Stackoverflow Q&A

Contents

[1. Abstraction, Information Hiding and Encapsulation 2](#_Toc393815618)

[i. Difference between abstraction and encapsulation? 2](#_Toc393815619)

[ii. Abstraction VS Information Hiding VS Encapsulation 2](#_Toc393815620)

[2. Initialization & Cleanup 4](#_Toc393815621)

[i. What does the explicit keyword in C++ mean? 4](#_Toc393815622)

[ii. Do the parentheses after the type name make a difference with new? 5](#_Toc393815623)

[iii. C++ call constructor from constructor 8](#_Toc393815624)

[iv. C++ superclass constructor calling rules 9](#_Toc393815625)

[v. Why do we not have a virtual constructor in C++? 9](#_Toc393815626)

[vi. Default constructor with empty brackets 10](#_Toc393815627)

[vii. Inheriting constructors 10](#_Toc393815628)

[viii. Why must const members be initialized in the constructor initializer rather than in its body? 11](#_Toc393815629)

[ix. What are Aggregates and PODs and how/why are they special? 11](#_Toc393815630)

[x. Is there an implicit default constructor in C++? 18](#_Toc393815631)

[xi. C++: constructor initializer for arrays 19](#_Toc393815632)

[xii. Conditions for automatic generation of default ctor, copy ctor, and default assignment operator? 20](#_Toc393815633)

[xiii. What is a non-trivial constructor in C++? 20](#_Toc393815634)

[xiv. Do I need to explicitly call the base virtual destructor? 21](#_Toc393815635)

[xv. What is the use of having destructor as private? 21](#_Toc393815636)

[xvi. What is the use of “delete this”? 22](#_Toc393815637)

[xvii. How does the C++ runtime system know when objects go out of scope? 22](#_Toc393815638)

[xviii. Why do we need a pure virtual destructor in C++? 22](#_Toc393815639)

[xix. Will an 'empty' destructor do the same thing as the generated destructor? 24](#_Toc393815640)

[xx. Object destruction in C++? 24](#_Toc393815641)

## Abstraction, Information Hiding and Encapsulation

### Difference between abstraction and encapsulation?

* A priori, they've got nothing in common.

Encapsulation:

* Most answers here focus on OOP but encapsulation begins much earlier; **every method is an encapsulation**:
* Here, distance encapsulates the calculation of the (euclidean) distance between two points in a plane: it hides implementation details. This is encapsulation, pure and simple.

point x = { 1, 4 };

point y = { 23, 42 };

int d = distance(x, y);

Abstraction:

* Abstraction is the process of generalization: taking a concrete implementation and making it applicable to different, albeit somewhat related, types of data. The classical example of abstraction is C's qsort function which sorts data.
* The thing about qsort is that it doesn't care about the data it sorts – in fact, it doesn't know what data it sorts. Rather, its input type is a type less pointer (void\*) which is just C's way of saying “I don't care about the type of data” (this is also called type erasure). The important point is that the implementation of qsort always stays the same, regardless of data type. The only thing that has to change is the compare function, which differs from data type to data type. qsort therefore expects the user to provide said compare function as a function argument.

### Abstraction VS Information Hiding VS Encapsulation

* Go to the source! Grady Booch says (in Object Oriented Analysis and Design, page 49, second edition):

***Abstraction*** and ***encapsulation*** are complementary concepts: ***abstraction*** focuses on the observable behavior of an object... ***encapsulation*** focuses upon the implementation that gives rise to this behavior... ***encapsulation*** is most often achieved through ***information*** ***hiding***, which is the process of hiding all of the secrets of object that do not contribute to its essential characteristics.

* In other words: ***abstraction*** = the object externally; ***encapsulation*** (achieved through ***information*** ***hiding***) = the object internally.

## Initialization & Cleanup

### What does the explicit keyword in C++ mean?

*Answer*:

* In C++, the compiler is allowed to make one implicit conversion to resolve the parameters to a function. What this means is that the compiler can use single parameter constructors to convert from one type to another in order to get the right type for a parameter. Here's an example class with a constructor that can be used for implicit conversions:

class AClass

{

public:

// Single parameter constructor, can be used as an implicit conversion

AClass(int foo) : aInt(foo) { }

int GetFoo() { return aInt; }

private:

int aInt;

};

* Here's a simple function that takes a AClass object:

void aFunction(AClass anObject)

{

int i = anObject.GetFoo();

}

* and here's where the aFunction() function is called.

int main()

{

aFunction(42);

}

* The parameter is not a AClass object, but an int. However, there exists a constructor for AClass that takes an int so this constructor can be used to convert the parameter to the correct type. The compiler is allowed to do this once for each parameter.
* Prefixing the explicit keyword to the constructor prevents the compiler from using that constructor for implicit conversions. Adding it to the above class will create a compiler error at the function call aFunction(42). It is now necessary to call for conversion explicitly with aFunction(AClass(42)).
* The reason you might want to do this is to avoid accidental construction that can hide bugs.

