9.11 — The copy constructor

BY ALEX ON NOVEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Recapping the types of initialization

Since we're going to talk a lot about initialization in the next few lessons, let's first recap the types of initialization that C++ supports: direct initialization, uniform initialization or copy initialization.

Here's examples of all of those, using our Fraction class:

```
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}
```

We can do a direct initialization:

```
int x(5); // Direct initialize an integer
Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor
```

In C++11, we can do a uniform initialization:

```
int x { 5 }; // Uniform initialization of an integer
Fraction fiveThirds {5, 3}; // Uniform initialization of a Fraction, calls Fraction(int, int) constructor
```

And finally, we can do a copy initialization:

```
int x = 6; // Copy initialize an integer
Fraction six = Fraction(6); // Copy initialize a Fraction, will call Fraction(6, 1)
Fraction seven = 7; // Copy initialize a Fraction.  The compiler will try to find a way to convert 7 to a Fraction, which will invoke the Fraction(7, 1) constructor.
```

With direct and uniform initialization, the object being created is directly initialized. However, copy initialization is a little more complicated. We'll explore copy initialization in more detail in the next lesson. But in order to do that effectively, we need to take a short detour.

The copy constructor

Now consider the following program:

```cpp
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor
    Fraction fCopy(fiveThirds); // Direct initialize -- with what constructor?
    std::cout << fCopy << '\n';
}
```

If you compile this program, you'll see that it compiles just fine, and produces the result:

```
5/3
```

Let's take a closer look at how this program works.

The initialization of variable fiveThirds is just a standard direct initialization that calls the Fraction(int, int) constructor. No surprises there. But what about the next line? The initialization of variable fCopy is also clearly a direct initialization, and you know that constructor functions are used to initialize classes. So what constructor is this line calling?

The answer is that this line is calling Fraction's copy constructor. A **copy constructor** is a special type of constructor used to create a new object as a copy of an existing object. And much like a default constructor, if you do not provide a copy constructor for your classes, C++ will create a public copy constructor for you. Because the compiler does not know much about your class, by default, the created copy constructor utilizes a method of initialization called memberwise initialization. **Memberwise initialization** simply means that each member of the copy is initialized directly from the member of the class being copied. In the above example, fCopy.m_numerator would be initialized from fiveThirds.m_numerator, etc...

Just like we can explicitly define a default constructor, we can also explicitly define a copy constructor. The copy constructor looks just like you'd expect it to:

```cpp
#include <cassert>
#include <iostream>
```

```cpp

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    // Copy constructor
    Fraction(const Fraction &fraction) :
        m_numerator(fraction.m_numerator), m_denominator(fraction.m_denominator)
        // Note: We can access the members of parameter fraction directly, because we're inside the Fraction class
    {
        // no need to check for a denominator of 0 here since fraction must already be a valid Fraction
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor
    Fraction fCopy(fiveThirds); // Direct initialize -- with Fraction copy constructor
    std::cout << fCopy << '\n';
}
```

When this program is run, you get:

```
Copy constructor called
5/3
```

The copy constructor we defined in the example above uses memberwise initialization, and is functionally equivalent to the one we'd get by default, except we've added an output statement to prove the copy constructor is being called.

Unlike with default constructors (where you should always provide your own default constructor), it's fine to use the default copy constructor if it meets your needs.

One interesting note: You've already seen a few examples of overloaded operator<<, where we're able to access the private members of parameter f1 because the function is a friend of the Fraction class. Similarly, member functions of a class can access the private members of parameters of the same class type. Since our Fraction copy constructor takes a parameter of the class type (to make a copy of), we're able to access the members of parameter fraction directly, even though it's not the implicit object.

Preventing copies

We can prevent copies of our classes from being made by making the copy constructor private:

```cpp
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

    // Copy constructor (private)
    Fraction(const Fraction &fraction) :
        m_numerator(fraction.m_numerator), m_denominator(fraction.m_denominator)
    {
        // no need to check for a denominator of 0 here since fraction must already be a vali
d Fraction
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) cons
tructor
    Fraction fCopy(fiveThirds); // Copy constructor is private, compile error on this line
    std::cout << fCopy << '\n';
}
```

Now when we try to compile our program, we'll get a compile error since fCopy needs to use the copy constructor, but can not see it since the copy constructor has been declared as private.

