is 1 byte generally.

It is nonzero to ensure that the two different objects will have different addresses, whether instantiated using or not using “new” operator.

**#include** <iostream>

**using** **namespace** std;

**class** Empty {};

**class** A : **public** Empty {**int** a;};

**class** D : **public** Empty {};

**class** Derived1 : **public** Empty {};

**class** Derived2 : **virtual** **public** Empty {};

**class** Derived3 : **public** Empty {

**char** c;

};

**class** Derived4 : **virtual** **public** Empty {

**char** c;

};

**class** Dummy {

**char** c;

};

**int** **main**(){

cout << "sizeof(Empty) " << **sizeof**(Empty) << endl; // 1

cout << "sizeof(Derived1) " << **sizeof**(Derived1) << endl; // 1

cout << "sizeof(Derived2) " << **sizeof**(Derived2) << endl; // 8

cout << "sizeof(Derived3) " << **sizeof**(Derived3) << endl; // 1

cout << "sizeof(Derived4) " << **sizeof**(Derived4) << endl; // 16

cout << "sizeof(Dummy) " << **sizeof**(Dummy) << endl; // 1

cout << **sizeof**(A) << endl; // 4

cout << **sizeof**(D) << endl; // 1

Empty a, b;

cout << "&a == &b value: " << (&a == &b) << endl; // 0

cout << ((**new** Empty) == (**new** Empty)) << endl; // 0

}

### this pointer in the constructor?

(<https://isocpp.org/wiki/faq/ctors#using-this-in-ctors>) Some people feel you should not use the this pointer in a constructor because the object is not fully formed yet. However you can use this in the constructor (in the {body} and even in the [initialization list](https://isocpp.org/wiki/faq/ctors#init-lists)) if you are careful.

**Here is something that always works: the {body} of a constructor (or a function called from the constructor) can reliably access the data members declared in a base class and/or the data members declared in the constructor’s own class. This is because all those data members are guaranteed to have been fully constructed by the time the constructor’s {body} starts executing.**

Here is something that never works: the {body} of a constructor (or a function called from the constructor) cannot get down to a derived class by calling a virtual member function that is overridden in the derived class. If your goal was to get to the overridden function in the derived class, [you won’t get what you want](https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors). Note that you won’t get to the override in the derived class independent of how you call the virtual member function: explicitly using the this pointer (e.g., this->method()), implicitly using the this pointer (e.g., method()), or even calling some other function that calls the virtual member function on your this object. The bottom line is this: even if the caller is constructing an object of a derived class, during the constructor of the base class, [your object is not yet of that derived class](https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors). You have been warned.

Here is something that sometimes works: if you pass any of the data members in this object to another data member’s [initializer](https://isocpp.org/wiki/faq/ctors#init-lists), you must make sure that the other data member has already been initialized. The good news is that you can determine whether the other data member has (or has not) been initialized using some straightforward language rules that are independent of the particular compiler you’re using. The bad news is that you have to know those language rules (e.g., base class sub-objects are initialized first (look up the order if you have multiple and/or virtual inheritance!), then data members defined in the class are initialized in the order in which they appear in the class declaration). If you don’t know these rules, then don’t pass any data member from the this object (regardless of whether or not you explicitly use the this keyword) to any other data member’s [initializer](https://isocpp.org/wiki/faq/ctors#init-lists)! And if you do know the rules, please be careful.

## About operator=

### Summary

1. Return a reference to \**this*, parameter const reference.

C& C::operator=(const C&);

1. Self-assignment

if (this == &rhs) return \*this

1. Call base class assignment if this is a derived class

Base::operator=(rhs); or

static\_cast<Base&>(\*this) = rhs;

### Item 15:  Have operator= return a reference to \*this

For the default version of operator= in a class C, the signature of the function is as follows (see [Item 45](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_FR.HTM#8160)):

C& C::operator=(const C&);

You'll almost always want to follow this convention of having operator= both take and return a reference to a class object, although at times you may overload operator= so that it takes different argument types.

A common error amongst new C++ programmers is to have operator= return void, a decision that seems reasonable until you realize it prevents chains of assignment. So don't do it.

Another common error is to have operator= return a reference to a const object, like this:

class Widget {

public:

... // note

const Widget& operator=(const Widget& rhs); // const

... // return

}; // type

The usual motivation is to prevent clients from doing silly things like this:

Widget w1, w2, w3;

...

(w1 = w2) = w3; // assign w2 to w1, then w3 to

// the result! (Giving Widget's

// operator= a const return value

// prevents this from compiling.)

Silly this may be, but not so silly that it's prohibited for the built-in types:

int i1, i2, i3;

...

(i1 = i2) = i3; // legal! assigns i2 to

// i1, then i3 to i1!

### Item 16:  Assign to all data members in operator=.

[Item 45](I:\\Dropbox\\~Family\\z-lei\\homepage\\1_Effective C++ 2nd\\EC\\EI45_FR.HTM" \l "8160" \t "_top) explains that C++ will write an assignment operator for you if you don't declare one yourself, and [Item 11](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI11_FR.HTM#2042) describes why you often won't much care for the one it writes for you, so perhaps you're wondering if you can somehow have the best of both worlds, whereby you let C++ generate a default assignment operator and you selectively override those parts you don't like. No such luck. If you want to take control of any part of the assignment process, you must do the entire thing yourself. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp2) [Item E16, P2](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp2)

In practice, this means that you need to assign to *every* data member of your object when you write your assignment operator(s): [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp3) [Item E16, P3](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp3)

template<class T> // template for classes associating

class NamedPtr { // names with pointers (from [Item 12](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_FR.HTM#2071))

public:

NamedPtr(const string& initName, T \*initPtr);

NamedPtr& operator=(const NamedPtr& rhs);

private:

string name;

T \*ptr;

};

template<class T>

NamedPtr<T>& NamedPtr<T>::operator=(const NamedPtr<T>& rhs)

{

if (this == &rhs)

return \*this; // see [Item 17](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI17_FR.HTM#2264)

// assign to all data members

name = rhs.name; // assign to name

\*ptr = \*rhs.ptr; // for ptr, assign what's

// pointed to, not the

// pointer itself

return \*this; // see [Item 15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI15_FR.HTM#2182)

}

This is easy enough to remember when the class is originally written, but it's equally important that the assignment operator(s) be updated if new data members are added to the class. For example, if you decide to upgrade the NamedPtr template to carry a timestamp marking when the name was last changed, you'll have to add a new data member, and this will require updating the constructor(s) as well as the assignment operator(s). In the hustle and bustle of upgrading a class and adding new member functions, etc., it's easy to let this kind of thing slip your mind. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp4) [Item E16, P4](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp4)

**The real fun begins when inheritance joins the party, because a derived class's assignment operator(s) must also handle assignment of its base class members**! Consider this: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp5) [Item E16, P5](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp5)

class Base {

public:

Base(int initialValue = 0): x(initialValue) {}

private:

int x;

};

class Derived: public Base {

public:

Derived(int initialValue)

: Base(initialValue), y(initialValue) {}

Derived& operator=(const Derived& rhs);

private:

int y;

};

The logical way to write Derived's assignment operator is like this: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp6) [Item E16, P6](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp6)

// erroneous assignment operator

Derived& Derived::operator=(const Derived& rhs)

{

if (this == &rhs) return \*this; // see [Item 17](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI17_FR.HTM#2264)

y = rhs.y; // assign to Derived's

// lone data member

return \*this; // see [Item 15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI15_FR.HTM#2182)

}

Unfortunately, this is incorrect, because the data member x in the Base part of a Derived object is unaffected by this assignment operator. For example, consider this code fragment: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp7) [Item E16, P7](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp7)

void assignmentTester()

{

Derived d1(0); // d1.x = 0, d1.y = 0

Derived d2(1); // d2.x = 1, d2.y = 1

d1 = d2; // d1.x = 0, d1.y = 1!

}

Notice how the Base part of d1 is unchanged by the assignment. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp8) [Item E16, P8](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp8)

The straightforward way to fix this problem would be to make an assignment to x in Derived::operator=. Unfortunately, that's not legal, because x is a private member of Base. Instead, you have to make an explicit assignment to the Base *part* of Derived from inside Derived's assignment operator. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp9) [Item E16, P9](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp9)

This is how you do it: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp10) [Item E16, P10](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp10)

// correct assignment operator

Derived& Derived::operator=(const Derived& rhs)

{

if (this == &rhs) return \*this;

Base::operator=(rhs); // call this->Base::operator=

y = rhs.y;

return \*this;

}

Here you just make an explicit call to Base::operator=. That call, like all calls to member functions from within other member functions, will use \*this as its implicit left-hand object. The result will be that Base::operator= will do whatever work it does on the Base part of \*this — precisely the effect you want. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp11) [Item E16, P11](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp11)

Alas, some compilers (incorrectly) reject this kind of call to a base class's assignment operator if that assignment operator was generated by the compiler (see [Item 45](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_FR.HTM#8160)). To pacify these renegade translators, you need to implement Derived::operator= this way:

Derived& Derived::operator=(const Derived& rhs)

{

if (this == &rhs) return \*this;

static\_cast<Base&>(\*this) = rhs; // call operator= on

// Base part of \*this

y = rhs.y;

return \*this;

}

This monstrosity casts \*this to be a reference to a Base, then makes an assignment to the result of the cast. That makes an assignment to only the Base part of the Derived object. Careful now! **It is important that the cast be to a *reference* to a Base object, not to a Base object itself**. If you cast \*this to be a Base object, you'll end up calling the copy constructor for Base, and the new object you construct (see [Item M19](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI19_FR.HTM#41177)) will be the target of the assignment; \*this will remain unchanged. Hardly what you want.

Regardless of which of these approaches you employ, once you've assigned the Base part of the Derived object, you then continue with Derived's assignment operator, making assignments to all the data members of Derived. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp14) [Item E16, P14](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp14)

A similar inheritance-related problem often arises when implementing derived class copy constructors. Take a look at the following, which is the copy constructor analogue of the code we just examined: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp15) [Item E16, P15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp15)

class Base {

public:

Base(int initialValue = 0): x(initialValue) {}

Base(const Base& rhs): x(rhs.x) {}

private:

int x;

};

class Derived: public Base {

public:

Derived(int initialValue)

: Base(initialValue), y(initialValue) {}

Derived(const Derived& rhs) // erroneous copy

: y(rhs.y) {} // constructor

private:

int y;

};

Class Derived demonstrates one of the nastiest bugs in all C++-dom: it fails to copy the base class part when a Derived object is copy constructed. Of course, the Base part of such a Derived object is constructed, but it's constructed using Base's *default* constructor. Its member x is initialized to 0 (the default constructor's default parameter value), regardless of the value of x in the object being copied! [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp16) [Item E16, P16](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp16)

To avoid this problem, Derived's copy constructor must make sure that Base's copy constructor is invoked instead of Base's default constructor. That's easily done. Just be sure to specify an initializer value for Base in the member initialization list of Derived's copy constructor: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp17) [Item E16, P17](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_DIR.HTM#dingp17)

class Derived: public Base {

public:

Derived(const Derived& rhs): **Base(rhs)**, y(rhs.y) {}

...

};

Now when a client creates a Derived by copying an existing object of that type, its Base part will be copied, too.

### Item 17:  Check for assignment to self in operator=[Item E17, P1](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI17_DIR.HTM#dingp1)

Using address equality, a general assignment operator looks like this:  ¤[Item E17, P18](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI17_DIR.HTM#dingp18)

C& C::operator=(const C& rhs)

{

// check for assignment to self

if (this == &rhs) return \*this;

...

}

This suffices for a great many programs.

The problems of aliasing and object identity are hardly confined to operator=. That's just a function in which you are particularly likely to run into them. In the presence of references and pointers, any two names for objects of compatible types may in fact refer to the same object. Here are some other situations in which aliasing can show its Medusa-like visage:  ¤[Item E17, P22](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI17_DIR.HTM#dingp22)

class Base {

void mf1(Base& rb); // rb and \*this could be

// the same

...

};

void f1(Base& rb1,Base& rb2); // rb1 and rb2 could be

// the same

class Derived: public Base {

void mf2(Base& rb); // rb and \*this could be

// the same

...

};

int f2(Derived& rd, Base& rb); // rd and rb could be

// the same

## Inheritance

### Virtual Base Classes (10.5c)

“Except that it results in a unique object in its derived classes, a virtual base class behaves the save way a nonvirtual base class does”.

#### Diamond virtual functions

|  |  |
| --- | --- |
| using namespace std;  class Base {  public:  Base() { cout << "Base" << endl; }  };    class Der1 : public **virtual** Base {  public:  Der1() { cout << "Der1" << endl; }  };    class Der2 : public **virtual** Base {  public:  Der2() { cout << "Der2" << endl; }  };    class Join : public Der1, public Der2 {  public:  Join() { cout << "Join" << endl; }  };    int main(){  Join j;  } | using namespace std;  class Base {  public:  Base() { cout << "Base" << endl; }  };    class Der1 : public **virtual** Base {  public:  Der1() { cout << "Der1" << endl; }  };    class Der2 : public Base {  public:  Der2() { cout << "Der2" << endl; }  };    class Join : public Der1, public Der2 {  public:  Join() { cout << "Join" << endl; }  };    int main(){  Join j;  } |
| Base  Der1  Der2  Join | Base  Der1  Base  Der2  Join |

|  |  |
| --- | --- |
| using namespace std;  class Base {  public:  Base() { cout << "Base" << endl; }  };    class Der1 : public Base {  public:  Der1() { cout << "Der1" << endl; }  };    class Der2 : public **virtual** Base {  public:  Der2() { cout << "Der2" << endl; }  };    class Join : public Der1, public Der2 {  public:  Join() { cout << "Join" << endl; }  };    int main(){  Join j;  } | using namespace std;  class Base {  public:  Base() { cout << "Base" << endl; }  };    class Der1 : public Base {  public:  Der1() { cout << "Der1" << endl; }  };    class Der2 : public Base {  public:  Der2() { cout << "Der2" << endl; }  };    class Join : public Der1, public Der2 {  public:  Join() { cout << "Join" << endl; }  };    int main(){  Join j;  } |
| Base  Base  Der1  Der2  Join | Base  Der1  Base  Der2  Join |

### Cross Delegation – Delegate to Sister

<http://www.parashift.com/c++-faq-lite/multiple-inheritance.html>

**class** Base {

**public**:

**virtual** **void** **foo**() = 0;

**virtual** **void** **bar**() = 0;

};

**class** Der1: **public** **virtual** Base {

**public**:

**void** **foo**(){ bar(); }

};

**class** Der2: **public** **virtual** Base {

**public**:

**void** **bar**(){ cout << "Der2::bar" << endl;}

};

**class** Join: **public** Der1, **public** Der2 {};

**int** **main**()

{

Join\* p1 = **new** Join();

Der1\* p2 = p1;

Base\* p3 = p1;

p1->foo(); // print out: Der2::bar

p2->foo(); // print out: Der2::bar

p3->foo(); // print out: Der2::bar

}

Believe it or not, when Der1::foo() calls this->bar(), it ends up calling Der2::bar(). Yes, that's right: a class that Der1 knows nothing about will supply the override of a virtual function invoked by Der1::foo(). This "cross delegation" can be a powerful technique for customizing the behavior of polymorphic classes.

### Inherit private data members

<http://stackoverflow.com/questions/2676443/inheriting-private-members-in-c>

A derived class doesn't inherit *access* to private data members. However, it does inherit a full parent object, including all private members of parent class.

One other point is, it must inherit private members because some protected or public method may need to access the private data members.

This can be verified by using sizeof() operator.

### More than 1 address

<https://stackoverflow.com/questions/14776469/more-than-1-address-for-derived-class-object> In Item 27 of "Effective C++" (3rd Edition, Page 118), Scott Meyers said:

class Base { ... };

class Derived: public Base { ... };

Derived d;

Base \*pb = &d;

Here we're just creating a base class pointer to a derived class object, but sometimes, the two pointers will not be the same. When that's the case, an offset is applied at runtime to the Derived\* pointer to get the correct Base\* pointer value.

This last example demonstrates that a single object (e.g., an object of type Derived) might have more than one address (e.g., its address when pointed to by a Base\* pointer and its address when pointed to by a Derived\* pointer).

Here is a bit hard to understand. I know that a pointer to the base class can point to an object of the derived class at runtime, this is called polymorphism or dynamic binding. But does the derived class object really have more than 1 address in the memory?

class B1{ int i; };

class B2{ int i; };

class D : public B1, public B2 { int i; };

int main() {

D aD;

std::cout << &aD << std::endl;

std::cout << static\_cast<B1\*>( &aD ) << std::endl;

std::cout << static\_cast<B2\*>( &aD ) << std::endl;

return 0;

}

There's no possible way for the B1 sub-object to have the same address as the B2 sub-object.