Example:

* You have a MyString(int size) class with a constructor that constructs a string of the given size. You have a function print(const MyString&), and you call it with print(3). You expect it to print "3", but it prints an empty string of length 3 instead.

Another Example of the keyword explicit

* Suppose you have a class String

class String

{

public:

// Allocate n bytes to the String object

String(int n);

// Initializes object with char \*p

String(const char \*p);

};

* Now if you try…

String mystring = 'x';

* The char 'x' will be converted to int and will call String(int n) constructor. But this is not what the user might have intended. So to prevent such conditions, we can define the class's constructor as explicit.

class String

{

public:

// Allocate n bytes

explicit String(int n);

// Initialize object with string p

String(const char \*p);

};

### Do the parentheses after the type name make a difference with new?

*Question:*

* If 'Test' is an ordinary class, is there any difference between:

Test\* test = new Test;

//and

Test\* test = new Test();

*Answer:*

* Sometimes the memory returned by the new operator will be initialized, and sometimes it won't depending on whether the type you're newing up is a POD (plain old data), or if it's a class that contains POD members and is using a compiler-generated default constructor.
* In C++1998 there are 2 types of initialization: zero and default
* In C++2003 a 3rd type of initialization, value initialization was added.
* Assume:

// POD

struct A { int m; };

// Non-POD, compiler generated default constructor

struct B { ~B(); int m; };

// Non-POD, default-initializing m

struct C { C() : m() {}; ~C(); int m; };

* In a C++98 compiler, the following should occur:
  + new A - indeterminate value
  + new A() - zero-initialize
  + new B - default construct (B::m is uninitialized)
  + new B() - default construct (B::m is uninitialized)
  + new C - default construct (C::m is zero-initialized)
  + new C() - default construct (C::m is zero-initialized)
* In a C++03 conformant compiler, things should work like so:
  + new A - indeterminate value
  + new A() - value-initialize A, which is zero-initialization since it's a POD.
  + new B - default-initializes (leaves B::m uninitialized)
  + new B() - value-initializes B which zero-initializes all fields since its default constructor is compiler generated as opposed to user-defined.
  + new C - default-initializes C, which calls the default constructor.
  + new C() - value-initializes C, which calls the default constructor.
* So in all versions of C++ there's a difference between "new A" and "new A()" because A is a POD.
* And there's a difference in behavior between C++98 and C++03 for the case "new B()".
* When constructing an object, sometimes you want/need the parenthesis, sometimes you absolutely cannot have them, and sometimes it doesn't matter.

Example:

#include <iostream>

using namespace std;

// POD

class A

{ public: int m; };

// Non-POD, compiler generated default constructor

class B

{ public: int m; ~B(); };

// Non-POD, default-initializing m

class C

{ public: int m; C() : m() {} ~C(); };

int main(void)

{

A\* a1 = new A;  // indeterminate value

A\* a2 = new A(); // zero-initialize

B\* b1 = new B; // default construct (B::m is uninitialized)

B\* b2 = new B(); // default construct (B::m is uninitialized)

C\* c1 = new C; // default construct (C::m is zero-initialized)

C\* c2 = new C(); // default construct (C::m is zero-initialized)

cout << "new A: " << a1->m << endl;

cout << "new A(): " << a2->m << endl;

cout << "new B: " << b1->m << endl;

cout << "new B(): " << b2->m << endl;

cout << "new C: " << c1->m << endl;

cout << "new C(): " << c2->m << endl;

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

new A: -842150451

new A(): 0

new B: -842150451

new B(): -842150451

new C: 0

new C(): 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### C++ call constructor from constructor

Question:

* As an c# developer I'm used to run through constructors:

class Test {

public Test() {

DoSomething();

}

public Test(int count) : this() {

DoSomethingWithCount(count);

}

public Test(int count, string name) : this(count) {

DoSomethingWithName(name);

}

}

* Is there a way to do this in C++?

Answer:

* Unfortunately there's no way to do this in C++03.
* In two ways we can simulate this:
  1. You can combine two (or more) constructors via default parameters:

class Foo

{

public:

// Combines two constructors (char) and (char, int)

Foo(char x, int y = 0);

...

