Contents

[1. Classes and Objects 4](#_Toc392941105)

[a. What is a class? 4](#_Toc392941106)

[b. What is an object? 4](#_Toc392941107)

[c. What is encapsulation? 4](#_Toc392941108)

[d. Why is the size of an empty class not zero? 4](#_Toc392941109)

[2. References 4](#_Toc392941110)

[a. What is a reference? 4](#_Toc392941111)

[b. What happens if you return a reference? 4](#_Toc392941112)

[c. What does object.method1().method2() mean? 5](#_Toc392941113)

[d. What does it mean that a reference must refer to an object, not a dereferenced NULL pointer? 5](#_Toc392941114)

[3. Inline functions 5](#_Toc392941115)

[a. What's the deal with inline functions? 5](#_Toc392941116)

[b. How do you tell the compiler to make a member function inline? 5](#_Toc392941204)

[c. Is there another way to tell the compiler to make a member function inline? 6](#_Toc392941205)

[d. With inline member functions that are defined outside the class, is it best to put the inline keyword next to the declaration within the class body, next to the definition outside the class body, or both? 6](#_Toc392941206)

[4. Constructors 7](#_Toc392941207)

[a. Is there any difference between List x; and List x();? 7](#_Toc392941208)

[b. Is the default constructor for Fred always Fred::Fred()? 8](#_Toc392941209)

[c. Should my constructors use “initialization lists” or “assignment”? 8](#_Toc392941210)

[d. How should initializers be ordered in a constructor’s initialization list? 10](#_Toc392941211)

[e. What is the “Named Constructor Idiom”? 11](#_Toc392941212)

[f. Why can’t I initialize my static member data in my constructor’s initialization list? 12](#_Toc392941213)

[g. Can I add = *initializer*; to the declaration of a class-scope static const data member? 13](#_Toc392941214)

[h. What’s the “static initialization order fiasco”? 14](#_Toc392941215)

[i. How do I prevent the “static initialization order fiasco”? 14](#_Toc392941216)

[j. Why am I getting an error after declaring a Foo object via Foo x(Bar())? 15](#_Toc392941217)

[k. What is the purpose of the explicit keyword? 16](#_Toc392941218)

[l. The "Nifty Counter" or "Schwarz Counter" idiom 17](#_Toc392941219)

[5. Destructors 19](#_Toc392941220)

[a. What’s the deal with destructors? 19](#_Toc392941221)

[b. What is “placement new” and why would I use it? 19](#_Toc392941222)

[c. Is there a way to force new to allocate memory from a specific memory area? 21](#_Toc392941223)

[6. Assignment Operators 21](#_Toc392941224)

[a. How do I handle self-assignment? 21](#_Toc392941225)

[b. I’m creating a derived class; should my assignment operators call my base class’s assignment operators? 21](#_Toc392941226)

[7. Operator Overloading 23](#_Toc392941227)

[a. What operators can/cannot be overloaded? 23](#_Toc392941228)

[b. Why can’t I overload . (dot), ::, sizeof, etc.? 23](#_Toc392941229)

[c. Can I overload operator== so it lets me compare two char[] using a string comparison? 24](#_Toc392941230)

[8. Friends 24](#_Toc392941231)

[a. What is a friend? 24](#_Toc392941232)

[b. Do friends violate encapsulation? 24](#_Toc392941233)

[c. When should you use 'friend'? 24](#_Toc392941234)

[d. What does it mean that "friendship isn't inherited, transitive, or reciprocal"? 25](#_Toc392941235)

[9. Inheritance — Basics 25](#_Toc392941236)

[a. Is inheritance important to C++? 25](#_Toc392941237)

[b. Is it okay to convert a pointer from a derived class to its base class? 25](#_Toc392941238)

[c. What’s the difference between public, private, and protected? 26](#_Toc392941239)

[10. Inheritance - virtual functions 26](#_Toc392941240)

[a. What is a “virtual member function”? 26](#_Toc392941241)

[b. Why are member functions not virtual by default? 26](#_Toc392941242)

[c. How can C++ achieve dynamic binding yet also static typing? 27](#_Toc392941243)

[d. What is a pure virtual function? 27](#_Toc392941244)

[e. What’s the diff between how virtual and non-virtual member functions are called? 29](#_Toc392941245)

[f. How virtual Table and \_vptr works 30](#_Toc392941246)

[g. When should my destructor be virtual? 33](#_Toc392941247)

[h. Why are destructors not virtual by default? 33](#_Toc392941248)

[i. Why don’t we have virtual constructors? 34](#_Toc392941249)

[j. What is a “virtual constructor”? 34](#_Toc392941250)

[11. Inheritance — Proper Inheritance and Substitutability 36](#_Toc392941251)

[a. Converting Derived\* → Base\* works okay; why doesn’t Derived\*\* → Base\*\* work? 36](#_Toc392941252)

[b. Is an array of Derived a kind-of array of Base? 37](#_Toc392941253)

[12. Inheritance — Abstract Base Classes 38](#_Toc392941254)

[a. What is an ABC? 38](#_Toc392941255)

[b. How do you define a copy constructor or assignment operator for a class that contains a pointer to a (abstract) base class? 39](#_Toc392941256)

[13. Inheritance — private and protected inheritance 39](#_Toc392941257)

[a. How are “private inheritance” and “composition” similar? 39](#_Toc392941258)

[b. Which should I prefer: composition or private inheritance? 40](#_Toc392941259)

[c. Why would the conversion between derived\* to base\* fails with private inheritance? 41](#_Toc392941260)

[14. Inheritance — Multiple and Virtual Inheritance 42](#_Toc392941261)

[a. What is a virtual base class? 42](#_Toc392941262)

[b. What is the exact order of destructors in a multiple and/or virtual inheritance situation? 43](#_Toc392941263)

[15. Templates 45](#_Toc392941264)

[a. What’s the idea behind templates? 45](#_Toc392941265)

[b. How do I explicitly select which version of a function template should get called? 45](#_Toc392941266)

[c. What is a “parameterized type”? 46](#_Toc392941267)

[d. What is “genericity”? 46](#_Toc392941268)

[e. What is SFINAE? 46](#_Toc392941269)

[f. Why can’t I separate the definition of my templates class from its declaration and put it inside a .cpp file? 48](#_Toc392941270)

[16. Pointers to Member Functions 49](#_Toc392941271)

[a. How can I avoid syntax errors when creating pointers to members? 49](#_Toc392941272)

[b. How can I avoid syntax errors when calling a member function using a pointer-to-member-function? 49](#_Toc392941273)

[c. How do I create and use an array of pointer-to-member-function? 49](#_Toc392941274)

[d. How do I declare a pointer-to-member-function that points to a const member function? 50](#_Toc392941275)

[e. What is the difference between the .\* and ->\* operators? 50](#_Toc392941276)

[f. Can I convert a pointer-to-member-function to a void\*? 51](#_Toc392941277)

[g. Can I convert a pointer-to-function to a void\*? 51](#_Toc392941278)

# Classes and Objects

## What is a class?

* The fundamental building block of OO software. A class defines a data type.

## What is an object?

* "Object" is "an instance of a class."

## What is encapsulation?

* A language mechanism for restricting access to some of the object's components. It’s about information hiding and/or organization.
* A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.
* Classes support encapsulation. But C structures don’t support encapsulation.