The copy constructor may be elided

Now consider the following example:

```cpp
#include <cassert>
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        assert(denominator != 0);
    }

        // Copy constructor
    Fraction(const Fraction &fraction) :
        m_numerator(fraction.m_numerator), m_denominator(fraction.m_denominator)
    {
        // no need to check for a denominator of 0 here since fraction must already be a vali
d Fraction
        std::cout << "Copy constructor called\n"; // just to prove it works
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    Fraction fiveThirds(Fraction(5, 3));
    std::cout << fiveThirds;
    return 0;
}
```

Consider how this program works. First, we direct initialize an anonymous Fraction object, using the Fraction(int, int) constructor. Then we use that anonymous Fraction object as an initializer for Fraction fiveThirds. Since the anonymous object is a Fraction, as is fiveThirds, this should call the copy constructor, right?

Run this and compile it for yourself. You'd probably expect to get this result (and you may):

```
copy constructor called
5/3
```

But in actuality, you're more likely to get this result:

```
5/3
```

Why didn't our copy constructor get called?

Note that initializing an anonymous object and then using that object to direct initialize our defined object takes two steps (one to create the anonymous object, one to call the copy constructor). However, the end result is essentially identical to just doing a direct initialization, which only takes one step.

For this reason, in such cases, the compiler is allowed to opt out of calling the copy constructor and just do a direct initialization instead. This process is called **elision**.

So although you wrote:

```
Fraction fiveThirds(Fraction(5, 3));
```

The compiler may change this to:

```
Fraction fiveThirds(5, 3);
```

which only requires one constructor call (to Fraction(int, int)). Note that in cases where elision is used, any statements in the body of the copy constructor are not executed, even if they would have produced side effects (like printing to the screen)!

Prior to C++17, copy elision is an optimization the compiler can make. As of C++17, some cases of copy elision (including the example above) have been made mandatory.

Finally, note that if you make the copy constructor private, any initialization that would use the copy constructor will cause a compile error, even if the copy constructor is elided!

9.12 -- Copy initialization

Index

9.10 -- Overloading typecasts

C++ TUTORIAL | PRINT THIS POST

64 comments to 9.11 — The copy constructor

Tompouce
November 30, 2019 at 3:07 am · Reply

Hello!
At the very end of this saction you said that trying to initialise an object in a way that would call the copy constructor would fail if the constructor was made private.
But I have notied that it doesn't seem to apply when giving an annonymous object to the private copy constructor.
Here's what I mean:

```
class Foo
{
private:
    Foo(const Foo &foo) { } // the copy constructor is private
};

Foo foo; // Making a named Foo object with the default constructor
Foo fooCopy(foo); // Produces an arror as expected

Foo otherFooCopy(Foo()); // Giving the private copy constructor an annonymous Foo object
                         // doesn't give an error, it just creates otherFooCopy just fine.
```

This seems like an inconsistency, is there an explanation for this? Or maybe I'm missing some detail.

Edit: Here I'm using direct initialisation to define otherFooCopy, but using copy or uniform initialisation exhibits the same behaviour.

nascardriver
November 30, 2019 at 5:10 am · Reply

```
Foo otherFooCopy(Foo());
```

That's a forward declaration of a function, not a variable.

You can create a variable with brace initialization

```
Foo otherFooCopy{ Foo{} };
```

This works, even if the copy constructor is private, because it doesn't use the copy constructor. If you're initializing an object of type `Foo` with a temporary `Foo`, the copy has to be elided (since C++17). Before C++17, you'd get an error, because the copy constructor is private.

Tompouce
November 30, 2019 at 5:59 am · Reply

Wow, I would have never guesed that this was a function forward declaration x)
(unless I had looked more closely at what my IDE was saying :p...)

So it's a forward declaration for a function that returns a Foo and takes... a function with no parameters, and returning Foo? Did I get this right?

nascardriver
November 30, 2019 at 6:07 am · Reply

Right

Puya
February 22, 2019 at 11:55 am · Reply

Hi Alex,
As always thank you for the great tutorials.
1- Would there ever be a point in defining the body of the copy constructor if it is private? Since if it is ever used the compiler will give an error. (except having one public copy constructor call a private one)
2- Is declaring a constructor as private the same as deleting it?
3- You might want to change `...since copy must already be a valid Fraction...` to `...since fraction must already be a valid Fraction...`.