};

* 1. Use an init method to share common code…

Update: This changed in C++11 (aka C++0x), which added support for the following syntax and is called “***Delegating Constructor***”

class Foo

{

public:

Foo(char x) { init(x, int(x) + 7); }

Foo(char x, int y) { init(x, y); }

...

private:

void init(char x, int y);

};

void Foo::init(char x, int y)

{

...

}

class Foo

{

public:

Foo(char x, int y) {}

Foo(int y) : Foo('a', y) {}

};

### C++ superclass constructor calling rules

Question:

* What are the C++ rules for calling the superclass constructor from a subclass one?

Answer:

* Base class constructors are automatically called for you if they have no argument. If you want to call a superclass constructor with an argument, you must use the subclass's constructor initialization list. Unlike Java, C++ supports multiple inheritance (for better or worse), so the base class must be referred to by name, rather than "super()".

class SuperClass

{

public:

SuperClass(int foo)

{

// do something with foo

}

};

class SubClass : public SuperClass

{

public:

// Call the superclass constructor in the subclass' initialization list.

SubClass(int foo, int bar) : SuperClass(foo)

{

// do something with bar

}

};

### Why do we not have a virtual constructor in C++?

* Virtual functions basically provide polymorphic behavior. That is, when you work with an object whose dynamic type is different than the static (compile time) type with which it is referred to, it provides behavior that is appropriate for the *actual* type of object instead of the static type of the object.
* Now try to apply that sort of behavior to a constructor. When you construct an object the static type is always the same as the actual object type since:

*To construct an object, a constructor needs the exact type of the object it is to create [...] Furthermore [...] you cannot have a pointer to a constructor*

### Default constructor with empty brackets

Question:

* Is there any good reason that an empty set of round brackets (parentheses) isn't valid for calling the default constructor in C++?
* I seem to type "()" automatically every time. Is there a good reason this isn't allowed?

MyObject  object; // OK - default constructor

MyObject  object(blah);// OK

MyObject  object(); // Error

Answer:

* This is known as "***C++'s most vexing parse***". Basically, anything that can be interpreted by compiler as function declaration will be interpreted as function declaration, even if resulting AST (Abstract syntax tree) doesn't compile.

### Inheriting constructors

Question:

* Why does the following code results in error?

Result:

class A

{

public:

explicit A(int x) {}

};

class B : public A

{

};

int main(void)

{

B \*b = new B(5);

delete b;

}

Error 1 error C2664: 'B::B(const B &)' : cannot convert argument 1 from 'int' to 'const B &'

Answer:

* In C++03 standard constructors cannot be inherited and you need to inherit them manually one by one by calling base implementation on your own.

class A

{

public:

explicit A(int x) {}

};

class B : public A

{

using A::A;

};

* If your compiler supports C++11 standard, there is a ***constructor inheritance***.

### Why must const members be initialized in the constructor initializer rather than in its body?

* In C++, an object is considered fully initialized when execution enters the body of the constructor.
* What you are missing is that initialization happens in the initialization list, and assignment happens in the body of the constructor. The steps in logic:

1. A const object can only be initialized.
2. An object has all of its members initialized in the initialization list. Even if you do not explicitly initialize them there, the compiler will happily do so for you :-)
3. Therefore, putting 1) and 2) together, a member which is const can only ever have a value assigned to it at initialization, which happens during the initialization list.

### What are Aggregates and PODs and how/why are they special?

Question:

* This FAQ is about Aggregates and PODs and covers the following material:
  1. What are Aggregates?
  2. What are PODs (Plain Old Data)?
  3. How are they related?
  4. How and why are they special?
  5. What changes for C++11?

Answer:

What are aggregates and why they are special?

* Formal definition from the C++ standard (C++03 8.5.1 §1):

*An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).*

* So, OK, let's parse this definition. First of all, any array is an aggregate. A class can also be an aggregate if…wait! Nothing is said about structs or unions, can't they be aggregates? Yes, they can. In C++, the term class refers to all classes, structs, and unions. So, a class (or struct, or union) is an aggregate if and only if it satisfies the criteria from the above definitions. What do these criteria imply?
  + This does not mean an aggregate class cannot have constructors; in fact it can have a default constructor and/or a copy constructor as long as they are implicitly declared by the compiler, and not explicitly by the user.
  + No private or protected non-static data members. You can have as many private and protected member functions (but not constructors) as well as many private or protected static data members and member functions as you like and not violate the rules for aggregate classes.
  + An aggregate class can have a user-declared/user-defined copy-assignment operator and/or destructor.
  + An array is an aggregate even if it is an array of non-aggregate class type.