## Why is the size of an empty class not zero?

* To ensure that the addresses of two different objects will be different.

# References

## What is a reference?

* An alias (an alternate name) for an object. A reference is the object. It is not a pointer to the object, nor a copy of the object. It is the object.

## What happens if you return a reference?

* The function call can appear on the left hand side of an assignment operator.

class Array

{

public:

int size() const;

float& operator[] (int index);

...

};

int main()

{

Array a;

for (int i = 0; i < a.size(); ++i)

a[i] = 7;    // This line invokes Array::operator[](int)

...

}

## What does object.method1().method2() mean?

* It chains these method calls, which is why this is called method chaining.
* The first thing that gets executed is object.method1(). This returns some object, which might be a reference to object (i.e., method1() might end with return \*this;), or it might be some other object. Let's call the returned object objectB. Then objectB becomes the this object of method2().
* The most common use of method chaining is in the iostream library.

E.g., cout << x << y works because cout << x is a function that returns cout.

## What does it mean that a reference must refer to an object, not a dereferenced NULL pointer?

* It means this is illegal:

T\* p = NULL;

T& r = \*p;  // Illegal

# Inline functions

## What's the deal with inline functions?

* When the compiler inline-expands a function call, the function's code gets inserted into the caller's code stream (conceptually similar to what happens with a #define macro). This can, depending on a zillion other things, improve performance, because the optimizer can procedurally integrate the called code — optimize the called code into the caller.
* There are several ways to designate that a function is inline, some of which involve the inline keyword, others do not. No matter how you designate a function as inline, it is a request that the compiler is allowed to ignore: it might inline-expand some, all, or none of the calls to an inline function.

## How do you tell the compiler to make a member function inline?

* When you declare an inline member function, it looks just like a normal member function:

class Fred

{

public:

void f(int i, char c);

};

* But when you define an inline member function, you prepend the member function's definition with the keyword inline, and you put the definition into a header file:

It's usually imperative that the function's definition (the part between the {...}) be placed in a header file. If you put the inline function's definition into a .cpp file, and if it is called from some other .cpp file, you'll get an "unresolved external" error from the linker.

inline void Fred::f(int i, char c)

{

...

}

## Is there another way to tell the compiler to make a member function inline?

* Yes: define the member function in the class body itself: Although this is easier on the person who writes the class, it's harder on all the readers since it mixes "what" a class does with "how" it does them. Because of this mixture, we normally prefer to define member functions outside the class body with the inline keyword. The insight that makes sense of this: in a reuse-oriented world, there will usually be many people who use your class, but there is only one person who builds it (yourself); therefore you should do things that favor the many rather than the few.

class Fred

{

public:

void f(int i, char c)

{

...

}

};

## With inline member functions that are defined outside the class, is it best to put the inline keyword next to the declaration within the class body, next to the definition outside the class body, or both?

* Best practice: only in the definition outside the class body.

class Foo

{

public:

// Best practice : don't put the inline keyword here

void method();

...

};

// Best practice : put the inline keyword here

inline void Foo::method()

{

...

}

# Constructors

## Is there any difference between List x; and List x();?

* A big difference! Suppose that List is the name of some class. Then function f() declares a local List object called x:

void f()

{

List x; // Local object named x (of class List)

// ...

}

* But function g() declares a function called x() that returns a List:

void g()

{

List x(); // Function named x (that returns a List)

// ...

}

## Is the default constructor for Fred always Fred::Fred()?

* No. A “default constructor” is a constructor that can be called with no arguments. One example of this is a constructor that takes no parameters:

class Fred

{

public:

// Default constructor: can be called with no args

Fred();

// ...

};

* Another example of a “default constructor” is one that can take arguments, provided they are given default values:

class Fred

{

public:

// Default constructor: can be called with no args

Fred(int i = 3, int j = 5);

// ...

};

## Should my constructors use “initialization lists” or “assignment”?

* For POD class members, it makes no difference; it's just a matter of style. For class members which are classes, then it avoids an unnecessary call to a default constructor. Consider:

class

{

public:

int x;

AClass() { x = 0; }

AClass(int x\_) { x = x\_; }

};

class BClass

{

private:

AClass a;

public:

BClass()

{

a.x = 3;

}

};

* In this case, the constructor for BClass will call the default constructor for AClass, and then initialize a.x to 3.
* A better way would be for BClass's constructor to directly call AClass's constructor in the ***initializer list*** as shown…

class BClass

{

private:

AClass a;

public:

BClass() : a(3)

{

}

};

* This would only call AClass's AClass(int) constructor and ***not*** its default constructor.
* Furthermore, if a class doesn't have a default constructor, or you have a const member variable, you must use an initialize list as shown below…

class AClass

{

public:

// No default constructor

AClass(int x\_) { x = x\_; }

int x;

}

class BClass

{

private:

AClass a;

const int y;

public:

// 'a' and 'y' MUST be initialized in an initializer list;

BClass() : a(3), y(2)

{

... // It is an error not to do so

}

};

So, here is the list which has the answers for when to use initialization list…

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

## How should initializers be ordered in a constructor’s initialization list?

* Immediate base classes (left to right), then member objects (top to bottom). In other words, the order of the initialization list should mimic the order in which initializations will take place.

class AClass

{

public:

AClass() { cout << "Initializing AClass\n"; }

void f() { cout << "Using AClass object\n"; }

};

class BClass

{

public:

BClass(AClass& y) { y.f(); }

};

class CClass

{

public:

// Bad: should have listed x\_ before y\_

CClass() : y\_(), x\_(y\_) { }

protected:

BClass x\_;

AClass y\_;

};

int main()

{

CClass z;

return 0;

}

/\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*

Using AClass object

Initializing AClass

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

## What is the “Named Constructor Idiom”?

* 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*** publicstatic ***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.***
* For example:

Suppose we are building a Point class that represents a position on the X-Y plane. Turns out there are two common ways to specify a 2-space coordinate: **rectangular coordinates** (X+Y), **polar coordinates** (Radius + Angle). Unfortunately the parameters for these two coordinate systems are the same: two floats. This would create an ambiguity error in the overloaded constructors:

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

{

private:

/\* 1 Constructor is declared as private \*/

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

float x\_, y\_;

public:

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

/\* 2 public static methods that return a point object. \*/

// Rectangular coordinates

static Point rectangular(float x, float y);

/\* public static methods that return a point object. \*/

// Polar coordinates

static Point polar(float radius, float angle);

};

// 3 static method for each different way to construct an object.

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

{

return Point(x, y);

}

// 3 static method for each different way to construct an object.

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

{

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

}

int main()

{

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

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

// ...

}

## Why can’t I initialize my static member data in my constructor’s initialization list?

* Because you must **explicitly** define your class’s static data members.

## Can I add = initializer; to the declaration of a class-scope static const data member?

* Yes, though with some important caveats (conditions).

Example:

// Fred.h

class Fred

{

public:

// Initializer for static data member

static const int maximum = 42;

// ...

};

* And, as with other static data members, it must be defined in exactly one compilation unit, though this time without the = initializer part:

// Fred.cpp

#include "Fred.h"

// Definition of static data member without initializer

const int Fred::maximum;

// ...