## **Item 45:  Functions C++ silently writes and calls.**

When is an empty class not an empty class? When C++ gets through with it. If you don't declare them yourself, your thoughtful compilers will declare their own versions of a copy constructor, an assignment operator, a destructor, and a pair of address-of operators. Furthermore, if you don't declare any constructors, they will declare a default constructor for you, too. All these functions will be public. In other words, if you write this:

class Empty{};

it's the same as if you'd written this:[Item E45, P3](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp3)

class Empty {

**public**:

Empty(); // **default constructor**

Empty(const Empty& rhs); // **copy constructor**

~Empty(); // **destructor**

Empty& operator=(const Empty& rhs); // **assignment operator**

Empty\* operator&(); // **address-of operators**

const Empty\* operator&() const; // **address-of operators**

};

Now these functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:  ¤[Item E45, P4](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp4)

const Empty e1; // default constructor;

// destructor

Empty e2(e1); // copy constructor

e2 = e1; // assignment operator

Empty \*pe2 = &e2; // address-of

// operator (non-const)

const Empty \*pe1 = &e1; // address-of

// operator (const)

Given that compilers are writing functions for you, what do the functions do? Well, the default constructor and the destructor don't really do anything. They just enable you to create and destroy objects of the class. (They also provide a convenient place for implementers to place code whose execution takes care of "behind the scenes" behavior — see Items [33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI33_FR.HTM#6729) and [M24](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI24_FR.HTM#41284).) **Note that the generated destructor is nonvirtual** (see [Item 14](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI14_FR.HTM#223029)) unless it's for a class inheriting from a base class that itself declares a virtual destructor. The default address-of operators just return the address of the object. These functions are effectively defined like this:  ¤[Item E45, P5](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp5)

inline Empty::Empty() {}

inline Empty::~Empty() {}

inline Empty \* Empty::operator&() { return this; }

inline const Empty \* Empty::operator&() const

{ return this; }

As for the copy constructor and the assignment operator, the official rule is this: the default copy constructor (assignment operator) performs **memberwise copy** construction (assignment) of the **nonstatic** data members of the class. That is, if m is a nonstatic data member of type T in a class C and C declares no copy constructor (assignment operator), m will be copy constructed (assigned) using the copy constructor (assignment operator) defined for T, if there is one. If there isn't, this rule will be recursively applied to m's data members until a copy constructor (assignment operator) or built-in type (e.g., int, double, pointer, etc.) is found. By default, objects of built-in types are copy constructed (assigned) using bitwise copy from the source object to the destination object. For classes that inherit from other classes, this rule is applied to each level of the inheritance hierarchy, so user-defined copy constructors and assignment operators are called at whatever level they are declared.[Item E45, P6](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp6)

I hope that's crystal clear.[Item E45, P7](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp7)

But just in case it's not, here's an example. Consider the definition of a NamedObject template, whose instances are classes allowing you to associate names with objects:

template<class T>

class NamedObject {

public:

NamedObject(const char \*name, const T& value);

NamedObject(const string& name, const T& value);

...

private:

string nameValue;

T objectValue;

};

Because the NamedObject classes declare at least one constructor, compilers won't generate default constructors, but because the classes fail to declare copy constructors or assignment operators, compilers will generate those functions (if they are needed).

Consider the following call to a copy constructor:[Item E45, P10](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp10)

NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1); // calls copy constructor

The copy constructor generated by your compilers must initialize no2.nameValue and no2.objectValue using no1.nameValue and no1.objectValue, respectively. The type of nameValue is string, and string has a copy constructor (which you can verify by examining string in the standard library — see [Item 49](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI49_FR.HTM#8392)), so no2.nameValue will be initialized by calling the string copy constructor with no1.nameValue as its argument. On the other hand, the type of NamedObject<int>::objectValue is int (because T is int for this template instantiation), and no copy constructor is defined for ints, so no2.objectValue will be initialized by copying the bits over from no1.objectValue.

The compiler-generated assignment operator for NamedObject<int> would behave the same way, but in general, compiler-generated assignment operators behave as I've described only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will refuse to generate an operator= for your class, and you'll receive some lovely diagnostic during compilation.

For example, suppose NamedObject were defined like this, where nameValue is a *reference* to a string and objectValue is a *const* T:

template<class T>

class NamedObject {

public:

// this ctor no longer takes a const name, because name-

// Value is now a reference-to-non-const string. The char\*

// ctor is gone, because we must have a string to refer to

NamedObject(string& name, const T& value);

... // as above, assume no

// operator= is declared

private:

**string& nameValue**; // this is now a reference

const T objectValue; // this is now const

};

Now consider what should happen here:[Item E45, P14](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp14)

string newDog("Persephone");

string oldDog("Satch");

NamedObject<int> p(newDog, 2); // as I write this, our dog

// **°**[Persephone](http://www.awl.com/cseng/cgi-bin/cdquery.pl?name=pers) is about to

// have her second birthday

NamedObject<int> s(oldDog, 29); // the family dog Satch

// (from my childhood)

// would be 29 if she were

// still alive

p = s; // what should happen to

// the data members in p?

Before the assignment, p.nameValue refers to some string object and s.nameValue also refers to a string, though not the same one. How should the assignment affect p.nameValue? After the assignment, should p.nameValue refer to the string referred to by s.nameValue, i.e., should the reference itself be modified? If so, that breaks new ground, because C++ doesn't provide a way to make a reference refer to a different object (see [Item M1](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI1_FR.HTM#11029)). Alternatively, should the string object to which p.nameValue refers be modified, thus affecting other objects that hold pointers or references to that string, i.e., objects not directly involved in the assignment? Is that what the compiler-generated assignment operator should do?[Item E45, P15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp15)

Faced with such a conundrum, C++ refuses to compile the code.

If you want to support assignment in a class containing a reference member, you must define the assignment operator yourself.

Compilers behave similarly for classes containing const members (such as objectValue in the modified class above); it's not legal to modify const members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers refuse to generate assignment operators for derived classes that inherit from base classes declaring the standard assignment operator private. After all, compiler-generated assignment operators for derived classes are supposed to handle base class parts, too (see Items [16](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI16_FR.HTM#2225) and [M33](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI33_FR.HTM#10947)), but in doing so, they certainly shouldn't invoke member functions the derived class has no right to call.  ¤[Item E45, P16](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp16)

All this talk of compiler-generated functions gives rise to the question, what do you do if you want to disallow use of those functions? That is, what if you deliberately don't declare, for example, an operator= because you never *ever* want to allow assignment of objects in your class? The solution to that little teaser is the subject of [Item 27](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI27_FR.HTM#6406). For a discussion of the often-overlooked interactions between pointer members and compiler-generated copy constructors and assignment operators, check out [Item 11](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI11_FR.HTM#2042).  ¤[Item E45, P17](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI45_DIR.HTM#dingp17)

## Item 12:  Prefer initialization to assignment in constructors.

When you write the NamedPtr constructor, you have to transfer the values of the parameters to the corresponding data members. There are two ways to do this. The first is to use the member initialization list: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp4) [Item E12, P4](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp4)

template<class T>

NamedPtr<T>::NamedPtr(const string& initName, T \*initPtr )

: name(initName), ptr(initPtr)

{}

The second is to make assignments in the constructor body: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp5) [Item E12, P5](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp5)

template<class T>

NamedPtr<T>::NamedPtr(const string& initName, T \*initPtr)

{

name = initName;

ptr = initPtr;

}

There are important differences between these two approaches. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp6) [Item E12, P6](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp6)

From a purely pragmatic point of view, there are times when the initialization list *must* be used. In particular, const and reference members may *only* be initialized, never assigned. So, if you decided that a NamedPtr<T> object could never change its name or its pointer, you might follow the advice of [Item 21](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI21_FR.HTM#6003) and declare the members const: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp7) [Item E12, P7](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI12_DIR.HTM#dingp7)

## Shading

If the base class and the derived class both have a function with the same name, no matter same signature or not:

* 1. If calling with *baseObject.function*(), the function defined in base class is called. Here the derived class function is irrelevant
  2. If the recipient object is a derived class object, **the derived class member function shadows out the base class function of the same name**.
  3. This shadowing can be overridden by using the scope resolution operator ::
  4. It is not a form of overloading. If a base class function and a derived class function have the same name but different signature, the base class function cannot be forced to execute for a recipient derived object by using base class function signature
  5. java does not shadow base class functions.

Example:

#include <iostream>

class B{

public:

void foo(int){

std::cout << "B::foo" << std::endl;

}

};

class D:public B{

public:

void foo(int, char){

std::cout << "D::foo" << std::endl;

}

};

int main()

{

D obj;

// obj.foo(10); // error: no matching function

obj.B::foo(10);

}

## Placement new

<https://isocpp.org/wiki/faq/dtors#placement-new>

There are many uses of placement new. The simplest use is to place an object at a particular location in memory. This is done by supplying the place as a pointer parameter to the new part of a new expression:

#include <new> // Must #include this to use "placement new"

#include "Fred.h" // Declaration of class Fred

void someCode()

{

char memory[sizeof(Fred)]; // Line #1

void\* place = memory; // Line #2

Fred\* f = new(place) Fred(); // Line #3 (see "DANGER" below)

// The pointers f and place will be equal

// ...

}

Line #1 creates an array of sizeof(Fred) bytes of memory, which is big enough to hold a Fred object. Line #2 creates a pointer place that points to the first byte of this memory (experienced C programmers will note that this step was unnecessary; it’s there only to make the code more obvious). Line #3 essentially just calls the constructor Fred::Fred(). The this pointer in the Fred constructor will be equal to place. The returned pointer f will therefore be equal to place.

ADVICE: Don’t use this “placement new” syntax unless you have to. Use it only when you really care that an object is placed at a particular location in memory. For example, when your hardware has a memory-mapped I/O timer device, and you want to place a Clock object at that memory location.

DANGER: You are taking sole responsibility that the pointer you pass to the “placement new” operator points to a region of memory that is big enough and is properly aligned for the object type that you’re creating. Neither the compiler nor the run-time system make any attempt to check whether you did this right. If your Fred class needs to be aligned on a 4 byte boundary but you supplied a location that isn’t properly aligned, you can have a serious disaster on your hands (if you don’t know what “alignment” means, please don’t use the placement new syntax). You have been warned.

You are also solely responsible for destructing the placed object. This is done by **explicitly calling the destructor**:

void someCode()

{

char memory[sizeof(Fred)];

void\* p = memory;

Fred\* f = new(p) Fred();

// ...

f->~Fred(); // Explicitly call the destructor for the placed object

}

This is about the only time you ever explicitly call a destructor.

## new vs malloc

<https://isocpp.org/wiki/faq/c#c-diffs>

In C, you can implicitly convert a void\* to a T\*. This is unsafe. Consider:

#include<stdio.h>

int main()

{

char i = 0;

char j = 0;

char\* p = &i;

void\* q = p;

int\* pp = q; /\* unsafe, legal C, not C++ \*/

printf("%d %d\n",i,j);

\*pp = -1; /\* overwrite memory starting at &i \*/

printf("%d %d\n",i,j);

}

The effects of using a T\* that doesn’t point to a T can be disastrous. Consequently, in C++, to get a T\* from a void\* you need an explicit cast. For example, to get the undesirable effects of the program above, you have to write:

int\* pp = (int\*)q;

or, using a new style cast to make the unchecked type conversion operation more visible:

int\* pp = static\_cast<int\*>(q);

Casts are best avoided.

One of the most common uses of this unsafe conversion in C is to assign the result of malloc() to a suitable pointer. For example:

int\* p = malloc(sizeof(int));

In C++, use the typesafe new operator:

int\* p = new int;

Incidentally, the new operator offers additional advantages over malloc():

* new can’t accidentally allocate the wrong amount of memory,
* new implicitly checks for memory exhaustion, and
* new provides for initialization

For example:

typedef std::complex<double> cmplx;

/\* C style: \*/

cmplx\* p = (cmplx\*)malloc(sizeof(int)); // error: wrong size

// forgot to test for p==0

if (\*p == 7) { /\* ... \*/ } // oops: forgot to initialize \*p

// C++ style:

cmplx\* q = new cmplx(1,2); // will throw bad\_alloc if memory is exhausted

if (\*q == 7) { /\* ... \*/ }

note: ***calloc*** is like ***malloc*** but initializes the allocated memory with a constant (0). It needs to be freed with free

#### nothrow constant

<http://www.cplusplus.com/reference/new/nothrow/>

This constant value is used as an argument for [operator new](http://www.cplusplus.com/operator%20new) and [operator new[]](http://www.cplusplus.com/operator%20new%5b%5d) to indicate that these functions shall not throw an exception on failure, but return a *null pointer* instead.  
  
By default, when the new operator is used to attempt to allocate memory and the handling function is unable to do so, a[bad\_alloc](http://www.cplusplus.com/bad_alloc) exception is thrown. But when nothrow is used as argument for new, it returns a *null pointer* instead.  
  
This constant (nothrow) is just a value of type [nothrow\_t](http://www.cplusplus.com/nothrow_t), with the only purpose of triggering an overloaded version of the function [operator new](http://www.cplusplus.com/operator%20new) (or [operator new[]](http://www.cplusplus.com/operator%20new%5b%5d)) that takes an argument of this type.  
  
In C++, the operator new function can be overloaded to take more than one parameter: The first parameter passed to the [operator new](http://www.cplusplus.com/operator%20new) function is always the size of the storage to be allocated, but additional arguments can be passed to this function by enclosing them in parentheses in the *new-expression*. For example:

int \* p = new (x) int;

is a valid expression that, at some point, calls:

operator new (sizeof(int),x);

By default, one of the versions of [operator new](http://www.cplusplus.com/operator%20new) is overloaded to accept a parameter of type [nothrow\_t](http://www.cplusplus.com/nothrow_t) (like nothrow). The value itself is not used, but that version of [operator new](http://www.cplusplus.com/operator%20new) shall return a *null pointer* in case of failure instead of throwing an exception.  
  
The same applies to the new[] operator and function [operator new[]](http://www.cplusplus.com/operator%20new%5b%5d)

<http://stackoverflow.com/questions/240212/what-is-the-difference-between-new-delete-and-malloc-free>

#### new/delete: Allocate/release memory

* 1. Memory allocated from 'Free Store'
  2. Returns a fully typed pointer.
  3. new (standard version) never returns a NULL (will throw on failure)
  4. Are called with Type-ID (compiler calculates the size)
  5. Has a version explicitly to handle arrays.
  6. Reallocating (to get more space) not handled intuitively (because of copy constructor).
  7. Whether they call malloc/free is implementation defined.
  8. Can add a new memory allocator to deal with low memory (set\_new\_handler)
  9. operator new/delete can be overridden legally
  10. **constructor/destructor used to initialize/destroy the object**

#### malloc/free: Allocates/release memory

* 1. Memory allocated from 'Heap'
  2. Returns a void\*
  3. Returns NULL on failure
  4. Must specify the size required in bytes.
  5. Allocating array requires manual calculation of space.
  6. Reallocating larger chunk of memory simple (No copy constructor to worry about)
  7. They will **NOT** call new/delete
  8. No way to splice user code into the allocation sequence to help with low memory.
  9. malloc/free cannot be overridden legally

Table comparison of the features:

Feature | new/delete | malloc/free

--------------------------+--------------------------------+-------------------------------

Memory allocated from | 'Free Store' | 'Heap'

Returns | Fully typed pointer | void\*

On failure | Throws (never returns NULL) | Returns NULL

Required size | Calculated by compiler | Must be specified in bytes

Handling arrays | Has an explicit version | Requires manual calculations

Reallocating | Not handled intuitively | Simple (no copy constructor)

Call of reverse | Implementation defined | No

Low memory cases | Can add a new memory allocator | Not handled by user code

Overridable | Yes | No

Use of (con-)/destructor | Yes | No

Technically memory allocated by new comes from the 'Free Store' while memory allocated by malloc comes from the 'Heap'. Whether these two areas are the same is an implementation details, which is another reason that malloc and new cannot be mixed.

## Friend declaration

(<https://en.cppreference.com/w/cpp/language/friend>) The friend declaration appears in a [class body](https://en.cppreference.com/w/cpp/language/class) and grants a function or another class access to private and protected members of the class where the friend declaration appears.