Now let's look at some examples:

class NotAggregate1

{

virtual void f(){} // Remember? No virtual functions

};

class NotAggregate2

{

int x; // x is private by default and non-static

};

class NotAggregate3

{

public:

NotAggregate3(int) {} // Oops, user-defined constructor

};

class Aggregate1

{

public:

NotAggregate1 member1;   // OK, public member

// OK, copy-assignment

Aggregate1& operator = (Aggregate1 const & rhs) {/\* \*/ }

private:

void f() {} // OK, just a private function

};

* You get the idea. Now let's see ***how aggregates are special***. They, unlike non-aggregate classes, can be initialized with curly braces {}. This initialization syntax is commonly known for arrays, and we just learnt that these are aggregates. So, let's start with them.

Type array\_name[n] = { a1, a2, ..., am };

if (m == n)

the ith element of the array is initialized with a­i

else if (m < n)

the first m elements of the array are initialized with a1, a2, ..., am and the other n - m elements are, if possible, *value-initialized* (see below for the explanation of the term)

else if (m > n)

the compiler will issue an error

else *(this is the case when* n *isn't specified at all like* inta[] = {1,2,3};*)*

the size of the array (n) is assumed to be equal to m, so int a[] = {1,2,3} is equivalent to int a[3] = {1,2,3};

* When an object of scalar type (bool, int, char, double, pointers, etc.) is value-initialized it means it is initialized with 0 for that type (false for bool, 0.0 for double, etc.).
* When an object of class type with a user-declared default constructor is value-initialized its default constructor is called. If the default constructor is implicitly defined then all non-static members are recursively value-initialized. This definition is imprecise and a bit incorrect but it should give you the basic idea. A reference cannot be value-initialized. Value-initialization for a non-aggregate class can fail if, for example, the class has no appropriate default constructor.

Examples of array initialization:

class A

{

public:

// No default constructor

A(int){}

};

class B

{

public:

// Default constructor available

B() {}

};

int main()

{

// OK n == m

A a1[3] = { A(2), A(1), A(14) };

// ERROR A has no default constructor.

// Unable to value-initialize a2[1] and a2[2]

A a2[3] = { A(2) };

// OK b1[1] and b1[2] are value initialized,

// in this case with the default constructor

B b1[3] = { B() };

// All elements are initialized with 0;

int Array1[1000] = { 0 };

// Attention: only the first element is 1, the rest are 0;

int Array2[1000] = { 1 };

// The braces can be empty too. All elements initialized with false

bool Array3[1000] = {};

// No initializer. This is different from an empty {} initializer in that

// the elements in this case are not value

// initialized, but have indeterminate values

// (unless, of course, Array4 is a global array)

int Array4[1000];

// ERROR, too many initializers

int array[2] = { 1, 2, 3, 4 };

}

* Now let's see how aggregate classes can be initialized with braces. Pretty much the same way. Instead of the array elements we will initialize the non-static data members in the order of their appearance in the class definition (they are all public by definition). If there are fewer initializers than members, the rest are value-initialized. If it is impossible to value-initialize one of the members which were not explicitly initialized, we get a compile-time error. If there are more initializers than necessary, we get a compile-time error as well.

struct X

{

int i1;

int i2;

};

struct Y

{

char c;

X x;

int i[2];

float f;

protected:

static double d;

private:

void g(){}

};

Y y = { 'a', { 10, 20 }, { 20, 30 } };

* In the above example y.c is initialized with 'a', y.x.i1 with 10, y.x.i2 with 20, y.i[0] with 20, y.i[1] with 30 and y.f is value-initialized, that is, initialized with 0.0. The protected static member d is not initialized at all, because it is static.
* Aggregate unions are different in that you may initialize only their first member with braces. I think that if you are advanced enough in C++ to even consider using unions (their use may be very dangerous and must be thought of carefully), you could look up the rules for unions in the standard yourself :).
* Now that we know what's special about aggregates, let's try to understand the restrictions on classes; that is, why they are there. We should understand that member wise initialization with braces implies that the class is nothing more than the sum of its members. If a user-defined constructor is present, it means that the user needs to do some extra work to initialize the members therefore brace initialization would be incorrect. If virtual functions are present, it means that the objects of this class have (on most implementations) a pointer to the so-called vtable of the class, which is set in the constructor, so brace-initialization would be insufficient. You could figure out the rest of the restrictions in a similar manner as an exercise :)
* So enough about the aggregates. Now we can define a stricter set of types, to wit, PODs…