* Caveats are…
  + The caveats are that you may do this only with integral or enumeration types, and that the initializer expression must be an expression that can be evaluated at ***compile-time***: it must only contain other constants, possibly combined with built-in operators. For example, 3\*4 is a ***compile-time*** constant expression, as is a\*b provided a and b are ***compile-time*** constants. After the declaration above, Fred::maximum is also a ***compile-time*** constant: it can be used in other ***compile-time*** constant expressions.
  + If you ever take the address of Fred::maximum, such as passing it by reference or explicitly saying &Fred::maximum, the compiler will make sure it has a unique address. If not, Fred::maximum won’t even take up space in your process’s static data area.

## What’s the “static initialization order fiasco”?

* The *static* initialization order fiasco is a very subtle and commonly misunderstood aspect of C++. Unfortunately it’s very hard to detect — the errors often occur before main() begins.
* In short, suppose you have two static objects x and y which exist in separate source files, say x.cpp and y.cpp. Suppose further that the initialization for the y object (typically the y object’s constructor) calls some method on the x object.
* The tragedy is that you have a 50%-50% chance of dying. If the compilation unit for x.cpp happens to get initialized first, all is well. But if the compilation unit for y.cpp get initialized first, then y’s initialization will get run before x’s initialization, and you’re toast.

## How do I prevent the “static initialization order fiasco”?

* To prevent the static initialization order fiasco, use the ***Construct On First Use Idiom***, described below.
* The basic idea of the **Construct On First Use Idiom** is to wrap your static object inside a function. For example, suppose you have two classes, Fred and Barney. There is a global Fred object called x, and a global Barney object called y. Barney’s constructor invokes the goBowling() method on the x object. The file x.cpp defines the x object:

// File x.cpp

#include "Fred.h"

Fred x;

// File y.cpp

#include "Barney.h"

Barney y;

// File Barney.cpp

#include "Barney.h"

Barney::Barney()

{

// ...

x.goBowling();

// ...

}

* You would have a static initialization disaster if y got constructed before x. As written above, this disaster would occur roughly 50% of the time, since the two objects are declared in different source files and those source files give no hints to the compiler or linker as to the order of static initialization.
* There are many solutions to this problem, but a very simple and completely portable solution is the Construct On First Use Idiom: replace the global Fred object x with a global function x() that returns the Fred object by reference.

// File x.cpp

#include "Fred.h"

Fred& x()

{

static Fred\* ans = new Fred();

return \*ans;

}

* Since static local objects are constructed the first time control flows over their declaration (only), the above new Fred() statement will only happen once: the first time x() is called. Every subsequent call will return the same Fred object (the one pointed to by ans). Then all you do is change your usages of x to x():

// File Barney.cpp

#include "Barney.h"

Barney::Barney()

{

// ...

x().goBowling();

// ...

}

* This is called the **Construct On First Use Idiom** because it does just that: the (logically namespace-scope / global) Fred object is constructed on its first use.

## Why am I getting an error after declaring a Foo object via Foo x(Bar())?

* Because that doesn’t create a Foo object - it declares a non-member function that *returns* a Foo object. The term “Most Vexing Parse” was coined by Scott Myers to describe this situation.

## What is the purpose of the explicit keyword?

* The explicit keyword is an optional decoration for constructors and conversion operators to tell the compiler that a certain constructor or conversion operator may not be used to implicitly cast an expression to its class type.

For example, without the explicit keyword the following code is valid:

class Foo

{

int x\_;

public:

Foo(int x) : x\_(x) {}

operator int() { return x\_; }

};

class Bar

{

double x\_;

public:

Bar(double x) : x\_(x) {}

operator double(){ return x\_; }

};

int main(void)

{

Foo a = 42; // Okay: calls Foo::Foo(int) passing 42 as an argument

Foo b(42); // Okay: calls Foo::Foo(int) passing 42 as an argument

Foo c = Foo(42);// Okay: calls Foo::Foo(int) passing 42 as an argument

Foo d = (Foo)42;// Okay: calls Foo::Foo(int) passing 42 as an argument

int e = d; // Okay: calls Foo::operator int()

Bar x = 3.14;// Okay: calls Bar::Bar(double) passing 3.14 as an argument

Bar y(3.14); // Okay: calls Bar::Bar(double) passing 3.14 as an argument

Bar z = Bar(3.14); // Okay: calls Bar::Bar(double) passing 3.14 as an argument

Bar w = (Bar)3.14; // Okay: calls Bar::Bar(double) passing 3.14 as an argument

double v = w; // Okay: calls Bar::operator double()

return 0;

}

* But sometimes you want to prevent this sort of implicit promotion or implicit type conversion.
* For example, if Foo is really an array-like container and 42 is the initial size, you might want to let your users say, Foo x(42); or perhaps Foo x = Foo(42); but not just Foo x = 42;. If that’s the case, you should use the explicit keyword.

class Foo

{

int x\_;

public:

explicit Foo(int x) : x\_(x) {}

explicit operator int() { return x\_; }

};

class Bar

{

double x\_;

public:

explicit Bar(double x) : x\_(x) {}

explicit operator double(){ return x\_; }

};

int main(void)

{

Foo a = 42; // **COMPILE**-**TIME** **error**: can't convert 42 to an object of type Foo

Foo b(42); // Okay: calls Foo::Foo(int) passing 42 as an argument

Foo c = Foo(42); // Okay: calls Foo::Foo(int) passing 42 as an argument

Foo d = (Foo)42; // Okay: calls Foo::Foo(int) passing 42 as an argument

int e = d; // **COMPILE**-**TIME** **error**: can't convert d to an integer

int f = int(d); // Okay: calls Foo::operator int()

Bar x = 3.14; // **COMPILE**-**TIME** **error**:can't convert 3.14 to an object of type Bar

Bar y(3.14); // Okay: calls Bar::Bar(double) passing 3.14 as an argument

Bar z = Bar(3.14); // Okay: calls Bar::Bar(double) passing 3.14 as an argument

Bar w = (Bar)3.14; // Okay: calls Bar::Bar(double) passing 3.14 as an argument

double v = w; // **COMPILE-TIME** **error**: can't convert w to a double

double u = double(w); // Okay: calls Bar::operator double()

return 0;

}

## The "Nifty Counter" or "Schwarz Counter" idiom

* Intent

Ensure a non-local static object is initialized before its first use and destroyed only after last use of the object.

* Motivation

When static objects use other static objects, the initialization problem becomes more complex. A static object must be initialized before its use if it has non-trivial initialization. Initialization order of static objects across compilation units is not well-defined. Multiple static objects, spread across multiple compilation units, might be using a single static object. Therefore, it must be initialized before use. One example is std::cout, which is typically used by a number of other static objects.

* Solution and Sample Code

The "***Nifty Counter***" or "***Schwarz*** ***counter***" idiom is an example of a reference counting idiom applied to the initialization of static objects.

//Stream.h

class StreamInitializer;

class Stream

{

friend class StreamInitializer;

public:

Stream()

{

// Constructor must be called before use.

}

};

// The counter is initialized at load-time, i.e.,

// before any of the static objects are initialized.

static int nifty\_counter;

static class StreamInitializer

{

public:

StreamInitializer()

{

if (0 == nifty\_counter++)

{

// Initialize Stream object's static members.

}

}

~StreamInitializer()

{

if (0 == --nifty\_counter)

{

// Clean-up.

}

}

} initializer; //Note object here in the header.