Alex
February 22, 2019 at 4:44 pm · Reply

1) A private copy constructor could still be used by any function that has access to private member functions, which includes member functions and friend functions.
2) No, for the same reason as #1. A deleted copy constructor can't be used by anybody.
3) Thanks, fixed!

Big Chungus
January 5, 2019 at 9:31 pm · Reply

```cpp
#include "pch.h"
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cassert>

class Dollars
{
    int m_dollars;
public:
    Dollars(int dollars = 0) : m_dollars(dollars)
    {
    }
    int getDollars()
    {
        return m_dollars;
    }
};
class Cents
{
    int m_cents;
public:
    Cents(int cents = 0): m_cents(cents)
    {

    }
    operator int() { return m_cents; }
    int getcents() { return m_cents; }
    void setcents(int cents) { m_cents = cents; }

    Cents(Dollars &d) : m_cents(d.getDollars() * 100)
    {
        std::cout << "COPY CONSTRUCTOR";
    }
};

void printInt(int value)
{
    std::cout << value;
}

int main()
{
    Dollars d(1233);
    Cents c(d);
    std::cout << c.getcents();
    return 0;
}
```

Look what I did with two class types!

lucieon
February 1, 2019 at 9:26 am · Reply

you could just write:

```
std::cout << c;
```

in main as you have overloaded the int typecast.

btnal

September 25, 2018 at 6:46 pm · Reply

When a class has properties that are pointers, what is the problem of using the default copy constructor? How do you handle the problem?

nascardriver

September 26, 2018 at 12:39 am · Reply

Hi btnal!

You need to perform a deep copy. Covered in lesson 9.15.

Ran

March 23, 2018 at 12:18 pm · Reply

I found a more creepy things after I read Alex code:

This is my original definition of the operatro+=

```
Averaged& Averaged::operator+=(const Averaged &avg)
{
    this->m_sum += avg.m_sum;
    this->m_number += (avg.m_number+1); // how to improve this +1??

    return *this;
}
[\code]

I treat the right hand side of the equation as an object of the class
Averaged.

To make the code running, I have to write a constructor like this:

[code]
Averaged(int32_t sum = 0, int8_t number = 0): m_sum(sum), m_number(number) {}
```

If I do the solution(using the default constructor) given by Alex, it won't compile.

My code, actually treat the number on the left hand side of that equation as the first private member m_sum using the initialization list. But you can see that, to make the program running, I did an awful thing:

```
    this->m_number += (avg.m_number+1); // how to improve this +1??
```

But I still do not know why "this" pointers still works in the following code?

[code]
Averaged& Averaged::operator+=(const Averaged &avg)
{
  this->m_sum += avg.m_sum;
  this->m_number += (avg.m_number+1); // how to improve this +1??

return *this;
}
[\code]

How two "this" work good?

nascardriver
March 24, 2018 at 3:43 am · Reply

Hi Ran!

> This is my original definition of the operator+=
When you only have the operator

```
Averaged& Averaged::operator+=(const Averaged &avg)
```

and write

```
a += 8; // Where @a is an @Averaged
```

the constructor of @Averaged will be called, 8 is converted to an @Averages, then the operator+= is called. This is not what you want. You should write an operator+= that takes an int.

> But I still do not know why "this" pointers still works in the
following code?
I can't think of an example where this is actually useful. All it does is allow you to use the object right after using the operator:

```
@a, @b, @c and @d are @Averaged objects
a = (b += c) + d;
// (b += c) returns @b so we can add @d is the same line.
```

Aitor
December 1, 2017 at 1:09 pm · Reply

Hi Alex,
I have encounter one issue that i don't understand with the following code. It could be great if you can find some time to clarify my doubt. Having the code below:

```
#include <iostream>

class Foo {

   static int objectCounter;
    int objectID = 0;
public:
    Foo() {
         objectID = objectCounter;
         std::cout << "constructor: " << objectID<< std::endl;
         ++objectCounter;
    }

    Foo( const Foo& rhs){
        objectID = objectCounter;
        std::cout << "CopyConstructor: " << objectID << std::endl;
        ++objectCounter;
    }

    ~Foo(){
        std::cout << "Desctructor: " << objectID << std::endl;
    }

};

// function that takes Foo by value and return it by value
 Foo funcion( Foo c1){
     Foo  localFoo;
     return   localFoo; /* here it should make a copy since its returning the object by value*/
 }


int main() {

   Foo  one ;
   Foo two=  funcion(one);

    return 0;
}
```