What are PODs and why they are special

* Formal definition from the C++ standard (C++03 9 §4):

*A POD-struct is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor. Similarly, a POD-union is an aggregate union that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor. A POD class is a class that is either a POD-struct or a POD-union.*

* Wow, this one's tougher to parse, isn't it? :) Let's leave unions out (on the same grounds as above) and rephrase in a bit clearer way:

*An aggregate class is called a POD if it has no user-defined copy-assignment operator and destructor and none of its non-static members is a non-POD class, array of non-POD, or a reference.*

* What does this definition imply? (Did I mention **POD** stands for **Plain Old Data** ?:)
  + All POD classes are aggregates, or, to put it the other way around, if a class is not an aggregate then it is sure not a POD
  + Classes, just like structs, can be PODs even though the standard term is POD-struct for both cases
  + Just like in the case of aggregates, it doesn't matter what static members the class has

Examples:

struct POD

{

int x;

char y;

// No harm if there's a function

void f() {}

// Static members do not matter

static std::vector<char> v;

};

struct AggregateButNotPOD1

{

int x;

// User-defined destructor

~AggregateButNotPOD1(){}

};

struct AggregateButNotPOD2

{

// Array of non-POD class

AggregateButNotPOD1 arrOfNonPod[3];

};

* POD-classes, POD-unions, scalar types, and arrays of such types are collectively called **POD-types.**  
  PODs are special in many ways. I'll provide just some examples.
* POD-classes are the closest to C structs. Unlike them, PODs can have member functions and arbitrary static members, but neither of these two change the memory layout of the object. So if you want to write a more or less portable dynamic library that can be used from C and even .NET, you should try to make all your exported functions take and return only parameters of POD-types.
* The lifetime of objects of non-POD class type begins when the constructor has finished and ends when the destructor has finished. For POD classes, the lifetime begins when storage for the object is occupied and finishes when that storage is released or reused.
* For objects of POD types it is guaranteed by the standard that when you memcpy the contents of your object into an array of char or unsigned char, and then memcpy the contents back into your object, the object will hold its original value. Do note that there is no such guarantee for objects of non-POD types. Also, you can safely copy POD objects with memcpy. The following example assumes T is a POD-type:

#define N sizeof(T)

char buf[N];

// obj initialized to its original value

T obj;

// between these two calls to memcpy, obj might be modified

memcpy(buf, &obj, N);

// at this point, each sub-object of obj of scalar type

// holds its original value

memcpy(&obj, buf, N);

* goto statement. As you may know, it is illegal (the compiler should issue an error) to make a jump via goto from a point where some variable was not yet in scope to a point where it is already in scope. This restriction applies only if the variable is of non-POD type. In the following example f() is ill-formed whereas g() is well-formed. Note that Microsoft's compiler is too liberal with this rule—it just issues a warning in both cases.

int f()

{

struct NonPOD { NonPOD(){} };

goto label;

NonPOD x;

label:

return 0;

}

int g()

{

struct POD { int i;  char c; };

goto label;

POD x;

label:

return 0;

}

* It is guaranteed that there will be no padding in the beginning of a POD object. In other words, if a POD-class A's first member is of type T, you can safely reinterpret\_cast from A\* to T\* and get the pointer to the first member and vice versa.

Conclusion

* It is important to understand what exactly a POD is because many language features, as you see, behave differently for them.

### Is there an implicit default constructor in C++?

Answer:

* If you do not define a constructor, the compiler will define a default constructor for you.
* The implementation of this default constructor is:
  + default construct the base class (if the base class does not have a default constructor, this is a compilation failure)
  + default construct each member variable in the order of declaration. (If a member does not have a default constructor, this is a compilation failure).
* Note: The POD data (int, float, pointer, etc.) do not have an explicit constructor but the default action is to do nothing (in the vane of C++ philosophy; we do not want to pay for something unless we explicitly ask for it).
* If no destructor/copy Constructor/Assignment operator is defined the compiler builds one of those for you (so a class always has a destructor/Copy Constructor/Assignment Operator (unless you cheat and explicitly declare one but don't define it)).
* The default implementation is:

*Destructor*:

* + If user-defined destructor is defined, execute the code provided.
  + Call the destructor of each member in reverse order of declaration
  + Call the destructor of the base class.