# Destructors

## What’s the deal with destructors?

* A destructor gives an object its last rites.
* Destructors are used to release any resources allocated by the object. E.g., class Lock might lock a semaphore, and the destructor will release that semaphore. The most common example is when the constructor uses new, and the destructor uses delete.
* Destructors are a “prepare to die” member function. They are often abbreviated “dtor”.

## What is “placement new” and why would I use it?

* 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"

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:

class Fred {};

void someCode()

{

char memory[sizeof(Fred)];

void\* p = memory;

Fred\* f = new(p)Fred();

// ...

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

}

* Another Example:

#include <iostream>

#include <new>

using namespace std;

class AClass{

public:

int a;

float b;

AClass()

{

cout << "Inside Ctor" << endl;

}

~AClass()

{

cout << "Inside Dtor" << endl;

}

};

int main(void)

{

char aMemory[sizeof(AClass)];

void\* vPtr = aMemory;

AClass\* aPtr = new(vPtr)AClass();

aPtr->~AClass();

return 0;

}

/\*\*OUTPUT\*\*

Inside Ctor

Inside Dtor

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

## Is there a way to force new to allocate memory from a specific memory area?

* Yes. The good news is that these “**Memory Pools**” are useful in a number of situations.

# Assignment Operators

## How do I handle self-assignment?

Fred& Fred::operator= (const Fred& f)

{

// Gracefully handle self assignment

if (this != &f)

{

// Put the normal assignment duties here...

}

return \*this;

}

## I’m creating a derived class; should my assignment operators call my base class’s assignment operators?

* Yes (if you need to define assignment operators in the first place).
* If you define your own assignment operators, the compiler will not automatically call your base class’s assignment operators for you. Unless your base class’s assignment operators themselves are broken, you should call them explicitly from your derived class’s assignment operators.
* However if you do not create your own assignment operators, the ones that the compiler create for you will automatically call your base class’s assignment operators.

class AClass

{

int a\_;

float b\_;

public:

AClass(int a, int b):a\_(a),b\_(b) {}

AClass& operator=(const AClass& rhs)

{

this->a\_ = rhs.a\_;

this->b\_ = rhs.b\_;

return \*this;

}

};

class BClass : public AClass

{

int c\_;

public:

BClass(int a, int b, int c): AClass(a, b), c\_(c) {}

~BClass(){}

BClass& operator=(const BClass& rhs)

{

// Calling Base class assignment operator

AClass::operator=(rhs);

this->c\_ = rhs.c\_;

return \*this;

}

};

int main(void)

{

BClass b1(2, 3, 4);

BClass b2(4, 4, 8);;

b2 = b1;

return 0;

}

# Operator Overloading

## What operators can/cannot be overloaded?

* The only C operators that can’t be are . and ?: (and sizeof, which is technically an operator).
* C++ adds a few of its own operators, most of which can be overloaded except :: and .\*.

## Why can’t I overload . (dot), ::, sizeof, etc.?

* There is no fundamental reason to disallow overloading of ?:. So far the committee just hasn’t seen the need to introduce the special case of overloading a ternary operator.
* sizeof cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it.
* In N::m neither N nor m are expressions with values; N and m are names known to the compiler and :: performs a (compile time) scope resolution rather than an expression evaluation.
* operator.(dot) could in principle be overloaded using the same technique as used for “->”. However, doing so can lead to questions about whether an operation is meant for the object overloading ‘.’ or an object referred to by ‘.’.

class AClass

{

public:

void f();

// ...

};

class BClass

{

// Assume that you can overload '.'

AClass\* p;

AClass& operator.()

{

return \*p;

}

void f();

// ...

};

void g(BClass& x)

{

x.f(); // BClass::f or AClass::f or error?

}

## Can I overload operator== so it lets me compare two char[] using a string comparison?

* No: at least one operand of any overloaded operator must be of some user-defined type (most of the time that means a class).

# Friends

## What is a friend?

* Something to allow your class to grant access to another class or function.
* Friends can be either functions or other classes. A class grants access privileges to its friends. Normally a developer has political and technical control over both the friend and member functions of a class.

## Do friends violate encapsulation?

* No! If they’re used properly, they enhance encapsulation.

## When should you use 'friend'?

* The 'friend' specifier allows the designated class access to protected data or functionality within the class making the friend statement.
* For example in the below code anyone may ask a Child for their name, but only the Mother and the Child may change the name.

class Child

{

friend class Mother;

public:

string getName(void);

protected:

void setName(string newName);

};

## What does it mean that "friendship isn't inherited, transitive, or reciprocal"?

* Just because I grant you friendship access to me doesn’t automatically grant your kids access to me, doesn’t automatically grant your friends access to me, and doesn’t automatically grant me access to you.
  + I don’t necessarily trust the kids of my friends. The privileges of friendship aren’t inherited. Derived classes of a friend aren’t necessarily friends. If class Fred declares that class Base is a friend, classes derived from Base don’t have any automatic special access rights to Fred objects.
  + I don’t necessarily trust the friends of my friends. The privileges of friendship aren’t transitive. A friend of a friend isn’t necessarily a friend. If class Fred declares class Bruce as a friend, and class Bruce declares class Betty as a friend, class Betty doesn’t necessarily have any special access rights to Fred objects.
  + You don’t necessarily trust me simply because I declare you my friend. The privileges of friendship aren’t reciprocal. If class Fred declares that class Bruce is a friend, Bruce objects have special access to Fred objects but Fred objects do not automatically have special access to Bruce objects.

# Inheritance — Basics

## Is inheritance important to C++?

* Yes. If you follow the OO paradigm, use it as a specification device.
* Human beings abstract things on two dimensions: part-of and kind-of. A Ford Taurus is-a-kind-of-a Car, and a Ford Taurus has-a Engine, Tires, etc.

## Is it okay to convert a pointer from a derived class to its base class?

* Yes. An object of a derived class is a kind of the base class. Therefore the conversion from a derived class pointer to a base class pointer is perfectly safe, and happens all the time.

## What’s the difference between public, private, and protected?

* A member (either data member or member function) declared in a private section of a class can only be accessed by member functions and friends of that class.
* A member (either data member or member function) declared in a protected section of a class can only be accessed by member functions and friends of that class, and by member functions and friends of derived classes.
* A member (either data member or member function) declared in a public section of a class can be accessed by anyone.

# Inheritance - virtual functions

## What is a “virtual member function”?

* Virtual member functions are key to the object-oriented paradigm, such as making it easy for old code to call new code.
* A virtual function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer. This allows algorithms in the base class to be replaced in the derived class, even if users don’t know about the derived class.
* The derived class can either fully replace (“override”) the base class member function, or the derived class can partially replace (“augment”) the base class member function. The latter is accomplished by having the derived class member function call the base class member function, if desired.

## Why are member functions not virtual by default?

* Because many classes are not designed to be used as base classes. For example, class complex.
* Also, objects of a class with a virtual function require space needed by the virtual function call mechanism - typically one word per object. This overhead can be significant, and can get in the way of layout compatibility with data from other languages (e.g. C and Fortran).