### Description

1) Designates a function or several functions as friends of this class

class Y {

int data; // private member

// the non-member function operator<< will have access to Y's private members

friend [std::ostream](http://en.cppreference.com/w/cpp/io/basic_ostream)& operator<<([std::ostream](http://en.cppreference.com/w/cpp/io/basic_ostream)& out, const Y& o);

friend char\* X::foo(int); // members of other classes can be friends too

friend X::X(char), X::~X(); // constructors and destructors can be friends

};

// friend declaration does not declare a member function

// this operator<< still needs to be defined, as a non-member

[std::ostream](http://en.cppreference.com/w/cpp/io/basic_ostream)& operator<<([std::ostream](http://en.cppreference.com/w/cpp/io/basic_ostream)& out, const Y& y)

{

return out << y.data; // can access private member Y::data

}

2) (only allowed in non-[local](https://en.cppreference.com/w/cpp/language/class#Local_classes) class definitions) Defines a non-member function, and makes it a friend of this class at the same time. Such non-member function is always [inline](https://en.cppreference.com/w/cpp/language/inline).

class X {

int a;

friend void friend\_set(X& p, int i) {

p.a = i; // this is a non-member function

}

public:

void member\_set(int i) {

a = i; // this is a member function

}

};

3) Designates the class, struct, or union named by the *elaborated-class-specifier* (see [elaborated type specifier](https://en.cppreference.com/w/cpp/language/elaborated_type_specifier)) as a friend of this class. This means that the friend's member declarations and definitions can access private and protected members of this class and also that the friend can inherit from private and protected members of this class. The name of the class that is used in this friend declaration does not need to be previously declared.

4) Designates the type named by the *simple-type-specifier* or *typename-specifier* as a friend of this class if that type is a (possibly [cv-qualified](https://en.cppreference.com/w/cpp/language/cv)) class, struct, or union; otherwise the friend declaration is ignored. This declaration will not forward declare new type.

class Y {};

class A {

int data; // private data member

class B { }; // private nested type

enum { a = 100 }; // private enumerator

friend class X; // friend class forward declaration (elaborated class specifier)

friend Y; // friend class declaration (simple type specifier) (since c++11)

};

class X : A::B { // OK: A::B accessible to friend

A::B mx; // OK: A::B accessible to member of friend

class Y {

A::B my; // OK: A::B accessible to nested member of friend

};

int v[A::a]; // OK: A::a accessible to member of friend

};

### Notes

Friendship is not transitive (a friend of your friend is not your friend)

Friendship is not inherited (your friend's children are not your friends)

# Type System and scopes

## Cast

### static\_cast

Converts between types using a combination of implicit and user-defined conversions, including:

* Standard conversion. For instance: from short to int or from int to float.
* User defined conversions (Class conversions.)
* Conversion from derived class to base class. (Take a look at the [**inheritance tutorial**](https://www.codingunit.com/cplusplus-tutorial-inheritance))

static\_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.

This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.

Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic\_cast is avoided.

|  |  |
| --- | --- |
| 1 2 3 4 | class CBase {};  class CDerived: public CBase {};  CBase \* a = new CBase;  CDerived \* b = static\_cast<CDerived\*>(a); |

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.  
  
static\_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

|  |  |
| --- | --- |
| 1 2 | double d=3.14159265;  int i = static\_cast<int>(d); |

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

It performs nonpolymorphic cast. It can be used for any standard conversion. No run-time checks are performed.

int i = 100;

double x = static\_cast<double>(i);

### dynamic\_cast

Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy

dynamic\_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic\_cast is always successful when we cast a class to one of its base classes:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | class CBase { };  class CDerived: public CBase { };  CBase b; CBase\* pb;  CDerived d; CDerived\* pd;  pb = dynamic\_cast<CBase\*>(&d); // ok: derived-to-base  pd = dynamic\_cast<CDerived\*>(&b); // wrong: base-to-derived |

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with dynamic\_cast unless the base class is polymorphic.

When a class is polymorphic, dynamic\_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class

Assume that **Base** is a polymorphic class and **Derived** is derived from **Base.**

Base \*bp, b\_obj;

Derived \*dp, d\_obj;

dp = dynamic\_cast<Derived \*>(&d\_obj); // √

bp = dynamic\_cast<Base \*>(&d\_obj); // √

bp = dynamic\_cast<Base \*>(&b\_obj); // √

dp = dynamic\_cast<Derived \*>(&b\_obj); // X

bp = &d\_obj;

dp = dynamic\_cast<Derived \*>(bp); // √

When failed to perform casting, dynamic\_cast returns null for point, and throws **bad\_cast** on references.

### reinterpret\_cast

* Between integers and pointers
* Between pointers and pointers
* Between function-pointers and function-pointers

It converts one type into a fundamentally different type.

char \*p = “this is a string.”;

int i = reinterpret\_cast<int>(p);

reinterpret\_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.  
  
It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.  
  
The conversions that can be performed by reinterpret\_cast but not by static\_cast have no specific uses in C++ are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

|  |  |
| --- | --- |
| 1 2 3 4 | class A {};  class B {};  A \* a = new A;  B \* b = reinterpret\_cast<B\*>(a); |

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

### const\_cast

This **const\_cast** can be used to cast away **constness** of a const pointer or a const reference, which points to a non-const object.

void sqr(const int \*val){

\*(const\_cast<int \*>(val)) = (\*val) \* (\*val);

}

void sqr(const int &val){

const\_cast<int &>(val) = val \* val;

}

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // const\_cast  #include <iostream>  using namespace std;  void print (char \* str)  {  cout << str << endl;  }  int main () {  const char \* c = "sample text";  print ( const\_cast<char \*> (c) );  return 0;  } | sample text |

Notes:

1. You cannot use const\_cast to change a const value such as “const int x = 100;”. Refer to Appendix: const\_cast confusion
2. Even not using const\_cast, you can use normal cast to change an int value through a const int \*. For example:

int a = 200; const int \*cp = &a; \*(int \*)cp = 300;

### const\_cast confusion

I've got the following program which uses a const\_cast to modify a const int variable:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | #include <iostream>  int main()  {  const int x = 99;  int\* changeConst = const\_cast<int\*>(&x);  \*changeConst = 55;  std::cout << "\nAddress of const value is: " << &x << '\n'  << "The non-const pointer points at: " << changeConst << " by means of const\_cast.\n\n";    if(&x == changeConst)  std::cout << "The pointer points to a constant value.\n\n";  else  std::cout << "The pointer points to a different value...\n\n";    std::cout << "Value of x: " << x << '\n'  << "Value of dereferenced pointer: " << \*changeConst << "\n\n";    if(&x == changeConst)  std::cout << "The pointer still points to a constant value.\n\n";  else  std::cout << "The pointer now points to a different value...\n\n";    } |

I get the following output:

|  |
| --- |
| Address of const value is: 0x7fff93b59acc  The non-const pointer points at: 0x7fff93b59acc by means of const\_cast.  The pointer points to a constant value.  Value of x: 99  Value of dereferenced pointer: 55  The pointer still points to a constant value. |

why, even though my pointer is pointing at x and when I dereference it I'm getting the value I supposedly changed it to but if I print x directly I get the old value?

***Answer***: There was a thread about this a few months ago. I can summarize its contents to this simple statement: Do **not** cast pointers to constants to pointers to variables.  
  
On line 5, you're telling the compiler "I promise the value of x will never change from 99". The compiler can then use x as a symbolic constant, and instead of using a run time variable, it will evaluate x at compile time and replace it with 99. These two lines generate the exact same code:

|  |  |
| --- | --- |
| 1 2 | std::cout <<x;  std::cout <<99; |

(Note: The compiler may not actually do this. This is one possible optimization that the compiler may or may not implement. However, the compiler I'm talking about is known to do this.)  
  
When you do &x, the compiler is forced to allocate a run time variable, because it's not possible to get the address of literal constants. The type of &x is 'const int \*'. This is so the compiler won't let you do something like \*&x=10;. However, the variable itself, like all data, can be modified. If you cast the pointer to (int \*), the compiler will now let you modify the variable. It's possible to do \*(int \*)&x=10;.  
However, this is a breach of the promise you made on line 5. You're modifying a variable you told the compiler would never change. The compiler isn't aware of this change because you're making it through a dereferenced pointer, and it will still understand a direct reference to 'x' as meaning '99'.  
This is what causes the situation where (x!=\*&x).  
  
Pointers to constants exist so that the compiler can keep you from changing things you shouldn't. If you then go and strip this information, the compiler can no longer keep you from shooting yourself in the foot.

## Type conversion

### Integral Promotions [2]

A *char*, *short int*, *enumerator*, *object of enumeration type*, or an *int-bit-field* may be used wherever an integer may be used. If an *int* can represent all the values of the original type, the value is converted to *int*; otherwise it is converted to *unsigned int*. This process is called ***integral promotion***.

C++ follow the ANSI C in defining integral promotions as “value-preserving”.

But many Classic C compilers use “unsigned-preserving” integral promotions, which breaks in the following example

int i=-1;

unsigned short us = 2;

int k = (i + us) < 42;

ANSI C: k = 1; which for Classic C, k=0.

### Arithmetic Conversion [2]

long double ← double ← float ← integral promotion

(<https://www.learncpp.com/cpp-tutorial/44-implicit-type-conversion-coercion/>) When evaluating expressions, the compiler breaks each expression down into individual subexpressions. The arithmetic operators require their operands to be of the same type. To ensure this, the compiler uses the following rules:

* If an operand is an integer that is narrower than an int, it undergoes integral promotion (as described above) to int or unsigned int.
* If the operands still do not match, then the compiler finds the highest priority operand and implicitly converts the other operand to match.

The priority of operands is as follows:

* long double (highest)
* double
* float
* unsigned long long
* long long
* unsigned long
* long
* unsigned int
* int (lowest)

### Pointer Conversion [2]

A constant expression that evaluates to zero is converted to *null pointer*.

C++ a *void \** cannot be assigned to an object of any type other than *void\** without explicit cast. (ANSI C allows, for example, *malloc*)

### Object Conversion

[The annotated C++ Reference Manual: Stroustrup Pp270] Type conversion of class objects can be specified by constructors and by conversion functions. Such conversions, often called *user-defined conversions*, are used implicitly in addtion to standard conversions.

For example, a function expecting an argument of type *X* can be called not only with an argument of type *X* but also with an argument of type *T* where a conversion from *T* to *X* exists.

User-defined conversions are applied only where they are unambiguous.

#### Conversion by Constructor

A constructor accepting a single argument specifieds a conversion from its argument type to the class. For example:

class X {

public:

X(int);

X(const char \* int = 0);

};

void f(X arg) { …… }

int main() {

X a = 1; // a = X(1)

X b = “Jessie”; // a = X(“Jessie”, 0)

a = 2; // a = X(2)

f(3); // f(X(3))

}

#### Conversion Functions

A member function of a class X with a name of the form

*Conversion-function-name:*

Operator *conversion-type-name*

*Conversion-type-name:*

*Type-specifier-list ptr-operatoropt*

specifies a conversion from X to the type specified by the conversion-type-name. Classes, enumerations, and typedef-names may not be declared in the type-specifier-list.

Conversion functions can do two things that canot be specified by constructors:

* 1. Define a conversion from a class to a basic type;
  2. Define a conversion from one class to another class without modifying the declaration for the other class.

For example:

class X {

public:

int\* operator int\*(); // error

operator void\*(int); // error

operator char\*(); // ok

operator int(); // ok

};

Void f(X a){

int i = int(a);

i = (int)a;

i = a;

}

### Implicit Conversion

Consider a class for representing rational numbers:

class Rational {

public:

Rational(int numerator = 0, int denominator = 1);

int numerator() const;

int denominator() const;

const Rational operator\*(const Rational& rhs) const;

private:

...

};

You'd also like to support mixed-mode operations, where Rationals can be multiplied with, for example, ints. When you try to do this, however, you find that it works only half the time: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp8) [Item E19, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp8)

result = oneHalf \* 2; // fine

result = 2 \* oneHalf; // error!

The object oneHalf is an instance of a class that contains an operator\*, so your compilers call that function. However, the integer 2 has no associated class, hence no operator\* member function. Your compilers will also look for a non-member operator\* (i.e., one that's in a visible namespace or is global) that can be called like this, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp11) [Item E19, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp11)

result = operator\*(2, oneHalf); // error!

but there is no non-member operator\* taking an int and a Rational, so the search fails. [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp12) [Item E19, P12](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp12)

Look again at the call that succeeds. You'll see that its second parameter is the integer 2, yet Rational::operator\* takes a Rational object as its argument. What's going on here? Why does 2 work in one position and not in the other? [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp13) [Item E19, P13](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp13)

What's going on is implicit type conversion. Your compilers know you're passing an int and the function requires a Rational, but they also know that they can conjure up a suitable Rational by calling the Rational constructor with the int you provided, so that's what they do (see [Item M19](http://lzhang.dyndns.org/books/1_EffectiveC2nd/MEC/MI19_FR.HTM#41177)). In other words, they treat the call as if it had been written more or less like this: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp14) [Item E19, P14](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp14)

const Rational temp(2); // create a temporary

// Rational object from 2

result = oneHalf \* temp; // same as

// oneHalf.operator\*(temp);

Of course, they do this only when non-explicit constructors are involved, because explicit constructors can't be used for implicit conversions; that's what explicit means. If Rational were defined like this, [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp15) [Item E19, P15](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp15)

class Rational {

public:

explicit Rational(int numerator = 0, // this ctor is

int denominator = 1); // now *explicit*

...

const Rational operator\*(const Rational& rhs) const;

...

};

*neither* of these statements would compile: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp16) [Item E19, P16](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI19_DIR.HTM#dingp16)

result = oneHalf \* 2; // error!

result = 2 \* oneHalf; // error!

### Slicing

Passing parameters by reference has another advantage: it avoids what is sometimes called the "slicing problem." When a derived class object is passed as a base class object, all the specialized features that make it behave like a derived class object are "sliced" off, and you're left with a simple base class object. This is almost never what you want.

### Ambiguity

#### Int 0 and NULL

More specifically, what will happen here? [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI25_DIR.HTM#dingp3) [Item E25, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI25_DIR.HTM#dingp3)

void f(int x);

void f(string \*ps);

f(0); // calls f(int) or f(string\*)?

f(static\_cast<string\*>(NULL)); // calls f(string\*)

f(static\_cast<string\*>(0)); // calls f(string\*)

The answer is that 0 is an int — a literal integer constant, to be precise — so f(int) will always be called.

#### Operator vs Constructor

Here's an example of potential ambiguity: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp3) [Item E26, P3](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp3)

class B; // forward declaration for

// class B

class A {

public:

A(const B&); // an A can be

// constructed from a B

};

class B {

public:

operator A() const; // a B can be

// converted to an A

};

There's nothing wrong with these class declarations — they can coexist in the same program without the slightest trouble. However, look what happens when you combine these classes with a function that takes an A object, but is actually passed a B object: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp4) [Item E26, P4](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp4)

void f(const A&);

B b;

f(b); // error! — ambiguous

#### What is converted to “double”

A similar form of ambiguity arises from standard conversions in the language — you don't even need any classes: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp7) [Item E26, P7](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp7)

void f(int);

void f(char);

double d = 6.02;

f(d); // error! — ambiguous

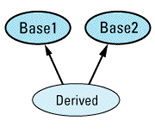
Should d be converted into an int or a char? The conversions are equally good, so compilers won't judge. Fortunately, you can get around this problem by using an explicit cast: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp8) [Item E26, P8](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp8)

f(static\_cast<int>(d)); // fine, calls f(int)

f(static\_cast<char>(d)); // fine, calls f(char)

#### Same function from different base

Multiple inheritance (see [Item 43](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI43_FR.HTM#7778)) is rife with possibilities for potential ambiguity. The most straightforward case occurs when a derived class inherits the same member name from more than one base class: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp9) [Item E26, P9](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp9)

class Base1 {

public:

int doIt();

};

class Base2 {

public:

void doIt ();

};

class Derived: public Base1, // Derived doesn't declare

public Base2 { // a function called doIt

...

};

Derived d;

d.doIt(); // error! — ambiguous

When class Derived inherits two functions with the same name, C++ utters not a whimper; at this point the ambiguity is only potential. However, the call to doIt forces compilers to face the issue, and unless you explicitly disambiguate the call by specifying which base class function you want, the call is an error: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp10) [Item E26, P10](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp10)

d.Base1::doIt(); // fine, calls Base1::doIt

d.Base2::doIt(); // fine, calls Base2::doIt

That doesn't upset too many people, but the fact that accessibility restrictions don't enter into the picture has caused more than one otherwise pacifistic soul to contemplate distinctly unpacifistic actions: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp11) [Item E26, P11](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI26_DIR.HTM#dingp11)

class Base1 { ... }; // same as above

class Base2 {

private:

void doIt(); // **this function is now**

}; // **private**

class Derived: public Base1, public Base2

{ ... }; // same as above

Derived d;

int i = d.doIt(); // error! — still ambiguous!

## RTTI

<https://www.codingunit.com/cplusplus-tutorial-typecasting-part-2-rtti-dynamic_cast-typeid-and-type_info>

Runtime Type Information (RTTI) is the concept of determining the type of any variable during execution (runtime.) The RTTI mechanism contains:

* The operator dynamic\_cast
* The operator typeid
* The struct type\_info

RTTI can only be used with polymorphic types. This means that with each class you make, you must have at least one virtual function (either directly or through inheritance.)