Copy Constructor:

* + Call the Base class Copy Constructor.
  + Call the copy constructor for each member variable in the order of declaration.

Assignment Operator:

* + Call the base class assignment operator
  + Call the assignment operator of each member variable in the order of declaration.
  + Return a reference to this.
* Note: Copy Construction/Assignment operator of POD Data is just copying the data (Hence the shallow copy problem associated with RAW pointers).

### C++: constructor initializer for arrays

Question:

* How do I initialize an array of objects properly in C++?

struct Foo

{

Foo(int x) { /\* ... \*/ }

};

struct Baz

{

Foo foo[3];

// ??? I know the following syntax is wrong, but what's correct?

Baz() : foo[0](4), foo[1](5), foo[2](6) { }

};

Answer:

* In C++98:
  + There is no way. You need a default constructor for array members and it will be called, afterwards, you can do any initialization you want in the constructor.
* In C++11:

struct Baz

{

Foo foo[3];

// VS2013 doesn’t support this!!!

Baz() : foo{ 4, 5, 6 } {}

};

### Conditions for automatic generation of default ctor, copy ctor, and default assignment operator?

* In the following, "auto-generated" means "implicitly declared as defaulted, but not defined as deleted". There are situations where the special member functions are declared, but defined as deleted.
  + The default constructor is auto-generated if there is no user-declared constructor (§12.1/5).
  + The copy constructor is auto-generated if there is no user-declared move constructor or move assignment operator (because there are no move constructors or move assignment operators in C++03, this simplifies to "always" in C++03) (§12.8/8).
  + The copy assignment operator is auto-generated if there is no user-declared move constructor or move assignment operator (§12.8/19).
  + The destructor is auto-generated if there is no user-declared destructor (§12.4/4).
* C++0x only:
  + The move constructor is auto-generated if there is no user-declared copy constructor, copy assignment operator or destructor, and if the generated move constructor is valid (e.g. if it wouldn't need to assign constant members) (§12.8/10).
  + The move assignment operator is auto-generated if there is no user-declared copy constructor, copy assignment operator or destructor, and if the generated move assignment operator is valid (e.g. if it wouldn't need to assign constant members) (§12.8/21).

### What is a non-trivial constructor in C++?

* In simple words a "trivial" special member function literally means a member function that does its job in a very straightforward manner. The "straightforward manner" means different thing for different kinds of special member functions.
* For a default constructor and destructor being "trivial" means literally "do nothing at all". For copy-constructor and copy-assignment operator, being "trivial" means literally "be equivalent to simple raw memory copying" (like copy with memcpy).
* If you define a constructor yourself, it is considered non-trivial, even if it doesn't do anything, so a trivial constructor must be implicitly defined by the compiler.
* In order for a special member function to satisfy the above requirements, the class must have a very simplistic structure, it must not require any hidden initializations when an object is being created or destroyed, or any hidden additional internal manipulations when it is being copied.
* For example, if class has virtual functions, it will require some extra hidden initializations when objects of this class are being created (initialize virtual method table and such), so the constructor for this class will not qualify as trivial.
* For another example, if a class has virtual base classes, then each object of this class might contain hidden pointers that point to other parts of the very same object. Such a self-referential object cannot be copied by a simple raw memory copy routine (like memcpy). Extra manipulations will be necessary to properly re-initialize the hidden pointers in the copy. For this reason the copy constructor and copy-assignment operator for this class will not qualify as trivial.
* For obvious reasons, this requirement is recursive: all sub objects of the class (bases and non-static members) must also have trivial constructors.

### Do I need to explicitly call the base virtual destructor?

Answer:

* No, destructors are called automatically in the reverse order of construction. (Base classes last). Do not call base class destructors.

### What is the use of having destructor as private?

Answer:

* 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.

### What is the use of “delete this”?

* "delete this" is commonly used for ref counted objects. For a ref counted object the decision of when to delete is usually placed on the object itself. Here is an example of what a Release method would look like

int MyRefCountedObject::Release()

{

\_refCount--;

if (0 == \_refCount)

{

delete this;

return 0;

}

return \_refCount;

}

### How does the C++ runtime system know when objects go out of scope?

* The runtime doesn't - the compiler keeps tabs on scope and generates the code to call the destructor. If you make a simple test application and look at the generated disassembly you'll see explicit destructor calls.
* Disassembly snippet from MSVC:

int main()

{

std::string s1;

00971416  call dword ptr[\_\_imp\_std::basic\_string<char, std::char\_traits<char>, std::allocator<char> >::basic\_string<char, std::char\_traits<char>, std::allocator<char> >(979290h)]