## How can C++ achieve dynamic binding yet also static typing?

* When you have a pointer to an object, the object may actually be of a class that is derived from the class of the pointer (e.g., a Vehicle\* that is actually pointing to a Car object; this is called “polymorphism”). Thus there are two types: the (static) type of the pointer (Vehicle, in this case), and the (dynamic) type of the pointed-to object (Car, in this case).
* *Static typing* means that the legality of a member function invocation is checked at the earliest possible moment: by the compiler at compile time. The compiler uses the static type of the pointer to determine whether the member function invocation is legal. If the type of the pointer can handle the member function, certainly the pointed-to object can handle it as well. E.g., if Vehicle has a certain member function, certainly Car also has that member function since Car is a kind-of Vehicle.
* *Dynamic binding* means that the address of the code in a member function invocation is determined at the last possible moment: based on the dynamic type of the object at run time. It is called “dynamic binding” because the binding to the code that actually gets called is accomplished dynamically (at run time). Dynamic binding is a result of virtual functions.

## What is a pure virtual function?

* A pure virtual function is a function that must be overridden in a derived class and need not be defined. A virtual function is declared to be “pure” using the curious =0 syntax. For example:

class Base

{

public:

void f1(); // not virtual

virtual void f2(); // virtual, not pure

virtual void f3() = 0; // pure virtual

};

Base b; // error: pure virtual f3 not overridden

* Here, Base is an abstract class (because it has a pure virtual function), so no objects of class Base can be directly created: Base is (explicitly) meant to be a base class. For example:

class Derived : public Base

{

// no f1: fine

// no f2: fine, we inherit Base::f2

void f3();

};

Derived d; // ok: Derived::f3 overrides Base::f3

* Abstract classes are immensely useful for defining interfaces. In fact, a class with no data and where all functions are pure virtual functions is often called an interface.
* You can provide a definition for a pure virtual function:
* This is very occasionally useful (to provide some simple common implementation detail for derived classes), but Base::f3() must still be overridden in some derived class. If you don’t override a pure virtual function in a derived class, that derived class becomes abstract:

class Base

{

public:

virtual void f3(void) = 0;

};

// We can provide a definition for a

// pure virtual function:

void Base::f3(void){

cout << "Inside PURE Virtual Function\n";

}

class Derived : public Base

{

public:

void anotherFun(void){

cout << "Inside Derived anotherFun\n";

}

void f3(void){

cout << "Implementing PURE virtual fun\n";

}

};

int main(void)

{

// Without implementing pure virtual function

// below code is error.

//Derived d;

Derived d;

d.f3();

return 0;

}

## What’s the diff between how virtual and non-virtual member functions are called?

* Non-virtual member functions are resolved statically. That is, the member function is selected statically (at compile-time) based on the type of the pointer (or reference) to the object.
* In contrast, virtual member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) ***based on the type of the object***, ***not the type of the pointer/reference to that object***. This is called “dynamic binding.” Most compilers use some variant of the following technique: if the object has one or more virtual functions, the compiler puts a hidden pointer in the object called a “virtual-pointer” or “v-pointer.” This v-pointer points to a global table called the “virtual-table” or “v-table.”
* The compiler creates a v-table for each class that has at least one virtual function. For example, if class Circle has virtual functions for draw() and move() and resize(), there would be ***exactly one v-table associated with class*** Circle, even if there were a gazillion Circle objects, and the v-pointer of each of those Circle objects would point to the Circle v-table. The v-table itself has pointers to each of the virtual functions in the class. For example, the Circle v-table would have three pointers: a pointer to Circle::draw(), a pointer to Circle::move(), and a pointer to Circle::resize().
* During a dispatch of a virtual function, the run-time system follows the object’s v-pointer to the class’s v-table, then follows the appropriate slot in the v-table to the method code.
* The space-cost overhead of the above technique is nominal: an extra pointer per object (but only for objects that will need to do dynamic binding), plus an extra pointer per method (but only for virtual methods). The time-cost overhead is also fairly nominal: compared to a normal function call, a virtual function call requires two extra fetches (one to get the value of the v-pointer, a second to get the address of the method). None of this runtime activity happens with non-virtual functions, since the compiler resolves non-virtual functions exclusively at compile-time based on the type of the pointer.

## How virtual Table and \_vptr works

* Background
  + Virtual Table is a lookup table of function pointers used to dynamically bind the virtual functions to objects at runtime. It is not intended to be used directly by the program, and as such there is no standardized way to access it.
* Virtual Table:
  + Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given its own virtual table as a secret data member.
  + This table is set up by the compiler at compile time.
  + A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.
  + Virtual table stores NULL pointer to pure virtual functions in ABC.
  + Virtual Table is created even for classes that have virtual base classes. In this case, the vtable has pointer to the shared instance of the base class along with the pointers to the classe's virtual functions if any.
* \_vptr:
  + This vtable pointer or \_vptr, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.
  + This \_vptr is inherited to all the derived classes.
  + Each object of a class with virtual functions transparently stores this \_vptr.
  + Call to a virtual function by an object is resolved by following this hidden \_vptr.
* Example:
  + Here we have 3 classes Base, D1 and D2. Where D1 and D2 are derived from class Base.
  + class Base has 2 virtual functions, function1() and function2().
  + D1 overrides function1().
  + D2 overrides function2().

#include<iostream>

using namespace std;

class Base

{

public:

virtual void function1() { cout << "Base :: function1()\n"; };

virtual void function2() { cout << "Base :: function2()\n"; };

virtual ~Base(){};

};

class D1 : public Base

{

public:

~D1(){};

virtual void function1() { cout << "D1 :: function1()\n"; };

};

class D2 : public Base

{

public:

~D2(){};

virtual void function2() { cout << "D2 :: function2\n"; };

};

int main()

{

D1 \*d = new D1;;

Base \*b = d;

b->function1();

b->function2();

delete (b);

return (0);

}

* Explanation:
  + Here in function main() b pointer gets assigned to D1's \_vptr and now starts pointing to D1's vtable. Then calling to a function1(), makes it's \_vptr straightway calls D1's vtable function1() and so in turn calls D1's method i.e. function1() as D1 has its own function1() defined it's class.
  + Whereas pointer b calling to a function2(), makes it's \_vptr points to D1's vatble which in-turn pointing to Base class's vtable function2() as shown in the diagram (as D1 class does not have it's own definition or function2()).
  + So, now calling delete on pointer b follows the \_vptr - which is pointing to D1's vtable calls it's own class's destructor i.e. D1 class's destructor and then calls the destructor of Base class - this as part of when derived object gets deleted it turn deletes it's embedded base object. That’s why we must always make Base class's destructor as virtual if it has any virtual functions in it.
* Here is a pictorial representation of **Virtual Table** and \_vptr for the above code:

Base

\*\_\_vptr;

virtual function1();

virtual function2();

virtual ~Base();

class D1 : public Base

\*\_\_vptr, (inherited);

virtual ~D1();

virtual function1();

class D2 : public Base

\*\_\_vptr, (inherited);

virtual ~D2();

virtual function2();

Base VTable

function1()

function2()

~Base()

D1 VTable

~D1()

function1()

function2()

D2 VTable

~D2()

function1()

function2()

## When should my destructor be virtual?

* Virtual destructors are useful when you can delete an instance of a derived class through a pointer to base class:
* Here, you'll notice that we didn't declare Base's destructor to be virtual. Now, let's have a look at the following snippet:

class Base

{

// some virtual methods

};

class Derived : public Base

{

~Derived()

{

// Do some important cleanup

}

}

Base \*b = new Derived();

// use b

delete b; // Here's the problem!