To obtain an object’s type, use **typeid**, which is included in the header <**typeinfo**>. It returns an object of type **type\_info**, which defines the following 4 public member functions:

bool operator==(const type\_info &ob);

bool operator!=(const type\_info &ob);

bool before(const type\_info &ob);

const char \*name();

template <class T> class myclass{…}

myclass<int> myint;

myclass<double> mydbl;

cout << typeid(myint).name(); // output myclass<int>

cout << typeid(mydbl).name(); // output myclass<double>

To simplify the logic in certain situation, you can use **dynamic\_cast** to replace **typeid.**

https://www.codingunit.com/cplusplus-tutorial-typecasting-part-2-rtti-dynamic\_cast-typeid-and-type\_info

## Keyword: const

Inside a member function of class C, the this pointer behaves as if it had been declared as follows: [¤](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI21_DIR.HTM#dingp27) [Item E21, P27](http://lzhang.dyndns.org/books/1_EffectiveC2nd/EC/EI21_DIR.HTM#dingp27)

**C \* const this; // for non-const member functions**

**const C \* const this; // for const member functions**

## Keyword: static in C

K&R explain it in section A4.1 of "The C Programming Language".

In short, the word static is used with two meanings:

Static is one of the two ***storage classes*** (the other being automatic). A static object keeps its value between invocations. The objects declared outside all blocks are always static and cannot be made automatic.

But, when the static keyword (big emphasis on it being used in code as a keyword) is used with a declaration, it gives that object internal linkage so it can only be used within that translation unit. But if the keyword is used in a function, it changes the storage class of the object (the object would only be visible within that function anyway). The opposite of static is the extern keyword, which gives an object external linkage.

Peter Van Der Linden gives these two meanings in "Expert C Programming":

1. Inside a function, retains its value between calls.
2. At the function level, visible only in this file.

## Keyword: static in C++

<http://en.cppreference.com/w/cpp/language/static>

Inside a class definition, the keyword [static](http://en.cppreference.com/w/cpp/keywords/static) declares members that are not bound to class instances.

The static keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member:

class X { static int n; }; // declaration (uses 'static')

int X::n = 1; // definition (does not use 'static')

The declaration inside the class body is not a definition and may declare the member to be of incomplete type (other than void), including the type in which the member is declared:

struct Foo;

struct S

{

static int a[]; // declaration, incomplete type

static Foo x; // declaration, incomplete type

static S s; // declaration, incomplete type (inside its own definition)

};

int S::a[10]; // definition, complete type

struct Foo {};

Foo S::x; // definition, complete type

S S::s; // definition, complete type

#### Static member functions

Static member functions are not associated with any object. When called, they have no this pointer.

Static member functions cannot be virtual, const, or volatile.

The address of a static member function may be stored in a regular [pointer to function](http://en.cppreference.com/w/cpp/language/pointer#Pointers_to_functions), but not in a [pointer to member function](http://en.cppreference.com/w/cpp/language/pointer#Pointers_to_member_functions).

#### Static data members

Static data members are not associated with any object. They exist even if no objects of the class have been defined. If the static member is declared thread\_local(since C++11), there is one such object per thread. Otherwise, there is only one instance of the static data member in the entire program, with static [storage duration](http://en.cppreference.com/w/cpp/language/storage_duration).

Static data members cannot be mutable.

Static data members of a class in namespace scope have [external linkage](http://en.cppreference.com/w/cpp/language/storage_duration) if the class itself has external linkage (i.e. is not a member of [unnamed namespace](http://en.cppreference.com/w/cpp/language/namespace#Unnamed_namespaces)). Local classes (classes defined inside functions) and unnamed classes, including member classes of unnamed classes, cannot have static data members.

#### Constant static members

If a static data member of integral or enumeration type is declared const (and not volatile), it can be initialized with a [initializer](http://en.cppreference.com/w/cpp/language/initialization) in which every expression is a [constant expression](http://en.cppreference.com/w/cpp/language/constexpr), right inside the class definition:

struct X

{

const static int n = {1};

const static int m{2}; // since C++11

const static int k;

};

const int X::k = 3;

## Keyword: extern

<http://stackoverflow.com/questions/496448/how-to-correctly-use-the-extern-keyword-in-c>

"extern" changes the linkage. With the keyword, the function / variable is assumed to be available somewhere else and the resolving is deferred to the linker.

There's a difference between "extern" on functions and on variables: on variables it doesn't instantiate the variable itself, i.e. doesn't allocate any memory. This needs to be done somewhere else. Thus it's important if you want to import the variable from somewhere else. For functions, this only tells the compiler that linkage is extern. As this is the default (you use the keyword "static" to indicate that a function is not bound using extern linkage) you don't need to use it explicitly.

## Temporary Variable

<http://stackoverflow.com/questions/6901494/crash-on-accessing-reference-to-string>

#include <iostream>

#include <string>

#include <vector>

using namespace std;

class myclass {

const string& m\_str;

public:

myclass(string s) : m\_str(s) {}

const string& getString() const { return m\_str; }

};

void test1 () {

const string str("honey");

myclass mc(str);

cout << mc.getString() << "\n";

}

void test2 () {

myclass mc("abc");

cout << mc.getString() << "\n";

}

const string &foo(const vector<string> &v)

{

return v[0];

}

void test3()

{

vector<string> v = {"ab", "cd"};

const string &s = foo(v);

cout << s << endl;

}

int main()

{

test1();

test2();

test3();

return 0;

}

the myclass constructor is taking a string by value which makes it a temporary. You're then binding this temporary to the m\_str member. As soo as the constructor exits your member reference becomes invalid. Instead: myclass(const string& s) : m\_str(s) {}

Even so that may not be a good idea. Generally speaking using references as members *can* be dangerous because you have to be very clear about lifetime semantics. You should consider just storing the string by value in your class unless you have a specific reason not to do so.

Note: I could not crash it with g++ -std=c++11 on MacBook.

## Item 23:  Don't try to return a reference when you must return an object. [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp1) [Item E23, P1](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp1)

But remember that a reference is just a *name*, a name for some *existing* object. Whenever you see the declaration for a reference, you should immediately ask yourself what it is another name for, because it must be another name for *something*

// the second wrong way to write this function

inline const Rational& operator\*(const Rational& lhs,

const Rational& rhs)

{

Rational \*result =

new Rational(lhs.n \* rhs.n, lhs.d \* rhs.d);

return \*result;

}

Well, you *still* have to pay for a constructor call, because the memory allocated by new is initialized by calling an appropriate constructor (see Items [5](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI5_FR.HTM#1869) and [M8](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\MEC\MI8_FR.HTM#33985)), but now you have a different problem: who will apply delete to the object that was conjured up by your use of new? [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp12) [Item E23, P12](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp12)

In fact, this is a guaranteed memory leak.

Perhaps the following implementation occurs to you, an implementation based on operator\* returning a reference to a *static* Rational object, one defined *inside* the function: [¤](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp15) [Item E23, P15](file:///I:\Dropbox\~Family\z-lei\homepage\1_Effective%20C++%202nd\EC\EI23_DIR.HTM#dingp15)

// the third wrong way to write this function

inline const Rational& operator\*(const Rational& lhs,

const Rational& rhs)

{

static Rational result; // static object to which a

// reference will be returned

*somehow multiply lhs and rhs and put the*

*resulting value inside result;*

return result;

}

This looks promising. But ((a\*b) == (c\*d)) will always evaluate to true, regardless of the values of a, b, c, and d!

## Item 31:  Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.

This problem usually raises its ugly head when you try to improve the efficiency of a function by returning its result by reference instead of by value.

# C and C++

## Major difference between C and C++

1. In C++, local variables can be declared anywhere within the block
2. In C, a function declaration int f() does not say anything about the function; in C++, it means int f(void)
3. In C++, all functions must be prototyped
4. C++ does not “default-to-int” if function is not declared.
5. In C++, sizeof(‘a’)=1 so as to differentiate char type from int type. But in C, sizeof(‘a’)=sizeof(int)=2: here there is automatic char-to-int promotion

### Call C code from C++

<https://isocpp.org/wiki/faq/mixing-c-and-cpp>

Just declare the C function extern "C" (in your C++ code) and call it (from your C or C++ code). For example:

1. *// C++ code*
2. extern "C" void f(int); *// one way*
3. extern "C" { *// another way*
4. int g(double);
5. double h();
6. };
7. void code(int i, double d)
8. {
9. f(i);
10. int ii = g(d);
11. double dd = h();
12. *// ...*
13. }

Note that C++ type rules, not C rules, are used. So you can’t call function declared extern "C" with the wrong number of arguments

### Call C++ from C

Just declare the C++ function extern "C" (in your C++ code) and call it (from your C or C++ code). For example:

1. *// C++ code:*
2. extern "C" void f(int);
3. void f(int i)
4. {
5. *// ...*
6. }

Now f() can be used like this:

1. */\* C code: \*/*
2. void f(int);
3. void cc(int i)
4. {
5. f(i);
6. */\* ... \*/*
7. }

Naturally, this works only for non-member functions. If you want to call member functions (incl. virtual functions) from C, you need to provide a simple wrapper. For example:

1. *// C++ code:*
2. class C {
3. *// ...*
4. virtual double f(int);
5. };
6. extern "C" double call\_C\_f(C\* p, int i) *// wrapper function*
7. {
8. return p->f(i);
9. }

Now C::f() can be used like this:

1. */\* C code: \*/*
2. double call\_C\_f(struct C\* p, int i);
3. void ccc(struct C\* p, int i)
4. {
5. double d = call\_C\_f(p,i);
6. */\* ... \*/*
7. }

If you want to call overloaded functions from C, you must provide wrappers with distinct names for the C code to use. For example:

1. *// C++ code:*
2. void f(int);
3. void f(double);
4. extern "C" void f\_i(int i) { f(i); }
5. extern "C" void f\_d(double d) { f(d); }

Now the f() functions can be used like this:

1. */\* C code: \*/*
2. void f\_i(int);
3. void f\_d(double);
4. void cccc(int i,double d)
5. {
6. f\_i(i);
7. f\_d(d);
8. */\* ... \*/*
9. }

Note that these techniques can be used to call a C++ library from C code even if you cannot (or do not want to) modify the C++ headers.

## C++11

### auto

### lambda

<http://en.cppreference.com/w/cpp/language/lambda>

Constructs a [closure](http://en.wikipedia.com/wiki/Closure_(computer_science)): an unnamed function object capable of capturing variables in scope**.**

The lambda expression is a prvalue expression whose value is (until C++17)whose result object is (since C++17) an unnamed temporary object of unique unnamed non-union non-aggregate class type, known as *closure type*, which is declared (for the purposes of [ADL](http://en.cppreference.com/w/cpp/language/adl)) in the smallest block scope, class scope, or namespace scope that contains the lambda expression.

#### Syntax

**[** *capture-list* **]** **(** *params* **)** **mutable**(optional) **constexpr**(opt)(17) *exception* **->** *ret* **{** *body* **}**

**[** *capture-list* **]** **(** *params* **)** **->** *ret* **{** *body* **}**

**[** *capture-list* **]** **(** *params* **)** **{** *body* **}**

**[** *capture-list* **]** **{** *body* **}**

***capture-list*:**

a comma-separated list of zero or more captures, optionally beginning with a *capture-default*.

can be passed as follows (see below for the detailed description):

* **[a,&b]** where *a* is captured by value and *b* is captured by reference.
* **[this]** captures the **this pointer** by value
* **[&]** captures all **automatic** variables **odr-used** in the body of the lambda by reference
* **[=]** captures all **automatic** variables **odr-used** in the body of the lambda by value
* **[]** captures nothing

#### Examples:

**#include <vector>**

**#include <iostream>**

**#include <algorithm>**

**#include <functional>**

**int main()**

**{**

**std::vector<int> c = {1, 2, 3, 4, 5, 6, 7};**

**int x = 5;**

**c.erase(std::remove\_if(c.begin(), c.end(),**

**[x](int n) { return n < x; }), c.end());**

**std::cout << "c: ";**

**std::for\_each(c.begin(), c.end(), [](int i){ std::cout << i << ' '; });**

**std::cout << '\n';**

**// the type of a closure cannot be named, but can be inferred with auto**

**auto func1 = [](int i) { return i + 4; };**

**std::cout << "func1: " << func1(6) << '\n';**

**// like all callable objects, closures can be captured in std::function**

**// (this may incur unnecessary overhead)**

**std::function<int(int)> func2 = [](int i) { return i + 4; };**

**std::cout << "func2: " << func2(6) << '\n';**

**}**

#### syntax

(<https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2017>) The ISO C++ Standard shows a simple lambda that is passed as the third argument to the std::sort() function:

#include <algorithm>

#include <cmath>

void abssort(float\* x, unsigned n) {

std::sort(x, x + n,

// Lambda expression begins

[](float a, float b) {

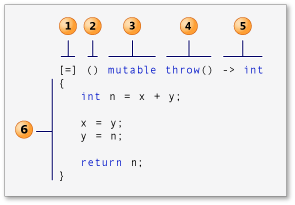
return (std::abs(a) < std::abs(b));

} // end of lambda expression

);

}

This illustration shows the parts of a lambda:



1. **capture clause** (Also known as the lambda-introducer in the C++ specification.)
2. **parameter list** Optional. (Also known as the lambda declarator)
3. mutable specification Optional.
4. exception-specification Optional.
5. trailing-**return-type** Optional.
6. lambda **body**.

#### Capture Clause

A lambda can introduce new variables in its body (in **C++14**), and it can also access, or capture, variables from the surrounding scope. A lambda begins with the capture clause (lambda-introducer in the Standard syntax), which ***specifies which variables are captured, and whether the capture is by value or by reference***. Variables that have the ampersand (&) prefix are accessed by reference and variables that do not have it are accessed by value.

An empty capture clause, [ ], indicates that the body of the lambda expression accesses no variables in the enclosing scope.

You can use the default capture mode (**capture-default** in the Standard syntax) to indicate how to capture any outside variables that are referenced in the lambda: [**&**] means all variables that you refer to are captured by reference, and [**=**] means they are captured by value. You can use a default capture mode, and then specify the opposite mode explicitly for specific variables. For example, if a lambda body accesses the external variable total by reference and the external variable factor by value, then the following capture clauses are equivalent:

[&total, factor]

[factor, &total]

[&, factor]

[factor, &]

[=, &total]

[&total, =]

Only variables that are mentioned in the lambda are captured when a capture-default is used.

If a capture clause includes a capture-default &, then no identifier in a capture of that capture clause can have the form & identifier. Likewise, if the capture clause includes a capture-default =, then no capture of that capture clause can have the form = identifier. An identifier or **this** cannot appear more than once in a capture clause. The following code snippet illustrates some examples.

struct S { void f(int i); };

void S::f(int i) {

[&, i]{}; // OK

[&, &i]{}; // ERROR: i preceded by & when & is the default

[=, this]{}; // ERROR: this when = is the default

[=, \*this]{ }; // OK: captures this by value. See below.

[i, i]{}; // ERROR: i repeated

}

A capture followed by an ellipsis is a pack expansion, as shown in this [variadic template](https://docs.microsoft.com/en-us/cpp/cpp/ellipses-and-variadic-templates?view=vs-2017)example:

template<class... Args>

void f(Args... args) {

auto x = [args...] { return g(args...); };

x();

}

To use lambda expressions in the body of a class method, pass ***this*** pointer to the capture clause to provide access to the methods and data members of the enclosing class

## L-Value and R-Value Expressions

(<https://docs.microsoft.com/en-us/cpp/c-language/l-value-and-r-value-expressions?view=vs-20>) Expressions that refer to memory locations are called "l-value" expressions. An l-value represents a storage region's "locator" value, or a "left" value, implying that it can appear on the left of the equal sign (**=**). L-values are often identifiers.

Expressions referring to modifiable locations are called "modifiable l-values." A modifiable l-value cannot have an array type, an incomplete type, or a type with the **const** attribute. For structures and unions to be modifiable l-values, they must not have any members with the **const** attribute. The name of the identifier denotes a storage location, while the value of the variable is the value stored at that location.

An identifier is a modifiable l-value if it refers to a memory location and if its type is arithmetic, structure, union, or pointer. For example, if ptr is a pointer to a storage region, then \*ptr is a modifiable l-value that designates the storage region to which ptr points.

Any of the following C expressions can be l-value expressions:

* An identifier of integral, floating, pointer, structure, or union type
* A subscript (**[ ]**) expression that does not evaluate to an array
* A member-selection expression (**->** or **.**)
* A unary-indirection (**\***) expression that does not refer to an array
* An l-value expression in parentheses
* A **const** object (a nonmodifiable l-value)

The term "r-value" is sometimes used to describe the value of an expression and to distinguish it from an l-value. All l-values are r-values but not all r-values are l-values.

(<https://www.geeksforgeeks.org/lvalue-and-rvalue-in-c-language/>) r-value” refers to data value that is stored at some address in memory. A r-value is an expression that can’t have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).