} // main ends here

0097146B  call dword ptr[\_\_imp\_std::basic\_string<char, std::char\_traits<char>, std::allocator<char> >::**~basic\_string**<char, std::char\_traits<char>, std::allocator<char> >(979294h)]

### Why do we need a pure virtual destructor in C++?

Question:

* I understand the need for a virtual destructor. But why do we need a pure virtual destructor? In one of the C++ articles, the author has mentioned that we use pure virtual destructor when we want to make a class abstract.
* But we can make a class abstract by making any of the member functions as pure virtual.
* So my questions are
  1. When do we really make a destructor pure virtual? Can anybody give a good real time example?
  2. When we are creating abstract classes is it a good practice to make the destructor also pure virtual? If yes. Then why?

Answer:

1. Probably the real reason that pure virtual destructors are allowed is that to prohibit them would mean adding another rule to the language and there's no need for this rule since no ill-effects can come from allowing a pure virtual destructor.
2. Nope, plain old virtual is enough in most (all?) cases.

* If you create an object with default implementations for its virtual methods and want to make it abstract without forcing anyone to override any **specific** method, you can make the destructor pure virtual. I don't see much point in it but it's possible.
* One may also assume that every deriving class would probably need to have specific clean-up code and use the pure virtual destructor as a reminder to write one but this seems contrived.

**Note:** The destructor is the only method that even if it is pure virtual **has** to have an implementation in order for the class it's defined in to be useful (yes pure virtual functions can have implementations).

struct foo

{

virtual void bar() = 0;

};

void foo::bar() { /\* default implementation \*/ }

class foof : public foo

{

// have to explicitly call default implementation.

void bar() { foo::bar(); }

};

### Will an 'empty' destructor do the same thing as the generated destructor?

* It will do the same thing (nothing, in essence). But it's not the same as if you didn't write it. Because writing the ***destructor will require a working base-class destructor***. If the base class destructor is private or if there is any other reason it can't be invoked, then your program is faulty.
* Consider this:

struct A { private: ~A(); };

struct B : A { };

* That is OK, as long as your don't require to destruct an object of type B (and thus, implicitly of type A) - like if you never call delete on a dynamically created object, or you never create an object of it in the first place. If you do, then the compiler will display an appropriate diagnostic.
* Now if you provide one explicitly…

struct A { private: ~A(); };

struct B : A { ~B() { /\* ... \*/ } };

* That one will try to implicitly call the destructor of the base-class, and will cause a diagnostic already at definition time of ~B.

### Object destruction in C++?

Question:

* When exactly are objects destroyed in C++, and what does that mean? Do I have to destroy them manually, since there is no Garbage Collector? How do exceptions come into play?

Answer:

* In the following text, I will distinguish between scoped objects, whose time of destruction is statically determined by their enclosing scope (functions, blocks, classes, expressions), and dynamic objects, whose exact time of destruction is generally not known until runtime.
* While the destruction semantics of class objects are determined by destructors, the destruction of a scalar object is always a no-op. Specifically, destructing a pointer variable does not destroy the pointee.

#### Scoped objects

automatic object:

* Automatic objects (commonly referred to as "local variables") are destructed, in reverse order of their definition, when control flow leaves the scope of their definition:
* If an exception is thrown during the execution of a function, all previously constructed automatic objects are destructed before the exception is propagated to the caller. This process is called stack unwinding. During stack unwinding, no further exceptions may leave the destructors of the aforementioned previously constructed automatic objects. Otherwise, the function std::terminate is called.

void some\_function()

{

Foo a;

Foo b;

if (some\_condition)

{

Foo y;

Foo z;

} < -- - z and y are destructed here

} < -- - b and a are destructed here

non-local static objects:

* Static objects defined at namespace scope (commonly referred to as "global variables") and static data members are destructed, in reverse order of their definition, after the execution of main:

struct X

{

// this is only a \*declaration\*,

// not a \*definition\*

static Foo x;

};

Foo a;

Foo b;

int main()

{

} < -- - y, x, b and a are destructed here

// this is the respective definition

Foo X::x;

Foo y;

* Note that the relative order of construction (and destruction) of static objects defined in different translation units is undefined.
* If an exception leaves the destructor of a static object, the function std::terminate is called.

local static objects:

* Static objects defined inside functions are constructed when (and if) control flow passes through their definition for the first time.1 They are destructed in reverse order after the execution of main:

Foo& get\_some\_Foo()