* Since Base's destructor is not virtual and b is a Base\* pointing to a Derived object, delete b; has undefined behavior. In most implementations, the call to the destructor will be resolved like any non-virtual code, meaning that the destructor of the Base class will be called but not the one of the Derived class, resulting in resources leak.
* In particular, here’s when you need to make your destructor virtual:
  + *if* someone will derive from your class,
  + *and if* someone will say new Derived, where Derived is derived from your class Base,
  + *and if* someone will say delete b, where the actual object’s type is Derived but the pointer b’s type is your class Base.

## Why are destructors not virtual by default?

* You don't pay for what you don't need. If you never delete through base pointer, you may not want the overhead of the in-directed destructor call.
* Because many classes are not designed to be used as base classes. virtual functions make sense only in classes meant to act as interfaces to objects of derived classes.

## Why don’t we have virtual constructors?

* Technical reason
  + The object exists only after the constructor ends. In order for the constructor to be dispatched using the virtual table, there has to be an existing object with a pointer to the virtual table, but how can a pointer to the virtual table exist if the object still doesn't exist?
* Logic reason
  + You use the virtual keyword when you want to declare a somewhat polymorphic behavior. But there is nothing polymorphic with constructors. In particular, "virtual" allow us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

## What is a “virtual constructor”?

* An idiom that allows you to do something that C++ doesn’t directly support.
* You can get the effect of a virtual constructor by a virtual clone() member function (for copy constructing), or a virtual create() member function (for the default constructor).

class Shape {

public:

virtual ~Shape() { } // A virtual destructor

virtual void draw() = 0; // A pure virtual function

virtual void move() = 0;

virtual Shape\* clone() const = 0; // Uses the copy constructor

virtual Shape\* create() const = 0; // Uses the default constructor

};

class Circle : public Shape {

public:

Circle\* clone() const { return new Circle(\*this); }

Circle\* create() const { return new Circle(); }

void draw() { cout << "Drawing Circle!!!\n"; }

void move() { cout << "Moving Circle!!!\n"; }

};

* In the clone()  member function, the new Circle(\*this); code calls Circle’s copy constructor to copy the state of this into the newly created Circle object. (Note: unless Circle is known to be final (AKA a leaf), you can reduce the chance of slicing by making its copy constructor protected.) In the create() member function, the new Circle() code calls Circle’s default constructor.
* Users use these as if they were “virtual constructors”:

void userCode(Shape& s)

{

Shape\* aShape = s.create();

aShape->draw();

aShape->move();

delete aShape;

}

* This function will work correctly regardless of whether the Shape is a Circle, Square, or some other kind-of Shape that doesn’t even exist yet.

# Inheritance — Proper Inheritance and Substitutability

## Converting Derived\* → Base\* works okay; why doesn’t Derived\*\* → Base\*\* work?

* Because converting Derived\*\* → Base\*\* would be invalid and dangerous.
* C++ allows the conversion Derived\* → Base\*, since a Derived object is a kind of a Base object. However trying to convert Derived\*\* → Base\*\* is flagged as an error.

class Vehicle

{

public:

virtual ~Vehicle() { }

virtual void startEngine() = 0;

};

class Car : public Vehicle

{

public:

virtual void startEngine();

virtual void openGasCap();

};

class NuclearSubmarine : public Vehicle

{

public:

virtual void startEngine();

virtual void fireNuclearMissle();

};

int main()

{

Car car;

Car\* carPtr = &car;

Car\*\* carPtrPtr = &carPtr;

// This is an error in C++

Vehicle\*\* vehiclePtrPtr = carPtrPtr;

NuclearSubmarine sub;

NuclearSubmarine\* subPtr = &sub;

\*vehiclePtrPtr = subPtr;

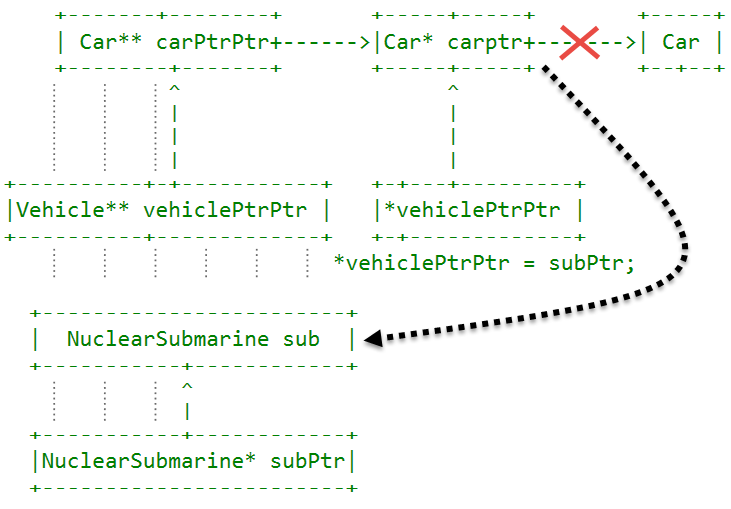
/\* This last line would have caused carPtr

 to point to sub! This might call fireNuclearMissle()!\*/

carPtr->openGasCap();

return 0;

}



## Is an array of Derived a kind-of array of Base?

* Nope.

class Base {

public:

virtual void f(); // 1

};

class Derived : public Base {

private:

int i\_; // 2

};

void userCode(Base\* arrayOfBase) {

arrayOfBase[1].f(); // 3

}

int main() {

Derived arrayOfDerived[10]; // 4

userCode(arrayOfDerived); // 5

}

* The compiler thinks this is perfectly type-safe. Line // 5 converts a Derived\* to a Base\*. But in reality it is horrendously evil: since Derived is larger than Base, the pointer arithmetic done on line // 3 is incorrect: the compiler uses sizeof(Base) when computing the address for arrayOfBase[1], yet the array is an array of Derived, which means the address computed on line // 3 (and the subsequent invocation of member function f()) isn’t even at the beginning of any object! It’s smack in the middle of a Derived object. Assuming your compiler uses the usual approach to virtual functions, this will reinterpret the int i\_ of the first Derived as if it pointed to a virtual table, it will follow that “pointer” (which at this point means we’re digging stuff out of a random memory location), and grab one of the first few words of memory at that location and interpret them as if they were the address of a C++ member function, then load that (random memory location) into the instruction pointer and begin grabbing machine instructions from that memory location. The chances of this crashing are very high.
* The root problem is that C++ can’t distinguish between a **pointer-to-a-thing** and a **pointer-to-an-array-of-things**. Naturally C++ “inherited” this feature from C.

# Inheritance — Abstract Base Classes

## What is an ABC?

* An abstract base class.
* At the design level, an abstract base class (ABC) corresponds to an abstract concept. If you asked a mechanic if he repaired vehicles, he’d probably wonder what kind-of vehicle you had in mind. Chances are he doesn’t repair space shuttles, ocean liners, bicycles, or nuclear submarines. The problem is that the term “Vehicle” is an abstract concept (e.g., you can’t build a “Vehicle” unless you know what kind of vehicle to build). In C++, class Vehicle would be an ABC, with Bicycle, SpaceShuttle, etc, being derived classes (an OceanLiner is-a-kind-of-a Vehicle). In real-world OO, ABCs show up all over the place.
* At the programming language level, an ABC is a class that has one or more pure virtual member functions. You cannot make an object (instance) of an ABC.

## How do you define a copy constructor or assignment operator for a class that contains a pointer to a (abstract) base class?