## Return by value

(<https://isocpp.org/wiki/faq/ctors#return-by-value-optimization>) Does return-by-value mean extra copies and extra overhead?

Not necessarily.

All(?) commercial-grade compilers optimize away the extra copy.

Some compilers also completely optimize away local variables. They do this the same way as described [earlier](https://isocpp.org/wiki/faq/ctors#return-by-value-optimization): the return-by-value in function is implemented as pass-by-pointer, where the pointer points to the location where the returned object is to be initialized.

So instead of constructing y as a local object, these compilers simply construct \*put\_result\_here, and everytime they see variable y used in the original source code, they substitute \*put\_result\_here instead. Then the line return y; becomes simply return; since the returned object has already been constructed in the location designated by the caller.

# Generic Programming

## Template

[Herbert Schildt: The Complete Reference, C++, Third Edition]

“Templates help you achieve one of the most elusive goals in programming: the creation of reusable code”.

### Typename

Two uses:

1. To substitute **class** in the template declaration
2. To inform the compiler that a name used in the template declaration is a type name rather than an object name

<http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fkeyword_typename.htm>

template<class T> class A

{

T::x(y);

typedef char C;

A::C d;

}

The statement T::x(y) is ambiguous. It could be a call to function x() with a nonlocal argument y, or it could be a declaration of variable y with type T::x. C++ will interpret this statement as a function call. In order for the compiler to interpret this statement as a declaration, you would add the keyword typename to the beginning of it. The statement A::C d; is ill-formed. The class A also refers to A<T> and thus depends on a template parameter. You must add the keyword typename to the beginning of this declaration:

typename A::C d;

<http://pages.cs.wisc.edu/~driscoll/typename.html>

there is a C++ parsing rule that says that qualified dependent names should be parsed as non-types even if it leads to a syntax error

This rule even holds if it doesn't make sense **even if it doesn't make sense to refer to a non-type**. For instance, suppose we were to do something more typical and declare an iterator instead of a pointer to an iterator:

template <class T>

void foo() {

**typename** T::iterator iter;

...

}

### Export - separate compilation model

<http://msmvps.com/blogs/vandooren/archive/2008/09/24/c-keyword-of-the-day-export.aspx>

The export keyword is a bit like the Higgs boson of C++. Theoretically it exists, it is described by the standard, and noone has seen it in the wild. :)

It seems a bit strange that a language should contain a keyword that all compiler makers refuse to implement. Comeau has some [interesting content here](http://www.comeaucomputing.com/iso/promises.html) as does [Wikipedia](http://en.wikipedia.org/wiki/Export_%28C%2B%2B%29#Standards_compliance).

The reason that I believe 'export' has never really taken off is that is seems to be an significant effort to implement it, combined with the fact that is doesn't achieve all that much. The export keyword allows a C++ programmer to declare a template class in a header, and then provide the implementation separately in a cpp file. This is the normal C++ way of doing things. Unfortunately, there's a snag.

Template class implementations cannot be compiled on their own. Template code itself is meaningless without specification of template arguments. Thus it is that a compiler doesn't know what to do with an implementation until the template class is used somewhere. At that point the template arguments will be known, and the compiler can compile the template class.

Since the template is only compiled then, the easiest thing to do would be to do what most compilers do now: demand that the implementation is in scope when it is used somewhere. I.e. the implementation has to be put in a header file, and that header file has to be included by the source file that is being compiled. the reason for this is that the compiler would simply not know where to look for it if the implementation was in a cpp file somewhere.

The export keyword would solve this by telling the compiler 'Look somewhere else for the implementation'. The compiler would then have to compile the implementation for that class, and save the object code somewhere for that combination of template arguments.

The export keyword sounds like a great thing, but it doesn't solve that many problems.

### Explicit specialization

“If you overload a generic function, that overloaded function overrides (or “hides”) the generic function relateive to that specific version”.

template <class X> void swapargs(X &a, X&b){…}

void swapargs(int &a, int &b){…}

int main(){

int i=10, j=20;

double x=10.1, y=23.2;

char a=’x’, b=’y’;

swapargs(i, j); // calls 2nd

swapargs(x, y); // calls template

swapargs(a, b); // calls template

}

“… the compiler does not generate this version of the generic swapargs() function, because the generic function is overridden by the explicitoverloading”

**New Style**

template**<>** void swapargs**<int>**(int &a, int &b){…}

“…While there is no advantage to using one specialization syntax over the other at this time, the new-style is probably a better approach for the long term.”

### Generic Function Restriction

“… a generic function must perform the same general action for all versions – only the type of data can differ”.

### Non-Type Arguments with Generic Classes

template <class AType, int size> class myarray

{

AType a[size];

public:

AType &operator[](int i);

};

template<class AType, int size>

AType &myarray<AType, size>::operator[](int i){…}

…

myarray<double, 15> doubleob;

bmarray<int, 10> intob;

Why non-type argument? “… Even though size is depicted as a ‘variable’ in the source code, **its value is known at compile time**”

“Non-type arguments are restricted to integers, pointers, or references”

“non-type arguments are treated as constants”.

### Default Arguments with Template Classes

template <class AType=int, int size=10> class myarray

{

AType a[size];

public:

AType &operator[](int i);

};

template<class AType, int size>

AType &myarray<AType, size>::operator[](int i){…}

…

myarray<double, 15> doubleob;

myarray<int, 10> intob;

myarray<> defaultarry;

Explicit Class Specification

template <class T> class myclass{…}

template<> class myclass<int>{…} // explicit specification

## STL

<http://www.cplusplus.com/reference/stl/>

### Example

#include <iostream>

#include <vector>

#include <**iterator**>

#include <algorithm>

#include <**functional**>

using namespace std;

main()

{

vector<int> v;

for (int i=0; i<20; ++i) v.push\_back(i);

**copy**(v.begin(), v.end(), **ostream\_iterator**<int>(cout, " "));

cout << endl;

vector<int>::iterator it = **remove\_if**(v.begin(), v.end(),

**not1**(**bind2nd**(**greater**<int>(), 10)));

copy(v.begin(), it, ostream\_iterator<int>(cout, " "));

cout << endl;

}

=======================================================

vector<string> words;  
ifstream file ("words.txt");  
if (file){  
    copy (**istream\_iterator**<string> (file),

istream\_iterator<string>(),

**back\_inserter** (words));

}

### Philosophy

The Standard Template Library is designed for use with a style of programming called generic programming. The essential idea behind generic programming is to create components that can be composed easily without losing any performance. In some sense, it moves the effort that is done at run-time in object-oriented programming (dynamic binding) to compile-time, using templates.

### Concepts and Modeling

[http://www.sgi.com/tech/stl/stl\_introduction.html] One very important question to ask about any template function, not just about STL algorithms, is what the set of types is that may correctly be substituted for the formal template parameters. Clearly, for example, int\* or double\* may be substituted for find's formal template parameter InputIterator. Equally clearly, int or double may not: find uses the expression \*first, and the dereference operator makes no sense for an object of type int or of type double. The basic answer, then, is that find implicitly defines a set of requirements on types, and that it may be instantiated with any type that satisfies those requirements. Whatever type is substituted for InputIterator must provide certain operations: it must be possible to compare two objects of that type for equality, it must be possible to increment an object of that type, it must be possible to dereference an object of that type to obtain the object that it points to, and so on.

Find isn't the only STL algorithm that has such a set of requirements; the arguments to [for\_each](http://www.sgi.com/tech/stl/for_each.html) and [count](http://www.sgi.com/tech/stl/count.html), and other algorithms, must satisfy the same requirements. These requirements are sufficiently important that we give them a name: we call such a set of type requirements a *concept*, and we call this particular concept [**Input Iterator**](http://www.sgi.com/tech/stl/InputIterator.html). We say that a type *conforms to a concept*, or that it *is a model of a concept*, if it satisfies all of those requirements. We say that int\* is a model of **Input Iterator** because int\* provides all of the operations that are specified by the **Input Iterator** requirements.

Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation: the author of find only has to consider the interface specified by the concept **Input Iterator**, rather than the implementation of every possible type that conforms to that concept. Similarly, if you want to use find, you need only to ensure that the arguments you pass to it are models of **Input Iterator.** This is the reason why find and reverse can be used with lists, vectors, C arrays, and many other types: **programming in terms of concepts**, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

## Special pointers

### unique\_ptr

#### Basic capabilities

To put it simply, unique\_ptr should be the default smart pointer used by new C++ code, replacing "raw" pointers as much as possible. unique\_ptr cleanly represents the single ownership idiom - it cannot be copied and assigned, and it cleans up the pointed object when it's destructed.

Here's some code to demonstrate this [[1]](https://eli.thegreenplace.net/2012/06/20/c11-using-unique_ptr-with-standard-library-containers#id6):

#include <iostream>

#include <cstdlib>

#include <memory>

**using** **namespace** std;

**struct** Foo {

Foo() {cerr << "Foo [" << **this** << "] constructed\n";}

**virtual** ~Foo() {cerr << "Foo [" << **this** << "] destructed\n";}

};

**int** main(**int** argc, **char**\*\* argv) {

// .. some code

{

unique\_ptr<Foo> fp(**new** Foo());

unique\_ptr<Foo> fp2(fp); // ERROR! can't copy unique\_ptr

unique\_ptr<Foo> fp3;

fp3 = fp; // ERROR! can't assign unique\_ptr

cerr << "Exiting scope\n";

} // fp will be destroyed, and will destruct the pointed object

**return** 0;

}

In addition to managing the pointed object's lifetime, unique\_ptr provides the other expected capabilities of a smart pointer: it overloads operator\* and operator->, provides a means to obtain the raw pointer (get), to relinquish control of the pointed object (release), and to replace the object it manages (reset). It also lets you customize the way the pointed object is deleted (if you don't want it to be the default delete operator), and has some other niceties - just consult your favorite C++ reference.

#### Containers - motivation

So unique\_ptr is a useful single-ownership smart pointer. But what makes it really shine (especially when compared to auto\_ptr) is that it can be used in standard containers.

Why is it so important to be able to place smart pointers into containers? Because holding objects by value is sometimes very expensive. Containers, especially when coupled with algorithms, tend to move objects around. Large objects are expensive to copy, hence we'd like to keep pointers to objects inside containers instead.

#### shared pointers?

Another smart pointer C++11 brings with it is the shared\_ptr/weak\_ptr pair, implementing a reference-counted approach to shared ownership. While much more flexible than unique\_ptr, shared\_ptr is slower and consumes more memory; managing the reference count is not free [[5]](https://eli.thegreenplace.net/2012/06/20/c11-using-unique_ptr-with-standard-library-containers#id10).

Which one to use depends on your exact needs, but I agree with Herb Sutter's [proposal](http://herbsutter.com/gotw/_103/) of using unique\_ptr by default and switching to shared\_ptr if the need arises.

### auto\_ptr Deprecated

(<https://stackoverflow.com/questions/2404115/is-auto-ptr-deprecated>) In C++0x std::auto\_ptr will be deprecated in favor of std::unique\_ptr. The choice of smart pointer will depend on your use case and your requirements, with std::unique\_ptr with move semantics for single ownership that can be used inside containers (using move semantics) and std::shared\_ptr when ownership is shared.

You should try to use the smart pointer that best fits the situation, choosing the correct pointer type provides other programmers with insight into your design

From the latest draft standard (n3035), section D.9

The class template auto\_ptr is deprecated. [ Note: The class template unique\_ptr(20.9.10) provides a better solution. —end note ]

Not only auto\_ptr is deprecated [in C++11 (D.10, page 1228)](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf), it will also be deleted [in a future version of C++](http://herbsutter.com/2014/11/24/updates-to-my-trip-report/):

Adopted N4190, and actually removed (not just deprecated) several archaic things from the C++ standard library, including **auto\_ptr**, bind1st/bind2nd, ptr\_fun/mem\_fun/mem\_fun\_ref, random\_shuffle, and a few more. Those are now all removed from the draft C++17 standard library and will not be part of future portable C++.

Another document about it: [Programming Language C++, Library Evolution Working Group - Document N4190](https://isocpp.org/files/papers/N4190.txt), if you want more information.

You can convert any code using auto\_ptr automaticaly, by using unique\_ptr instead:

Any code using auto\_ptr can be mechanically converted to using unique\_ptr, with move() inserted whenever auto\_ptr was being "copied".

#### Why is auto\_ptr deprecated?

It takes ownership of the pointer in a way that no two pointers should contain the same object. Assignment transfers ownership and resets the rvalue auto pointer to a null pointer. Thus, they can’t be used within STL containers due to the aforementioned inability to be copied.

### auto\_ptr

<http://www.gotw.ca/publications/using_auto_ptr_effectively.htm>

What *auto\_ptr* does is own a dynamically allocated object and perform automatic cleanup when the object is no longer needed.

Destructor will always be called during stack unwinding. The cleanup happens automatically.

Using an *auto\_ptr* is just about as easy as using a built-in pointer, and to "take back" the resource and assume manual ownership again, we just call release()

We can use *auto\_ptr*'s reset() function to reset the *auto\_ptr* to own a different object. If the *auto\_ptr* already owned an object, though, it first deletes the already-owned object, so calling reset() is much the same as destroying the *auto\_ptr* and creating a new one that owns the new object:

#### Copies are NOT equivalent.

It turns out that this has important effects when you try to use auto\_ptrs with generic code that does make copies and isn't necessarily aware that copies aren't equivalent (after all, usually copies are!). Consider the following code that I regularly see posted on the C++ newsgroups:

// Example 8: Danger, Will Robinson!

//

vector< auto\_ptr<T> > v;

/\* ... \*/

sort( v.begin(), v.end() );

**It is never safe to put auto\_ptrs into standard containers**. Some people will tell you that their compiler and library compiles this fine, and others will tell you that they've seen exactly this example recommended in the documentation of a certain popular compiler; don't listen to them.

The problem is that auto\_ptr does not quite meet the requirements of a type you can put into containers, because copies of auto\_ptrs are not equivalent. For one thing, there's nothing that says a vector can't just decide to up and make an "extra" internal copy of some object it contains. For another, when you call generic functions that will copy elements, like sort() does, the functions have to be able to assume that copies are going to be equivalent. At least one popular sort internally takes a copy of a "pivot" element, and if you try to make it work on auto\_ptrs it will merrily take a copy of the pivot auto\_ptr object (thereby taking ownership and putting it in a temporary auto\_ptr on the side), do the rest of its work on the sequence (including taking further copies of the now-non-owning auto\_ptr that was picked as a pivot value), and when the sort is over the pivot is destroyed and you have a problem: At least one auto\_ptr in the sequence (the one that was the pivot value) no longer owns the pointer it once held, and in fact the pointer it held has already been deleted!

So the standards committee bent over backwards to do everything it could to help you out: The Standard auto\_ptr was deliberately and specifically designed to break if you try to use it with the standard containers (or, at least, to break with most natural implementations of the standard library). To do this, the committee used a trick: auto\_ptr's copy constructor and copy assignment operator take references to non-const to the right-hand-side object. The standard containers' single-element insert() functions (or push\_back() function) take a reference to const, and hence won't work with auto\_ptrs.

<http://www.cplusplus.com/reference/std/memory/auto_ptr/>

This class provides a limited *garbage collection* facility for pointers, by allowing pointers to have the elements they point to automatically destroyed when the *auto\_ptr* object is itself destroyed.  
  
auto\_ptr objects have the peculiarity of *taking ownership* of the pointers assigned to them: An auto\_ptr object that has ownership over one element is in charge of destroying the element it points to and to deallocate the memory allocated to it when itself is destroyed. The destructor does this by calling operator delete automatically.  
  
Therefore, no two auto\_ptr objects should *own* the same element, since both would try to destruct them at some point. When an assignment operation takes place between two auto\_ptr objects, *ownership* is transferred, which means that the object losing ownership is reset to no longer point to the element (it is set to the *null pointer*).

#### Public members

|  |  |
| --- | --- |
| [**(constructor)**](http://www.cplusplus.com/reference/std/memory/auto_ptr/auto_ptr/) | Construct auto\_ptr object |
| [**(destructor)**](http://www.cplusplus.com/reference/std/memory/auto_ptr/%7Eauto_ptr/) | Destruct auto\_ptr |
| [**get**](http://www.cplusplus.com/reference/std/memory/auto_ptr/get/) | Get pointer |
| [**operator\***](http://www.cplusplus.com/reference/std/memory/auto_ptr/operator*/) | Dereference object |
| [**operator->**](http://www.cplusplus.com/reference/std/memory/auto_ptr/operator-%3E/) | Dereference object member |
| [**operator=**](http://www.cplusplus.com/reference/std/memory/auto_ptr/operator=/) | Release and copy auto\_ptr |
| [**release**](http://www.cplusplus.com/reference/std/memory/auto_ptr/release/) | Release pointer |
| [**reset**](http://www.cplusplus.com/reference/std/memory/auto_ptr/reset/) | Deallocate object pointed and set new value |
| [**(conversion operators)**](http://www.cplusplus.com/reference/std/memory/auto_ptr/operators/) | Conversion operators |

#### Constructors

explicit auto\_ptr (X\* p=0) throw();

auto\_ptr (auto\_ptr& a) throw();

template<class Y> auto\_ptr (auto\_ptr<Y>& a) throw();

auto\_ptr (auto\_ptr\_ref<X> r) throw();

Since auto\_ptr objects take ownership of the pointer they *point to*, when a new auto\_ptr is constructed from another auto\_ptr, the former owner *releases* it.