{

static Foo x;

return x;

}

Bar& get\_some\_Bar()

{

static Bar y;

return y;

}

int main()

{

// note that get\_some\_Bar is called \*first\*

get\_some\_Bar().do\_something();

get\_some\_Foo().do\_something();

} < -- - x and y are destructed here

  // hence y is destructed \*last\*

* If an exception leaves the destructor of a static object, the function std::terminate is called.

base class sub objects and member sub objects:

* When control flow leaves the destructor body of an object, its member sub objects (also known as its "data members") are destructed in reverse order of their definition. After that, its base class sub objects are destructed in reverse order of the base-specifier-list:

class Foo : Bar, Baz

{

Quux x;

Quux y;

public:

~Foo()

{

} < -- - y and x are destructed here,

}; // followed by the Baz and Bar base class sub objects

* If an exception is thrown during the *construction* of one of Foo's subobjects, then all its previously constructed subobjects will be destructed before the exception is propagated. The Foo destructor, on the other hand, will *not* be executed, since the Foo object was never fully constructed.
* Note that the destructor body is not responsible for destructing the data members themselves. You only need to write a destructor if a data member is a handle to a resource that needs to be released when the object is destructed (such as a file, a socket, a database connection, a mutex, or heap memory).

array elements:

* Array elements are destructed in descending order. If an exception is thrown during the construction of the n-th element, the elements n-1 to 0 are destructed before the exception is propagated.

temporary objects:

* A temporary object is constructed when a prvalue expression of class type is evaluated. The most prominent example of a prvalue expression is the call of a function that returns an object by value, such as T operator+(const T&, const T&). Under normal circumstances, the temporary object is destructed when the full-expression that lexically contains the prvalue is completely evaluated:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ full-expression

\_\_\_\_\_\_\_\_\_\_\_ subexpression

\_\_\_\_\_\_\_ subexpression

some\_function(a + " " + b);

^ both temporary objects are destructed here

* The above function call some\_function(a + " " + b) is a full-expression because it is not part of a larger expression (instead, it is part of an expression-statement). Hence, all temporary objects that are constructed during the evaluation of the subexpressions will be destructed at the semicolon. There are two such temporary objects: the first is constructed during the first addition, and the second is constructed during the second addition. The second temporary object will be destructed before the first.
* If an exception is thrown during the second addition, the first temporary object will be destructed properly before propagating the exception.
* If a local reference is initialized with a prvalue expression, the lifetime of the temporary object is extended to the scope of the local reference, so you won't get a dangling reference:

{

const Foo& r = a + " " + b;

^ first temporary(a + " ") is destructed here

// ...

} < -- - second temporary(a + " " + b) is destructed not until here

* If a prvalue expression of non-class type is evaluated, the result is a value, not a temporary object. However, a temporary object will be constructed if the prvalue is used to initialize a reference:

const int& r = i + j;

#### Dynamic objects and arrays

* In the following section, destroy X means "first destruct X and then release the underlying memory". Similarly, create X means "first allocate enough memory and then construct X there".

dynamic objects:

* A dynamic object created via p = new Foo is destroyed via delete p. If you forget to delete p, you have a resource leak. You should never attempt to do one of the following, since they all lead to undefined behavior:
  + destroy a dynamic object via delete[] (note the square brackets), free or any other means
  + destroy a dynamic object multiple times
  + access a dynamic object after it has been destroyed
* If an exception is thrown during the construction of a dynamic object, the underlying memory is released before the exception is propagated. (The destructor will not be executed prior to memory release, because the object was never fully constructed.)

dynamic arrays:

* A dynamic array created via p = new Foo[n] is destroyed via delete[] p (note the square brackets). If you forget to delete[] p, you have a resource leak. You should never attempt to do one of the following, since they all lead to undefined behavior:
  + destroy a dynamic array via delete, free or any other means
  + destroy a dynamic array multiple times
  + access a dynamic array after it has been destroyed
* If an exception is thrown during the construction of the n-th element, the elements n-1 to 0 are destructed in descending order, the underlying memory is released, and the exception is propagated.
* (You should generally prefer std::vector<Foo> over Foo\* for dynamic arrays. It makes writing correct and robust code much easier.)

reference-counting smart pointers:

* A dynamic object managed by several std::shared\_ptr<Foo> objects is destroyed during the destruction of the last std::shared\_ptr<Foo> object involved in sharing that dynamic object.
* (You should generally prefer std::shared\_ptr<Foo> over Foo\* for shared objects. It makes writing correct and robust code much easier.)