* If the class “owns” the object pointed to by the (abstract) base class pointer, use the **Virtual Constructor Idiom** in the (abstract) base class. As usual with this idiom, we declare a pure virtual clone() method in the base class.

# Inheritance — private and protected inheritance

## How are “private inheritance” and “composition” similar?

* private inheritance is a syntactic variant of composition (AKA aggregation and/or has-a).
* Example: the “Car has-a Engine” relationship can be expressed using simple composition and using *private* inheritance:

// Composition

class Engine

{

public:

Engine(int numCylinders);

// Starts this Engine

void start();

};

class Car

{

public:

// Initializes this Car

// with 8 cylinders

Car() : e\_(8) { }

// Start this Car by starting

// its Engine

void start() { e\_.start(); }

private:

Engine e\_; // Car has-a Engine

};

// private inheritance

// Car has-a Engine

class Car : private Engine

{

public:

// Initializes this Car

// with 8 cylinders

Car() : Engine(8) { }

// Start this Car by starting

// its Engine

using Engine::start;

};

* There are several similarities between these two variants:
  + In both cases there is exactly one Engine member object contained in every Car object.
  + In neither case can users (outsiders) convert a Car\* to an Engine\*.
  + In both cases the Car class has a start() method that calls the start() method on the contained Engine object.
* There are also several distinctions:
  + The simple-composition variant is needed if you want to contain several Engines per Car.
  + The private-inheritance variant can introduce unnecessary multiple inheritance.
  + The private-inheritance variant allows members of Car to convert a Car\* to an Engine\*.
  + The private-inheritance variant allows access to the protected members of the base class.
  + The private-inheritance variant allows Car to override Engine’s virtual functions
  + The private-inheritance variant makes it *slightly* simpler (20 characters compared to 28 characters) to give Car a start() method that simply calls through to the Engine’s start() method.

## Which should I prefer: composition or private inheritance?

* Use composition when you can, private inheritance when you have to.
* Normally you don’t want to have access to the internals of too many other classes, and private inheritance gives you some of this extra power (and responsibility). But private inheritance isn’t evil; it’s just more expensive to maintain, since it increases the probability that someone will change something that will break your code.
* A legitimate, long-term use for private inheritance is when you want to build a class Fred that uses code in a class Bruce, and the code from class Bruce needs to invoke member functions from your new class, Fred. In this case, Fred calls non-virtuals in Bruce, and Bruce calls (usually pure virtuals) in itself, which are overridden by Fred. This would be much harder to do with composition.

class Bruce

{

protected:

void fredCallsBruce()

{

cout << "Bruce::fredCallsBruce()\n";

// Bruce calling pure virtual function

BruceCallsFred();

}

// A pure virtual function

virtual void BruceCallsFred() = 0;

};

class Fred : private Bruce

{

public:

void barney()

{

cout << "Fred::barney()\n";

Bruce::fredCallsBruce();

}

protected:

virtual void BruceCallsFred()

{

cout << "Fred::BruceCallsFred()\n";

}

};

## Why would the conversion between derived\* to base\* fails with private inheritance?

* A base class B of N is accessible at R, if
  + an invented public member of B would be a public member of N, or
  + R occurs in a member or friend of class N, and an invented public member of B would be a private or protected member of N, or

int main(void)

{

// Error

Bruce\* b = new Fred;

return 0;

}

* + R occurs in a member or friend of a class P derived from N, and an invented public member of B would be a private or protected member of P, or
  + there exists a class S such that B is a base class of S accessible at R and S is a base class of N accessible at R."
* Here 'B' is 'Base', 'N' is 'Derived' and 'R' is main.
  1. Consider the 2nd bullet- 'R occurs in a member or friend of a class N,...'. This clause does not apply as 'R'(main) is neither a member nor friend of 'N'(Derived)
  2. Consider the 3rd bullet- 'R occurs in a member or friend of a class P....'. This Claus also does not apply for the same reasons as above
  3. Consider the 4th bullet- Once again this clause does not apply
* Thus we can conclude that 'Base' is not an accessible class of 'Derived'.
* If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class [$11.2/5].

# Inheritance — Multiple and Virtual Inheritance

## What is a virtual base class?

* A virtual base class, used in virtual inheritance, is a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritance.
* Example:

class A { public: void Foo() {} };

class B : public A {};

class C : public A {};

class D : public B, public C {};

* The above class hierarchy results in the "***dreaded diamond***" which looks like this:
* An instance of D will be made up of B, which includes A, and C which also includes A. So you have two "instances" (for want of a better expression) of A.

          +-----+

   +----->+  A  +<------+

   |      +-----+       |

   |                    |

   |                    |

+--+--+              +--+--+

|  B  |              |  C  |

+--+--+              +--+--+

   ^                    ^

   |                    |

   |                    |

   |       +-----+      |

   +-------|  D  |------+

           +-----+

* When you have this scenario, you have the possibility of ambiguity. What happens when you do this:

D d;

d.Foo(); // is this B's Foo() or C's Foo() ??

* Virtual inheritance is there to solve this problem. When you specify virtual when inheriting your classes, you're telling the compiler that you only want a single instance.

class A { public: void Foo() {} };

class B : public virtual A {};

class C : public virtual A {};

class D : public B, public C {};

* This means that there is only one "instance" of A included in the hierarchy. Hence

D d;

d.Foo(); // no longer ambiguous

## What is the exact order of destructors in a multiple and/or virtual inheritance situation?

* Short answer: the exact opposite of the constructor order.

class VirtualBaseA;

class VirtualB : virtual VirtualBaseA;

class VirtualC : virtual VirtualBaseA;

class NonVirtualClass1;

class NonVirtualClass2;

class DataMember1;

class DataMember2;

class MostDerived : VirtualB, VirtualC, NonVirtualClass1, NonVirtualClass2

{

public:

DataMember1 a;

DataMember2 b;

MostDerived(){ cout << "MostDerived\n\n"; }

~MostDerived(){ cout << "~MostDerived\n"; }

};

##############Destruction##############

~MostDerived

~DataMember2

~DataMember1

~NonVirtualClass2

~NonVirtualClass1

~VirtualC

~VirtualB

~VirtualBaseA

#######################################

##############Construction##############

VirtualBaseA

VirtualB

VirtualC

NonVirtualClass1

NonVirtualClass2

DataMember1

DataMember2

MostDerived

########################################

* The sub-object corresponding to most-derived class Most Derived runs first, followed by the destructor for its Non Virtual Class base classes in reverse declaration-order.
* The destructors for the virtual base classes are executed in the reverse order they appear in a depth-first left-to-right traversal of the graph of base classes, where left to right refer to the order of appearance of base class names.

# Templates

## What’s the idea behind templates?

* A template is a cookie-cutter that specifies how to cut cookies that all look pretty much the same.
* In the same way, a class template is a cookie cutter for a description of how to build a family of classes that all look basically the same and a function template describes how to build a family of similar looking functions.