#include <iostream>

#include <memory>

using namespace std;

int main () {

auto\_ptr<int> p1 (new int);

\*p1.get()=10;

auto\_ptr<int> p2 (p1);

cout << "p2 points to " << \*p2 << "\n";

// (p1 is now null-pointer auto\_ptr)

return 0;

}

#### Operator =

auto\_ptr& operator= (auto\_ptr& a) throw();

template <class Y> auto\_ptr& operator= (auto\_ptr<Y>& a) throw();

auto\_ptr& operator= (auto\_ptr\_ref<X> r) throw();

Copies the value of the pointer held by a (or r).

The object on the left-hand side ***takes ownership*** of the pointer (i.e., it is now in charge of freeing the memory block when destroyed). Therefore, the *auto\_ptr* object on the right-hand side is automatically released (i.e., it is set to point to the null pointer) after the copy.

If the (left-hand side) object was being used to point to an object before the operation, the pointed object is destroyed (by calling operator delete).

#include <string>  
#include <iostream>  
#include <memory>  
using namespace std;  
   
class A{  
 string mName;  
public:  
 A(string name) { mName = name; }  
 ~A(){ cout << "~A" << mName << endl;  
 }   
};  
   
main(){  
 auto\_ptr<A> ap1(new A("p1") );  
 auto\_ptr<A> ap2(new A("p2") );  
 ap1.reset();  
 cout << "Here after reset" << endl;  
 ap1 = ap2;  
 cout << "Here after assignment" << endl;  
}

#### Reset

void reset (X\* p=0) throw();

Deallocate object pointed and set new value

Destructs the object pointed by the auto\_ptr object, if any, and deallocates its memory (by calling operator delete). If a value for *p* is specified, the internal pointer is initialized to that value (otherwise it is set to the *null pointer*).  
  
To only release the ownership of a pointer without destructing the object pointed by it, use member function [release](http://www.cplusplus.com/auto_ptr::release) instead.

|  |  |
| --- | --- |
| main()  {  auto\_ptr<A> ap1(new A("p1") );  auto\_ptr<A> ap2(new A("p2") );    ap1.reset(new A("p4"));  cout << "Here after reset" << endl;    ap2 = auto\_ptr<A>(new A("p3"));  } | Output  =====================  ~Ap1  Here after reset  ~Ap2  ~Ap3  ~Ap4 |

#### Release

X\* release() throw();

Release pointer

Sets the auto\_ptr internal pointer to *null pointer* (which indicates it points to no object) **without destructing the object** currently pointed by the auto\_ptr.  
  
To force a destruction of the object pointed, use member function [reset()](http://www.cplusplus.com/auto_ptr::reset) instead.  
  
The function returns a pointer to the object it pointed before the call, which is no longer its responsibility to destruct.

# Data Structure

## Tree

Here is a simple implementation for tree and nodes.

**struct Node{**

**int data;**

**Node \*left;**

**Node \*right;**

**};**

**void add\_edge(Node \*nodes[], int p, int left, int right)**

**{**

**if (left > 0) nodes[p-1]->left = nodes[left-1];**

**if (right > 0) nodes[p-1]->right = nodes[right-1];**

**}**

**void MLR(Node \*node)**

**{**

**if (node == NULL) return;**

**cout << node->data << endl;**

**MLR(node->left);**

**MLR(node->right);**

**}**

**void test\_0()**

**{**

**const int N = 5;**

**Node \*nodes[N];**

**for (int i=0; i<N; ++i)**

**nodes[i] = new Node(i);**

**add\_edge(nodes, 1, 2, 3);**

**add\_edge(nodes, 2, 4, 5);**

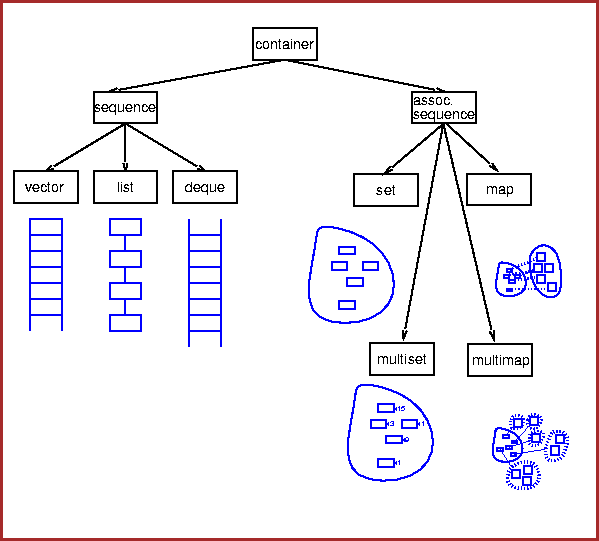
**MLR(nodes[0]);**

**for (int i=0; i<N; ++i)**

**delete nodes[i];**

**}**

# Containers

Containers are objects that conceptually contain other objects. They use certain basic properties of the objects (ability to copy, etc.) but otherwise do not depend on the type of object they contain.

STL containers may contain pointers to objects, though in this case you will need to do a little extra work.

vectors, lists, deques, sets, multisets, maps, multimaps, queues, stacks, and priority queues, *did I miss any?* are all provided.

Perhaps more importantly, built-in containers (C arrays) and user-defined containers can also be used as STL containers; this is generally useful when applying operations to the containers, e.g., sorting a container. Using user-defined types as STL containers can be accomplished by satisfying the requirements listed in the STL container requirements definition.

## Standard Containers

### Array (c++11)

<http://www.cplusplus.com/reference/array/array/>

<http://www.oracle.com/technetwork/articles/servers-storage-dev/c-array-containers-2252536.html>

Classical arrays have two issues: Fixed arrays decay into pointers, losing the array length information; and dynamic arrays have messy deallocation issues and are challenging to resize without error. Array and vector are to fix them.

The new *std::array* and *std::tuple* containers provide developers with additional ways to manage structured data efficiently.

static contiguous array. Properties:

* ***Sequence***

Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

* ***Contiguous storage***

The elements are stored in contiguous memory locations, allowing constant time random access to elements. Pointers to an element can be offset to access other elements.

* ***Fixed-size aggregate***

The container uses implicit constructors and destructors to allocate the required space statically. Its size is compile-time constant. No memory or time overhead.

Examples:

std::array<int, 5> myarray;

myarray = {0, 1, 2, 3, 4}; // okay

myarray = {9, 8, 7}; // okay, elements 3 and 4 are set to zero!

myarray = {0, 1, 2, 3, 4, 5}; // not allowed, too many elements in initializer list!

Member Functions:

|  |  |
| --- | --- |
| **Capacity** |  |
| [size](http://www.cplusplus.com/reference/array/array/size/) | Return size (public member function ) |
| [max\_size](http://www.cplusplus.com/reference/array/array/max_size/) | Return maximum size (public member function ) |
| [empty](http://www.cplusplus.com/reference/array/array/empty/) | Test whether array is empty (public member function ) |
|  |  |
| **Element access** | |
| [operator[]](http://www.cplusplus.com/reference/array/array/operator%5b%5d/) | Access element (public member function ) |
| [at](http://www.cplusplus.com/reference/array/array/at/) | Access element (public member function ) |
| [front](http://www.cplusplus.com/reference/array/array/front/) | Access first element (public member function ) |
| [back](http://www.cplusplus.com/reference/array/array/back/) | Access last element (public member function ) |
| [data](http://www.cplusplus.com/reference/array/array/data/) | Get pointer to data (public member function ) |
|  |  |
| **Modifiers** |  |
| [fill](http://www.cplusplus.com/reference/array/array/fill/) | Fill array with value (public member function ) |
| [swap](http://www.cplusplus.com/reference/array/array/swap/) | Swap content (public member function ) |

### Tuples (c++11)

<http://www.cplusplus.com/reference/tuple/>

A tuple is a generalization of a pair, an ordered set of heterogeneous elements. One way to imagine using a tuple is to hold a row of data in a database.

Tuples are objects that pack elements of -possibly- different types together in a single object, just like [pair](http://www.cplusplus.com/pair) objects do for pairs of elements, but generalized for any number of elements.  
  
Conceptually, they are similar to plain old data structures (C-like structs) but instead of having named data members, its elements are accessed by their order in the [tuple](http://www.cplusplus.com/tuple).  
  
The selection of particular elements within a [tuple](http://www.cplusplus.com/tuple) is done at the template-instantiation level, and thus, it must be specified at compile-time, with helper functions such as [get](http://www.cplusplus.com/get) and [tie](http://www.cplusplus.com/tie).  
  
The [tuple](http://www.cplusplus.com/tuple) class is closely related to the [pair](http://www.cplusplus.com/pair) class (defined in header [<utility>](http://www.cplusplus.com/%3Cutility%3E)): Tuples can be constructed from pairs, and pairs can be treated as tuples for certain purposes.  
  
[array](http://www.cplusplus.com/array) containers also have certain tuple-like functionalities.

|  |  |
| --- | --- |
| **Object creation** |  |
| [make\_tuple](http://www.cplusplus.com/reference/tuple/make_tuple/) | Construct tuple (function template ) |
| [forward\_as\_tuple](http://www.cplusplus.com/reference/tuple/forward_as_tuple/) | Forward as tuple (function template ) |
| [tie](http://www.cplusplus.com/reference/tuple/tie/) | Tie arguments to tuple elements (function template ) |
| [tuple\_cat](http://www.cplusplus.com/reference/tuple/tuple_cat/) | Concatenate tuples (function template ) |
|  |  |
| **Element access** |  |
| [get](http://www.cplusplus.com/reference/tuple/get/) | Get element (function template ) |

Example:

// tuple example

#include <iostream> // std::cout

#include <tuple> // std::tuple, std::get, std::tie, std::ignore

int main ()

{

std::tuple<int,char> foo (10,'x');

auto bar = std::make\_tuple ("test", 3.1, 14, 'y');

std::get<2>(bar) = 100; // access element

int myint; char mychar;

std::tie (myint, mychar) = foo; // unpack elements

std::tie (std::ignore, std::ignore, myint, mychar) = bar; // unpack (with ignore)

mychar = std::get<3>(bar);

std::get<0>(foo) = std::get<2>(bar);

std::get<1>(foo) = mychar;

std::cout << "foo contains: ";

std::cout << std::get<0>(foo) << ' ';

std::cout << std::get<1>(foo) << '\n';

return 0;

}

### Vector: dynamic contiguous array

Vectors are sequence containers representing arrays that can change in size.  
  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

## Associative Containers

<http://www.gibmonks.com/c_plus/ch23lev1sec3.html> The STL's associative containers provide direct access to store and retrieve elements via keys (often called search keys). The four associative containers are *multiset*, *set*, *multimap* and *map*.

Each associative container maintains its keys in **sorted** order. Iterating through an associative container traverses it in the sort order for that container.

Classes multiset and set provide operations for manipulating sets of values where the values are the keys: there is not a separate value associated with each key. The primary difference between a multiset and a set is that a multiset allows duplicate keys and a set does not.

Classes multimap and map provide operations for manipulating values associated with keys (these values are sometimes referred to as mapped values). The primary difference between a multimap and a map is that a multimap allows duplicate keys with associated values to be stored and a map allows only unique keys with associated values.

All associative containers also support several other member functions, including *find*, *lower\_bound*, *upper\_bound* and *count*.

### Set

<http://www.cplusplus.com/reference/set/set/>

Sets are containers that store unique elements following a specific order.  
  
In a set, the value of an element also identifies it (the value is itself the *key*, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.  
  
Internally, the elements in a set are always sorted following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/set::key_comp) (of type Compare).  
  
set containers are generally slower than [unordered\_set](http://www.cplusplus.com/unordered_set) containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.  
  
Sets are typically implemented as *binary search trees*.

|  |  |
| --- | --- |
| **Capacity**: |  |
| [empty](http://www.cplusplus.com/reference/set/set/empty/) | Test whether container is empty (public member function ) |
| [size](http://www.cplusplus.com/reference/set/set/size/) | Return container size (public member function ) |
| [max\_size](http://www.cplusplus.com/reference/set/set/max_size/) | Return maximum size (public member function ) |
| **Modifiers**: |  |
| [insert](http://www.cplusplus.com/reference/set/set/insert/) | Insert element (public member function ) |
| [erase](http://www.cplusplus.com/reference/set/set/erase/) | Erase elements (public member function ) |
| [swap](http://www.cplusplus.com/reference/set/set/swap/) | Swap content (public member function ) |
| [clear](http://www.cplusplus.com/reference/set/set/clear/) | Clear content (public member function ) |
| [emplace](http://www.cplusplus.com/reference/set/set/emplace/) | Construct and insert element (public member function ) |
| [emplace\_hint](http://www.cplusplus.com/reference/set/set/emplace_hint/) | Construct and insert element with hint (public member function ) |
| **Observers**: |  |
| [key\_comp](http://www.cplusplus.com/reference/set/set/key_comp/) | Return comparison object (public member function ) |
| [value\_comp](http://www.cplusplus.com/reference/set/set/value_comp/) | Return comparison object (public member function ) |
| **Operations**: |  |
| [find](http://www.cplusplus.com/reference/set/set/find/) | Get iterator to element (public member function ) |
| [count](http://www.cplusplus.com/reference/set/set/count/) | Count elements with a specific value (public member function ) |
| [lower\_bound](http://www.cplusplus.com/reference/set/set/lower_bound/) | Return iterator to lower bound (public member function ) |
| [upper\_bound](http://www.cplusplus.com/reference/set/set/upper_bound/) | Return iterator to upper bound (public member function ) |
| [equal\_range](http://www.cplusplus.com/reference/set/set/equal_range/) | Get range of equal elements (public member function ) |

### unordered\_set

Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.  
  
In an unordered\_set, the value of an element is at the same time its *key*, that identifies it uniquely. Keys are immutable, therefore, the elements in an unordered\_set cannot be modified once in the container - they can be inserted and removed, though.  
  
Internally, the elements in the unordered\_set are not sorted in any particular order, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *values* (with a constant average time complexity on average).  
  
unordered\_set containers are faster than [set](http://www.cplusplus.com/set) containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.  
  
Iterators in the container are at least [forward iterators](http://www.cplusplus.com/ForwardIterator).

### Map

Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order.  
  
In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value\_type, which is a [pair](http://www.cplusplus.com/pair) type combining both:

|  |  |  |
| --- | --- | --- |
|  | typedef pair<const Key, T> value\_type; |  |

Internally, the elements in a map are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/map::key_comp) (of type Compare).  
  
map containers are generally slower than [unordered\_map](http://www.cplusplus.com/unordered_map) containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.  
  
The mapped values in a [map](http://www.cplusplus.com/map) can be accessed directly by their corresponding key using the *bracket operator*(([operator[]](http://www.cplusplus.com/map::operator%5b%5d)).  
  
Maps are typically implemented as *binary search trees*.

|  |  |
| --- | --- |
| **Capacity**: |  |
| [empty](http://www.cplusplus.com/reference/map/map/empty/) | Test whether container is empty (public member function ) |
| [size](http://www.cplusplus.com/reference/map/map/size/) | Return container size (public member function ) |
| [max\_size](http://www.cplusplus.com/reference/map/map/max_size/) | Return maximum size (public member function ) |
|  |  |
| **Element access**: |  |
| [operator[]](http://www.cplusplus.com/reference/map/map/operator%5b%5d/) | Access element (public member function ) |
| [at](http://www.cplusplus.com/reference/map/map/at/) | Access element (public member function ) |
|  |  |
| **Modifiers**: |  |
| [insert](http://www.cplusplus.com/reference/map/map/insert/) | Insert elements (public member function ) |
| [erase](http://www.cplusplus.com/reference/map/map/erase/) | Erase elements (public member function ) |
| [swap](http://www.cplusplus.com/reference/map/map/swap/) | Swap content (public member function ) |
| [clear](http://www.cplusplus.com/reference/map/map/clear/) | Clear content (public member function ) |
| [emplace](http://www.cplusplus.com/reference/map/map/emplace/) | Construct and insert element (public member function ) |
| [emplace\_hint](http://www.cplusplus.com/reference/map/map/emplace_hint/) | Construct and insert element with hint (public member function ) |
|  |  |
| **Observers**: |  |
| [key\_comp](http://www.cplusplus.com/reference/map/map/key_comp/) | Return key comparison object (public member function ) |
| [value\_comp](http://www.cplusplus.com/reference/map/map/value_comp/) | Return value comparison object (public member function ) |
|  |  |
| **Operations**: |  |
| [find](http://www.cplusplus.com/reference/map/map/find/) | Get iterator to element (public member function ) |
| [count](http://www.cplusplus.com/reference/map/map/count/) | Count elements with a specific key (public member function ) |
| [lower\_bound](http://www.cplusplus.com/reference/map/map/lower_bound/) | Return iterator to lower bound (public member function ) |
| [upper\_bound](http://www.cplusplus.com/reference/map/map/upper_bound/) | Return iterator to upper bound (public member function ) |
| [equal\_range](http://www.cplusplus.com/reference/map/map/equal_range/) | Get range of equal elements (public member function ) |

### Multiple-key map

Multimaps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order, and where multiple elements can have equivalent keys.  
  
In a multimap, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values*store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type value\_type, which is a [pair](http://www.cplusplus.com/pair) type combining both:

|  |  |  |
| --- | --- | --- |
|  | typedef pair<const Key, T> value\_type; |  |

Internally, the elements in a multimap are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/multimap::key_comp) (of type Compare).  
  
multimap containers are generally slower than [unordered\_multimap](http://www.cplusplus.com/unordered_multimap) containers to access individual elements by their*key*, but they allow the direct iteration on subsets based on their order.  
  
Multimaps are typically implemented as *binary search trees*.

## Adaptor

Sometimes you have a class that does the right thing, but has the wrong interface for your purposes. Adaptors are classes that sit between you and another class, and translate the messages you want to send into the messages the other class wants to receive.

For example, the copy function expects an input iterator to get its data from. The istream class has the right functionality: it acts as a source of data, but it has the wrong interface: it uses << etc.

There is an adaptor called istream\_iterator that provides the interface that copy expects, translating requests into istream operations.

Other adaptors provide backward-moving iterators from forward-moving iterators, and queues from lists, for example.

### Container Adaptors

<http://www.gibmonks.com/c_plus/ch23lev1sec4.html>

The STL provides three container adapters: stack, queue and priority\_queue.

Adapters are **not first-class containers**, because they do **not provide the actual data-structure implementation** in which elements can be stored and because adapters do not support iterators.

The benefit of an adapter class is that the programmer can choose an appropriate underlying data structure. All three adapter classes provide member functions push and pop that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure.

### Stack

A stack can be implemented with any of the sequence containers: *vector*, *list* and *deque*. By default, a stack is implemented with a deque.

Performance Tip:

1. Each of the common operations of a stack is implemented as an inline function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.
2. For the best performance, use class deque or vector as the underlying container for a stack.

Examples:

1. // stack with default underlying deque

std::stack< int > intDequeStack;

1. // stack with underlying vector

std::stack< int, std::vector< int > > intVectorStack;

1. // stack with underlying list

std::stack< int, std::list< int > > intListStack;

<http://www.cplusplus.com/reference/stack/stack/>

**LIFO stack**

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.  
  
**stack**s are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed*/*popped* from the *"back"* of the specific container, which is known as the *top* of the stack.  
  
The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

* empty
* size
* back
* push\_back
* pop\_back

The standard container classes [vector](http://www.cplusplus.com/vector), [deque](http://www.cplusplus.com/deque) and [list](http://www.cplusplus.com/list) fulfill these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container [deque](http://www.cplusplus.com/deque) is used.

Member functions

|  |  |
| --- | --- |
| [(constructor)](http://www.cplusplus.com/reference/stack/stack/stack/) | Construct stack (public member function ) |
| [empty](http://www.cplusplus.com/reference/stack/stack/empty/) | Test whether container is empty (public member function ) |
| [size](http://www.cplusplus.com/reference/stack/stack/size/) | Return size (public member function ) |
| [top](http://www.cplusplus.com/reference/stack/stack/top/) | Access next element (public member function ) |
| [push](http://www.cplusplus.com/reference/stack/stack/push/) | Insert element (public member function ) |
| [emplace](http://www.cplusplus.com/reference/stack/stack/emplace/) | Construct and insert element (public member function ) |
| [pop](http://www.cplusplus.com/reference/stack/stack/pop/) | Remove top element (public member function ) |
| [swap](http://www.cplusplus.com/reference/stack/stack/swap/) | Swap contents (public member function ) |

Why pop does not return a value?

<http://stackoverflow.com/questions/25035691/why-doesnt-stdqueuepop-return-value>

It could indeed have done the same thing. The reason it didn't, is because a pop that returned the popped element is unsafe in the presence of exceptions (having to return by value and thus creating a copy).

### Queue

<http://www.cplusplus.com/reference/queue/queue/>

**queue**s are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.  
  
**queue**s are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the *"back"* of the specific container and *popped* from its *"front"*.  
  
The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

* empty
* size
* front
* back
* push\_back
* pop\_front

The standard container classes [deque](http://www.cplusplus.com/deque) and [list](http://www.cplusplus.com/list) fulfill these requirements. By default, if no container class is specified for a particular queue class instantiation, the standard container [deque](http://www.cplusplus.com/deque) is used.

### Double ended queue

**deque** (usually pronounced like *"deck"*) is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).  
  
Specific libraries may implement *deques* in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.  
  
Therefore, they provide a functionality similar to [vectors](http://www.cplusplus.com/vector), but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike [vectors](http://www.cplusplus.com/vector), [deques](http://www.cplusplus.com/deque) are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes *undefined behavior*.  
  
Both [vectors](http://www.cplusplus.com/vector) and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than [vectors](http://www.cplusplus.com/vector), but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.  
  
For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than [lists](http://www.cplusplus.com/list) and [forward lists](http://www.cplusplus.com/forward_list).

### Priority Queue

Class priority\_queue provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. A priority\_queue can be implemented with STL sequence containers *vector* or *deque*. By default, a priority\_queue is implemented with a vector as the underlying container.

When elements are added to a priority\_queue, they are inserted in priority order, such that the highest-priority element (i.e., the largest value) will be the first element removed from the priority\_queue. This is usually accomplished via a sorting technique called heapsort that always maintains the largest value (i.e., highest-priority element) at the front of the data structuresuch a data structure is called a heap. The comparison of elements is performed with comparator function object less< T > by default, but the programmer can supply a different comparator.

### Deque vs vector

<http://www.gotw.ca/gotw/054.htm>; <http://www.cplusplus.com/reference/stl/deque/>

Deques may be implemented by specific libraries in different ways, but in all cases they allow for the individual elements to be accessed through random access iterators, with storage always handled automatically (expanding and contracting as needed).

Deque sequences have the following properties:

\* Individual elements can be accessed by their position index.

\* Iteration over the elements can be performed in any order.

\* Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

Therefore they provide a ***similar functionality as the one provided by vectors***, but with ***efficient insertion and deletion*** of elements also at the beginning of the sequence and not only at its end. On the drawback side, unlike vectors, *deques are not guaranteed to have all its elements in contiguous storage locations, eliminating thus the possibility of safe access through pointer arithmetics*.

Both vectors and deques provide thus a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors are very similar to a plain array that grows by *reallocating all of its elements in a unique block when its capacity is exhausted*, the elements of a deques can be divided in *several chunks of storage*, with the class keeping all this information and providing a uniform access to the elements. Therefore, deques are a little more complex internally, but this generally allows them to grow more efficiently than the vectors with their capacity managed automatically, specially in large sequences, because massive reallocations are avoided.

## Iterator

<http://www.cplusplus.com/reference/iterator/>

An *iterator* is any object that, pointing to some element in a range of elements (such as an array or a [container](http://www.cplusplus.com/stl)), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (\*) operators).  
  
The most obvious form of iterator is a *pointer*: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other kinds of iterators are possible. For example, each [container](http://www.cplusplus.com/stl) type (such as a [list](http://www.cplusplus.com/list)) has a specific *iterator* type designed to iterate through its elements.  
  
Notice that while a pointer is a form of iterator, not all iterators have the same functionality of pointers; Depending on the properties supported by iterators, they are classified into five different categories:

* [Input](http://www.cplusplus.com/InputIterator) and [output](http://www.cplusplus.com/OutputIterator) iterators are the most limited types of iterators: they can perform sequential single-pass input or output operations.
* [Forward iterators](http://www.cplusplus.com/ForwardIterator) have all the functionality of [input iterators](http://www.cplusplus.com/InputIterator) and -if they are not *constant iterators*- also the functionality of [output iterators](http://www.cplusplus.com/OutputIterator), although they are limited to one direction in which to iterate through a range (forward). All [standard containers](http://www.cplusplus.com/stl) support at least forward iterator types.
* [Bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator) are like [forward iterators](http://www.cplusplus.com/ForwardIterator) but can also be iterated through backwards.
* [Random-access iterators](http://www.cplusplus.com/RandomAccessIterator) implement all the functionality of [bidirectional iterators](http://www.cplusplus.com/BidirectionalIterator), and also have the ability to access ranges non-sequentially: distant elements can be accessed directly by applying an offset value to an iterator without iterating through all the elements in between. These iterators have a similar functionality to standard pointers (pointers are iterators of this category).

Iterators are central to generic programming because they are an **interface between containers and algorithms**: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators.

### forward\_iterator\_tag

<http://www.cplusplus.com/reference/iterator/ForwardIterator/>

struct forward\_iterator\_tag {};

Empty class to identify the category of an iterator as a *forward iterator.*

Support -at least- the following operations:

* Is [*default-constructible*](http://www.cplusplus.com/DefaultConstructible)*,*[*copy-constructible*](http://www.cplusplus.com/CopyConstructible)*,*[*copy-assignable*](http://www.cplusplus.com/CopyAssignable)*and*[*destructible*](http://www.cplusplus.com/Destructible)

X a; X b(a); b = a;

* Can be compared for equivalence using the equality/inequality operators  
  (meaningful when both iterator values iterate over the same underlying sequence).

a == b; a != b;

* Can be dereferenced as an *rvalue* (if in a *dereferenceable state*).

\*a; a->m;

* For *mutable iterators* (*non-constant iterators*):  
  Can be dereferenced as an *lvalue* (if in a *dereferenceable state*).

\*a = t;

* Can be incremented (if in a *dereferenceable state*).  
  The result is either also *dereferenceable* or a *past-the-end* iterator.  
  Two iterators that compare equal, keep comparing equal when both are increased.

++a; a++; \*a++;

### Reverse Iterator

int main(){

vector<int> l;

for (int i=0; i<10; ++i) l.push\_back( i );

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

l.erase(find(l.begin(), l.end(), 3));

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

l.insert(find(l.begin(), l.end(), 6), 30);

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

}

Output is:

0 1 2 3 4 5 6 7 8 9

0 1 2 4 5 6 7 8 9

0 1 2 4 5 30 6 7 8 9

Compare with the following reverse iterators:

int main(){

vector<int> l;

for (int i=0; i<10; ++i) l.push\_back( i );

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

l.erase(find(l.rbegin(), l.rend(), 3).base());

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

l.insert(find(l.rbegin(), l.rend(), 6).base(), 30);

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

}

Output is:

0 1 2 3 4 5 6 7 8 9

0 1 2 3 5 6 7 8 9

0 1 2 3 5 6 30 7 8 9

### back\_inserter

Use an adaptor like back\_inserter on an empty vector. Back\_inserter indirectly invokes a push\_back on the container passed to it. The code would look like this:

int main()

{

vector<int> Num;

generate\_n(back\_inserter(Num), 10, Sequence) ;

**copy(Num.begin(), Num.end(), ostream\_iterator<int>(cout, " "));**

return 0;

}

Back\_inserter can be used on any container on which push\_back is defined. There is also a front\_inserter which can be used on any container on which push\_front is defined.

### Iterator Adaptor

# Algorithm

### Count

template<class InIter, class T>

size\_t **count**(InIter start, InIter end, const T &val);

template<class InIter,class UnPred>

size\_t **count\_if**(InIter start, InIter end, UnPred pfn);

example:

for (i=0;i<10;++i) if (rand()%2) v.push\_back(true);

else v.push\_back(false);

count(v.begin(), v.end(), true);

example:

bool divisibleBy3(int i){

if (i%3==0) return true; else return false;

}

for (i=0; i<20; ++i) v.push\_back(i);

count\_if(v.begin(), v.end(), divisibleBy3);

### Remove & Replace Elements

To remove or replace some elements, and put results in the given iterator.

template<class InIter, class OutIter, class T>

OutIter **remove\_copy** (InIter start, InIter end,

OutIter result, const T &val);

template<class InIter, class OutIter, class T>

OutIter **replace\_copy** (InIter start, InIter end,

OutIter result, const T &old, const T &new);

template <class ForwardIterator, class Predicate>

ForwardIterator **remove\_if**(ForwardIterator first,

ForwardIterator last, Predicate pred);

example:

char str[] = “The STL is power programming.”;

vector<char> v, v2(30);

for (i=0; str[i]; ++i) v.push\_back(str[i]);

remove\_copy(v.begin(), v.end(), v2.begin(), ‘ ‘);

replace\_copy(v.begin(),v.end(), v2.begin(), ‘ ‘, ‘:’);

remove\_if(v.begin(), v.end(), bind2nd(greater<int>(), 8));

### Reverse a Sequence

template<class BiIter> void **reverse**(BiIter start, BiIter end);

example:

for (i=0; i<10; ++i) v.push\_back(i);

reverse(v.begin(), v.end());

### Transform

Transform applies a specified function object to each element in a source range or to a pair of elements from two source ranges and copies the return values of the function object into a destination range.

It is an overloaded function with the following flavors:

template<class InIter, class OutIter, class Func>

OutIter transform(InIter start, InIter end,

OutIter result, Func unaryfunc);

template<class InIter1, class InIter2, class OutIter, class Func>

OutIter transform(InIter1 start1, InIter1 end1,

InIter2 start2, OutIter result, Func binaryfunc);

example:

double reciprocal(double i){

return 1.0/i;

}

for (i=1; i<10; ++i) v.push\_back( (double)i);

for (i=1; i<10; ++i) x.push\_back( 3.0 );

transform(v.begin(), v.end(), v.begin(), reciprocal);

transform(v.begin(), v.end(), v.begin(), negate<double>());

transform(v.begin(), v.end(), x.begin(),

v.begin(),divides<double>());

### Copy

Assigns the same new value to every element in a specified range.

The following example writes to cout

**#include <iterator>**

**#include <algorithm>**

**copy(Num.begin(), Num.end(), ostream\_iterator<int>(cout, " "));**

### Sort – uses introsort

Earlier versions of sort used the quicksort algorithm (C. A. R. Hoare, Comp. J. 5, 1962), using a pivot chosen by median of three (R. C. Singleton, CACM 12, 1969). Quicksort has O(N log(N)) average complexity, but quadratic worst-case complexity. The current implementation of sort, however, uses the **introsort** algorithm (D. R. Musser from Rensselaer Poytechnic Institute, "Introspective Sorting and Selection Algorithms", Software Practice and Experience 27(8):983, 1997.) whose worst case complexity is O(N log(N)). Introsort is very similar to median-of-three quicksort, and is at least as fast as quicksort on average. (<http://www.cs.rpi.edu/~chapaa/stl/>)

The average of a sort complexity is O(N log N)

#### Sort a List in Descending Order

int main(){

list<int> l;

for (int i=0; i<10; ++i) l.push\_back(std::rand() % 10);

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

l.sort( **greater**<**int**>() );

copy(l.begin(), l.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

}

Use less<int>() or void will sort the list in ascending order

You cannot use general sort to sort a list, because it requires **RandomAccessIterator**

#### sort vs qsort

<https://isocpp.org/wiki/faq/c#c-diffs>

To a novice,

qsort(array,asize,sizeof(elem),elem\_compare);

looks pretty weird, and is harder to understand than

sort(vec.begin(),vec.end());

To an expert, the fact that sort() tends to be faster than qsort() for the same elements and the same comparison criteria is often significant. Also, sort() is generic, so that it can be used for any reasonable combination of container type, element type, and comparison criterion. For example:

struct Record {

string name;

*// ...*

};

struct name\_compare { *// compare Records using "name" as the key*

bool operator()(const Record& a, const Record& b) const

{ return a.name<b.name; }

};

void f(vector<Record>& vs)

{

sort(vs.begin(), vs.end(), name\_compare());

*// ...*

}

In addition, most people appreciate that sort() is type safe, that no casts are required to use it, and that they don’t have to write a compare() function for standard types.

For a more detailed explanation, see Stroustrup’s paper “Learning C++ as a New language”, which can be downloaded from [his publications list](http://stroustrup.com/papers.html).

The primary reason that sort() tends to outperform qsort() is that the comparison inlines better.

### Partial\_sort – uses heapsort

Arranges a specified number of the smaller elements in a range into a nondescending order or according to an ordering criterion specified by a binary predicate.