## How do I explicitly select which version of a function template should get called?

* When you call a function template, the compiler tries to deduce the template type. Most of the time it can do that successfully, but every once in a while you may want to help the compiler deduce the right type - either because it cannot deduce the type at all, or perhaps because it would deduce the wrong type.
* For example, you might be calling a function template that doesn’t have any parameters of its template argument types, or you might want to force the compiler to do certain promotions on the arguments before selecting the correct function template.

template<typename T>

void aFunction(void) {

cout << "In function template: " << endl;

}

template<typename T>

int bFunction(T x, T y) {

return x + y;

}

int main(void) {

aFunction<int>(); // type T will be int

aFunction<std::string>(); // type T will be std::string

int m = 0;

long n = 1;

// no instance of function template "bFunction"

// matches the argument list argument types are: (int, long)

// bFunction(m, n);

bFunction<int>(m, n);

return 0;

}

## What is a “parameterized type”?

* Another way to say, “class templates.”
* A parameterized type is a type that is parameterized over another type or some value. List<int> is a type (List) parameterized over another type (int).

## What is “genericity”?

* Yet another way to say, “class templates.”

## What is SFINAE?

* **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror.
* In the process of template argument deduction, a C++ compiler attempts to instantiate signatures of a number of candidate overloaded functions to make sure that exactly one overloaded function is available as a perfect match for a given function call. If an invalid argument or return type is formed during the instantiation of a function template, the instantiation is removed from the overload resolution set instead of causing a compilation error. As long as there is one and only one function to which the call can be dispatched, the compiler issues no errors.
* Example:
* Calling function multiply in main causes the compiler to instantiate the signature of the templatized function even though the first multiply function is a better match. During instantiation an ***invalid*** type is produced: int::multi\_result. Due to **SFINAE**, however, the invalid instantiation is neglected automatically. At the end, there is exactly one multiply function that can be called. So the compilation is successful.

long multiply(int i, int j)

{

return i \* j;

}

template <class T>

// Note the T::multi\_result

typename T::multi\_result multiply(T t1, T t2)

{

return t1 \* t2;

}

int main(void)

{

multiply(4, 5);

return 0;

}

* SFINAE is often exploited in determining properties of types at compile-time.
* Example, consider the following isPointer meta-function that determines at compile-time if the given type is a pointer of some sort.

template <class T>

struct isPointer

{

template <class U>

static char is\_ptr(U \*);

template <class X, class Y>

static char is\_ptr(Y X::\*);

template <class U>

static char is\_ptr(U(\*)());

static double is\_ptr(...);

static T t;

enum { value = sizeof(is\_ptr(t)) == sizeof(char) };

};

struct Foo {

int bar;

};

int main(void)

{

typedef int \* IntPtr;

typedef int Foo::\* FooMemberPtr;

typedef int(\*FuncPtr)();

cout << isPointer<IntPtr>::value << endl;       // prints 1

cout << isPointer<FooMemberPtr>::value << endl; // prints 1

cout << isPointer<FuncPtr>::value << endl;      // prints 1

cout << isPointer<int>::value << endl; // prints 0

return 0;

}

* The isPointer meta-function above would not work without SFINAE. It defines 4 overloaded is\_ptr functions, three of which are templates that accept one argument each: a pointer to a variable, pointer to a member variable, or a simple function pointer. All three functions return a char, which is deliberate. The last is\_ptr function is a catch-all function that uses ellipsis as it parameter. This function, however, returns a double, which is always greater in size compared to a character.
* When the isPointer is passed a type that is really a pointer (e.g., IntPtr), value is initialized to true as a result of the comparison of two sizeof expressions. The first sizeof expression calls is\_ptr. If at all it is a pointer, only one of the overloaded template functions match and not others. Due to SFINAE, however, no error is raised because at least one function is found to be suitable. If none of the functions are suitable, the function with ellipsis is used instead. That function however, returns a double, which is larger than a character and so the value is initialized to false as the comparison of sizeof fails.
* Note that, none of the is\_ptr functions have definitions. Only declarations are sufficient to trigger SFINAE rule in the compiler. Those functions themselves must be templates, however. That is, a class template with regular functions will not participate in SFINAE. The functions that participate in SFINAE must be templates.

## Why can’t I separate the definition of my templates class from its declaration and put it inside a .cpp file?

* A template is not a class or a function. A template is a “pattern” that the compiler uses to generate a family of classes or functions.
* In order for the compiler to generate the code, it must see both the template definition (not just declaration) and the specific types/whatever used to “fill in” the template. For example, if you’re trying to use a Foo<int>, the compiler must see both the Foo template and the fact that you’re trying to make a specific Foo<int>.
* Your compiler probably doesn’t remember the details of one .cpp file while it is compiling another .cpp file. It could, but most do not and if you are reading this FAQ, it almost definitely does not. BTW this is called the “separate compilation model.”

# Pointers to Member Functions

## How can I avoid syntax errors when creating pointers to members?

* Use a typedef.

## How can I avoid syntax errors when calling a member function using a pointer-to-member-function?

* Use a #define macro.

class Fred

{

public:

int f(char x, int y) { return x + y; }

int g(char x, int y) { return x \* y; }

int h(char x, int y) { return x - y; }

int i(char x, int y) { return x / y; }

};

typedef int (Fred::\*ptrToMemFun) (char x, int y);

#define CALL\_MEMBER\_FN(Fred,ptrToMemFun) ((Fred).\*(ptrToMemFun))

int main()

{

ptrToMemFun p = &Fred::f;

Fred a;

int x = CALL\_MEMBER\_FN(a, p)('c', 9);

cout << x << endl;

return 0;

}

## How do I create and use an array of pointer-to-member-function?

* Use both the typedef and the #define macro described earlier.

typedef int (Fred::\*ptrToMemFun) (char x, int y);

#define CALL\_MEMBER\_FN(Fred,ptrToMemFun) ((Fred).\*(ptrToMemFun))

ptrToMemFun a[] = { &Fred::f, &Fred::g, &Fred::h, &Fred::i };

## How do I declare a pointer-to-member-function that points to a const member function?

class Fred

{

public:

int f(int i) const { return i + 1; }

int g(int i) const { return i + 2; }

int h(int i) const { return i + 3; }

};

// const here

typedef int (Fred::\*ptrToMemFun)(int) const;

#define CALL\_MEMBER\_FN(Fred,ptrToMemFun) ((Fred).\*(ptrToMemFun))

int main()

{

ptrToMemFun p = &Fred::f;

Fred a;

int x = CALL\_MEMBER\_FN(a, p)('c');

cout << x << endl;

return 0;

}

## What is the difference between the .\* and ->\* operators?

class Fred {

public:

int f(char x, int y) { return x + y; }

};

typedef int (Fred::\*ptrToMemFun) (char x, int y);

void sample(Fred x, Fred& y, Fred\* z, ptrToMemFun func){

cout << (x.\*func)(42, 3) << endl;

cout << (y.\*func)(42, 2) << endl;

cout << (z->\*func)(42, 1) << endl;

}

int main() {

ptrToMemFun p = &Fred::f;

Fred a;

sample(a, a, &a, p);

return 0;

}

* use .\* when the left-hand argument is a reference to an object, and ->\* when it is a pointer to an object.

## Can I convert a pointer-to-member-function to a void\*?

* No! Technical details: pointers to member functions and pointers to data are not necessarily represented in the same way. A pointer to a member function might be a data structure rather than a single pointer.

## Can I convert a pointer-to-function to a void\*?

* No! Technical details: void\* pointers are pointers to data, and function pointers point to functions. The language does not require functions and data to be in the same address space, so, by way of example and not limitation, on architectures that have them in different address spaces, the two different pointer types will not be comparable.

Further reading: functionoid and functor