More informally, this means that it rearranges the elements in the range [first, last) so that they are partially in ascending order. Specifically, it places the smallest middle - first elements, sorted in ascending order, into the range [first, middle). The remaining last - middle elements are placed, in an unspecified order, into the range [middle, last)

The sort algorithm is not stable and does not guarantee that the relative ordering of equivalent elements will be preserved. The algorithm stable\_sort does preserve this original ordering.

Partial\_sort uses **heapsort** to do the sorting.

### Generate

A Generator is a kind of [function object](http://www.sgi.com/tech/stl/functors.html): an object that is called as if it were an ordinary C++ function. A Generator is called with no arguments.

The generate algorithm assigns element values in a specified range by a Generator or a Function.

*template<class ForwardIterator, class Generator>*

*void generate(ForwardIterator First, ForwardIterator Last, Generator Gen);*

*Gen is any function object that returns a value to be assigned at the position addressed by the forward iterator.*

The range referenced must be valid; all pointers must be dereferenceable and within the sequence the last position is reachable from the first by incrementation. That is, [First, Last) is a valid range.

The complexity of the algorithm is linear. There are exactly Last - First invocations of gen.

Algorithm generate essentially generates values at a specified range starting and ending at the position addressed by the iterators. The value is generated by the Generator class. In order to practice the use of generate, we are assumed that we want to populate values in a container.

For example, we want to populate an integer vector with 10 elements with a value from 0 to 9. It is not hard to think about a couple lines of code that creates an integer vector with 20 elements and then creates a for loop to assign each integer value to each element in the vector. However this does not utilize the power of generic programming.

The very major design issue of STL that is taken into consideration is to make code more generic. In this example, a code for generating values for the vector can be separated to a function. The function that generates the values can be as simple as our example, returning an incremented value each time it is called. The function can be very complex as a very long encryption code for military purpose. No matter which function being used the code keeps very simple, the power of generic programming.

After a function is defined and container is created, generate can be called. The first parameter is an iterator referencing to the first element wanted to populate. The second parameter is also an iterator but referencing to the element next to last element wanted to populate. The last parameter is a Generator class or function object that will return a value each time it is called.

Example: to fill a vector with random numbers

vector<int> v(100);

generate(v.begin(), v.end(), rand);

### Fill

Assigns the same new value to every element in a specified range.

It assigns the value \_Val to every element in the range [\_First, last). That is, for every iterator i in [\_First, \_Last), it performs the assignment \*i = \_Val.

*template<class ForwardIterator, class Type>*

*void fill(ForwardIterator \_First, ForwardIterator \_Last, const Type& \_Val);*

# Function Objects

### Overview

(<https://docs.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl?view=vs-2017>) A *function object*, or *functor*, is ***any type that implements operator().*** This operator is referred to as the *call operator* or sometimes the *application operator*. The C++ Standard Library uses function objects primarily as sorting criteria for containers and in algorithms.

Function objects provide two main advantages over a straight function call. The first is that a function object can contain state. The second is that a function object is a type and therefore can be used as a template parameter.

### Build-in Function Objects – <functional>

For example, to invoke the binary function object plus(), use this syntax

**plus<float>() – default constructor to create an “object”**

Built-in unary function objects

|  |  |
| --- | --- |
| logical\_not | Negate |

Build-in binary function objects

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Plus | minus | multiple | divides | modules |
| equal\_to | greater | greater\_equal | logical\_and |  |
| not\_equal\_to | less | less\_equal | logical\_or |  |

### Creating a Function Object

You will simply need to create a class that overloads the **operator()** function.

For the greatest flexibility, you will want to use one of the following classes defined by STL as a base class for your function objects

template<class Argument, class Result>

struct unary\_function{ // **use not1() to negate it**

typedef Argument argument\_type;

typedef Result result\_type;

}

template<class Argument1, class Argument2, class Result>

struct binary\_function{ // **use not2() to negate it**

typedef Argument1 first\_argument\_type;

typedef Argument2 second\_argument\_type;

typedef Result result\_type;

}

example:

class reciprocal:unary\_function<double, double>{

public:

result\_type operator()(argument\_type i)

{

return (result\_type) 1.0/i;

}

}

## Binder: bind1st, bind2nd

bind1st(binfunc\_obj, value);

bind2nd(binfunc\_obj, value);

**remove\_if(v.begin(), v.end(), bind2nd(greater<int>(), 8));**

# Interview Questions

## STL Q&A

### Why does std::stack use std::deque by default?

<http://stackoverflow.com/questions/102459/why-does-stdstack-use-stddeque-by-default>

As the container grows, a reallocation for a vector requires copying all the elements into the new block of memory. Growing a deque allocates a new block and links it to the list of blocks - no copies are required.

### Q: auto\_ptr?

std::auto\_ptr is a template class that holds a pointer, and deallocates it when it goes out of scope.

01.#include <memory> // for std::auto\_ptr

02.try

03.{

04.    pJohn = new Person("John", 18, E\_MALE);

05.    auto\_ptr<Person> pxJohn(pJohn); // pxJohn now owns pJohn

06.

07.    ProcessPerson(pJohn);

08.

09.    // when pxJohn goes out of scope, it will delete pJohn

10.}

11.catch (PersonException &cException)

12.{

13.    cerr << "Failed to process person: " << cException.what() << endl;

14.}

Note that std::auto\_ptr should not be set to point to arrays. This is because it uses the delete operator to clean up, not the delete[] operator. In fact, there is no array version of std::auto\_ptr! It turns out, there isn’t really a need for one. In the standard library, if you want to do dynamically allocated arrays, you’re supposed to use the std::vector class, which will deallocate itself when it goes out of scope

## Namespace

1. There maybe more than one namespace declaration of the same name. This allows a namespace to be split over several files, or even separated in the same file.
2. Unnamed namespace allow to establish unique identifier that are known

## Mutable

A mutable member variable can be modified by a const member function.

## Volatile

A member function marked volatile causes **this** pointer to be treated as a volatile pointer.

## Array-based IO – deprecated

#include <strstream>

## Elaborated type specifier

Elaborated type specifiers may be used to refer to a previously-declared class name (class, struct, or union) or to a previously-declared enum name even if the name was [hidden by a non-type declaration](https://en.cppreference.com/w/cpp/language/lookup). They may also be used to declare new class names.

class T {

public:

class U;

private:

int U;

};

int main()

{

int T;

T t; // error: the local variable T is found

class T t; // OK: finds ::T, the local variable T is ignored

T::U\* u; // error: lookup of T::U finds the private data member

class T::U\* u; // OK: the data member is ignored

}

## Q&A

### Q: reference vs pointers

A reference is a variable which refers to another variable, it is an alias of another variable.

<https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html>

<http://www.cplusplus.com/articles/ENywvCM9/>

1. No null reference, but there are null pointers. The results are undefined for the case below:

char char \*pc = 0; // set pointer to null

char& rc = \*pc; // make reference refer to

// dereferenced null pointer

A reference must always refer to some object. As a result, if you have a variable whose purpose is to refer to another object, but it is possible that there might not be an object to refer to, you should make the variable a pointer, because then you can set it to null. On the other hand, if the variable must always refer to an object, i.e., if your design does not allow for the possibility that the variable is null, you should probably make the variable a reference.

1. References must be initialized, because a reference must refer to an object.

string& rs; // error! References must

// be initialized

string s("xyzzy");

string& rs = s; // okay, rs refers to s

1. References are strongly typed.
2. Reference may be more efficient, because there's no need to test the validity of a reference before using it.
3. Reference cannot be reassigned, it always refers to the object with which it is initialized.
4. There's no "reference arithmetic" (but you can take the address of an object pointed by a reference and do pointer arithmetic on it as in &obj + 5)
5. You can't take the address of a reference like you can with pointers.

To refer to an array:

int array[10];

int (&array\_ref)[10] = array;

### Q: Array size for member and non-member variables

const int M = 10;

class myclass{

private:

static const int N;

int array[myclass::N];

};

const int myclass::N = 100;

**compiler error**: ‘variable length array in structure’ will never be supported.

int global\_array[M];

class myclass{

private:

static const int N = 100;

int array[myclass::N];

};

### Q: Create a function which can write to both file and standard output

A: Jut use **ostream** as the argument. E.g: void myoutput(ostream &out);

### Q: link a C++ program to C functions?

A: By using the extern "C" linkage specification around the C function declarations.

### Q: Explain the scope resolution operator.

A: It permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

Your answer was egregiously over-limiting, since the 99% of the time the scope resolution operator is \*not\* used to access a variable in global scope. More often it is used to access static class members and types. See the “scope” entry in the index in Stroustrup’s C++ 3rd edition, and notice the subheading “resolution operator” points to page 228, where this by far most common usage is exampled

### Q: What are the differences between a C++ struct and C++ class?

A: The default member and base-class access specifiers are different.

Members of a class are private by default, whereas members of a struct are public by default.

Inheritance between classes is also private by default, and inheritance between structs is public by default.

### Q: How many ways are there to initialize an int with a constant?

A: Two.

There are two formats for initializers in C++ as shown in the example that follows. The first format uses the traditional C notation. The second format uses constructor notation.

int foo = 123;

int bar (123);

### Q: How does throwing and catching exceptions differ from using setjmp and longjmp?

A: The throw operation calls the destructors for automatic objects instantiated since entry to the try block.

### Q: Delete this: What is your reaction to this line of code?

A: It’s not a good practice.

### Q: What is a default constructor?

A: A constructor that has no arguments.

### Q: What is a conversion constructor?

A: A constructor that accepts one argument of a different type.

Q: What is the difference between a copy constructor and an overloaded assignment operator?

A: A copy constructor constructs a new object by using the content of the argument object. An overloaded assignment operator assigns the contents of an existing object to another existing object of the same class.

### Q: When should you use multiple inheritance?

A: There are three acceptable answers: "Never," "Rarely," and "When the problem domain cannot be accurately modeled any other way."

### Q: What is a virtual destructor?

A: The simple answer is that a virtual destructor is one that is declared with the virtual attribute.

### Q: When need a virtual destructor?

A:

### Q: Exceptions and destructors

Unlike constructors, where throwing exceptions can be a useful way to indicate that object creation succeeded, exceptions should not be thrown in destructors.

The problem occurs when an exception is thrown from a destructor during the stack unwinding process. If that happens, the compiler is put in a situation where it doesn’t know whether to continue the stack unwinding process or handle the new exception. The end result is that your program will be terminated immediately.

Consequently, the best course of action is just to abstain from using exceptions in destructors altogether. Write a message to a log file instead.

### Q: Inheritance specifier?

Basic thumb rule in inheritance. You cannot inherit private members.

Public inheritance: keep their original access specifications

inherit protected members,methods of base class with visibilty modifier as protected.  
inherit public members,methods of base class with visibilty modifier as public.

Private inheritance: all members from the base class are inherited as private

inherit protected as well as public members,methods of base class with visibilty modifier as private

Protected inheritance:

public and protected members become protected

private members stay private.

### Q: Explain the ISA and HASA class relationships. How would you implement each in a class design?

A: A specialized class "is" a specialization of another class and, therefore, has the ISA relationship with the other class. An Employee ISA Person. This relationship is best implemented with inheritance. Employee is derived from Person. A class may have an instance of another class. For example, an employee "has" a salary, therefore the Employee class has the HASA relationship with the Salary class. This relationship is best implemented by embedding an object of the Salary class in the Employee class.

### Q: When is a template a better solution than a base class?

A: When you are designing a generic class to contain or otherwise manage objects of other types, when the format and behavior of those other types are unimportant to their containment or management, and particularly when those other types are unknown (thus, the genericity) to the designer of the container or manager class.

### Q: What is a mutable member?

A: One that can be modified by the class even when the object of the class or the member function doing the modification is const.

### Q: What is an explicit constructor?

A: A conversion constructor declared with the explicit keyword. The compiler does not use an explicit constructor to implement an implied conversion of types. It’s purpose is reserved explicitly for construction.

### Q: What is the Standard Template Library?

A: A library of container templates approved by the ANSI committee for inclusion in the standard C++ specification.

A programmer who then launches into a discussion of the generic programming model, iterators, allocators, algorithms, and such, has a higher than average understanding of the new technology that STL brings to C++ programming.

### Q: Describe run-time type identification.

A: The ability to determine at run time the type of an object by using the typeid operator or the dynamic\_cast operator.

### Q: What problem does the namespace feature solve?

A: Multiple providers of libraries might use common global identifiers causing a name collision when an application tries to link with two or more such libraries. The namespace feature surrounds a library’s external declarations with a unique namespace that eliminates the potential for those collisions.

This solution assumes that two library vendors don’t use the same namespace identifier, of course.

### Q: Are there any new intrinsic (built-in) data types?

A: Yes. The ANSI committee added the bool intrinsic type and its true and false value keywords.

Download [Teach Yourself C++ in 21 days for free](http://hellnet.perverz.hu/ebookz/Teach%20Yourself%20series/Teach%20Yourself%20C++%20In%2021%20Days.Zip)

## Shallow/Deep/Lazy Copy

<http://www.fredosaurus.com/notes-cpp/oop-condestructors/shallowdeepcopy.html>

A ***shallow copy*** of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied. but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.

A ***deep copy*** copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disasterous consequences.

Deep copies need ...

If an object has pointers to dynamically allocated memory, and the dynamically allocated memory needs to be copied when the original object is copied, then a deep copy is required.

A class that requires deep copies generally needs:

\* A constructor to either make an initial allocation or set the pointer to NULL.

\* A destructor to delete the dynamically allocated memory.

\* A copy constructor to make a copy of the dynamically allocated memory.

\* An overloaded assignment operator to make a copy of the dynamically allocated memory.

### Lazy Copy

<http://en.wikipedia.org/wiki/Object_copy> A lazy copy is a combination of both strategies above. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify an object, it can determine if the data is shared (by examining the counter) and can do a deep copy if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can also cause problems.

Lazy copy is related to copy-on-write.

### Other notes

The default copy constructor does a member-wise copy of an object. For our MyString

class, the default copy constructor will copy the length integer and the characters

pointer. However, the characters themselves are not copied. This is called a **shallow**

**copy**, because it only copies the data one level deep in a class hierarchy.

When would you use it ?

You will use deep copy when you have dynamically allocated members in a class. Do not forget to mention that you need a corresponding destructor to deallocate.

## Bit Operation

### What’s potentially wrong with the following code?

long value;

//some stuff

value &= 0xFFFF;

Note: Hint to the candidate about the base platform they’re developing for. If the person still doesn’t find anything wrong with the code, they are not experienced with C++. [<http://www.devbistro.com/tech-interview-questions/Cplusplus.jsp>]

[<http://stackoverflow.com/questions/4251946/what-is-wrong-with-this-bit-manipulation-code-from-an-interview-question>]

A hexadecimal literal is represented by the first of the following types that is large enough to contain it: int, unsigned int, long, and unsigned long.

If int has a width of 16 bits, then 0xFFFF is larger than the maximum value representable by an int. Thus, 0xFFFF is of type unsigned int, which is guaranteed to be large enough to represent 0xFFFF.

When the usual arithmetic conversions are performed for evaluation of the &, the unsigned int is converted to a long. The conversion of a 16-bit unsigned int to long is well-defined because every value representable by a 16-bit unsigned int is also representable by a 32-bit long.

There's no sign extension needed because the initial type is not signed, and the result of using 0xFFFF is the same as the result of using 0xFFFFL.

Alternatively, if int is wider than 16 bits, then 0xFFFF is of type int. It is a signed, but positive, number. In this case both operands are signed, and long has the greater conversion rank, so the int is again promoted to long by the usual arithmetic conversions.

As others have said, you should avoid performing bitwise operations on signed operands because the numeric result is dependent upon how signedness is represented.

Aside from that, there's nothing particularly wrong with this code. I would argue that it's a style concern that value is not initialized when it is declared, but that's probably a nit-pick level comment and depends upon the contents of the //some stuff section that was omitted.

It's probably also preferable to use a fixed-width integer type (like uint32\_t) instead of long for greater portability, but really that too depends on the code you are writing and what your basic assumptions are.

## Instantiate objects without default Constructor

For class

class Myclass{

public:

Myclass(int a){m=a;};

private :

int m;

};

* 1. how will u create an array of objects of Myclass on stack?

int i[] = {1,2,3,4,5,6,7,8,9,10};

MyClass myClass[10] = {1,2,3,4,5,6,7,8,9,10};

* 1. how will u crete an array of objects on heap?

MyClass\* mc[10];  
for (int i=0; i < 10; i++)

mc[i] = new MyClass(i); <--- it will call user-defined constructor here.

# Reference:

1. The Annotated C++ Refernce Manual, M Ellis, B Stroustrup, 1990 AT&T
2. <http://www.cplusplus.com/doc/tutorial/typecasting/>
3. <http://www.parashift.com/c++-faq-lite/ctors.html>