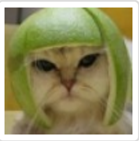
The output of these is the following:

constructor: 0
CopyConstructor: 1
constructor: 2
Desctructor: 1
Desctructor: 2
Desctructor: 0

Because the return type is by value and the localFoo its gonna go out of scope it should be call copy constructor again to my understanding, making a new object with id 3, or even a 4th object.
My questions are the folowing:
- Shouldn't be,object two, call to the constructor after the object localFoo had return?
- Shoudn't be, the copy constructor be called again to make a copy of localFoo when localFoo is   returned to main?
- Is it the copy constructor being also elided in this situation?

Im just confusing myself a bit...hehe

Thanks for your time, and sorry for the inconvenience,Aitor



Alex
December 4, 2017 at 5:27 pm · Reply

Yes, the copy constructor would normally be called once when one is copied into parameter c1, and once when localFoo is used to initialize two. However, given the actual output, it looks like the return value of funcion is getting elided (as a compiler optimization). Try compiling using a debug build target and see if you see the copy constructor get called twice.

Pointer
February 6, 2018 at 11:26 am · Reply

For me this programm does not compile. It compiled when definition of the static variable was added in the global namespace:

```
int Foo::objectCounter;
```

Aitor
February 6, 2018 at 11:38 am · Reply

This is because i forgot to paste it there, sorry about that!!.

Priya
November 29, 2017 at 10:55 pm · Reply

Copy constructors are widely used for creating a duplicates of objects known as cloned objects. Duplicate object in the sense the object will have the same characteristics of the original object from which duplicate object is created.

Akshay
October 15, 2017 at 10:15 am · Reply

Why is copy constructor's argument a 'const'? I read on other sites stating that compiler creates temporary object and to hold them non-const ref would not be a good idea, I could not get it clearly, please can you explain the exact reason in details.

Alex
October 20, 2017 at 2:00 pm · Reply

A non-const reference can only bind to l-values. A const reference can bind to l-values or r-values, which makes it much more flexible. There are cases where we want to copy r-values, and we don't need to modify the argument anyway, so it makes sense to use the const reference.

hit4man47
May 6, 2017 at 8:05 am · Reply

Hey alex,

```
#include<iostream>
using namespace std;
class dummy
{
public:

dummy()
    {
 cout<<"normal";
}

dummy(dummy & x)
    {
cout<<"copy";
}

};

int main()
{
dummy o1;
dummy o2;
    o2=o1;
return 0;
}
```

Why copy constructor is not called??

Alex
May 6, 2017 at 3:15 pm · Reply

Because when o2=o1 is executed, o2 isn't being constructed -- it's being assigned. It was already constructed on the previous line.

If you want to call the copy constructor, create o2 like this: "dummy o2(o1)".

Hit4man47
May 7, 2017 at 11:00 am · Reply

Thanks alex.
You owe me a big one.You are great.
Can't describe you just in words how awesome you are.

deeksha
January 19, 2017 at 1:16 am · Reply

Hi Alex, my question is, Why pass by value is not allowed in copy constructor.
Note:I have read the answers on internet. They say pass by value is pass by copy, means this would call the copy constructor,and it goes in a loop.
But I can't visualise the loop. Could you please make it simpler for me using an example?

e.g.

```
class complex1{
int mem1;
int mem2;
public:
  complex1(int a,int b):mem1(a),mem2(b){}
  complex1(complex1 c){mem1=c.mem1;mem2=c.mem2;}
};


int main(){
complex1 c1(2,3);
complex1 c2(c1); // This won't compile, But my question is what could have happened?
// The c in the copy constructor needs to copy value from c1, so it calls complex1 c(c1).
// And then what happens? What is the loop?
return 0;}
```

Thank you. You are awesome.

Alex
January 19, 2017 at 11:11 am · Reply

First, let's agree that the copy constructor is used to copy one object into another object of the class type. Second, let's agree that the copy constructor is used when passing a class object by value.

Now consider what happens when you instantiate "complex1 c2(c1)". Since c1 is passed by value in this example, argument c1 needs to be copied into constructor parameter c. Per the above, this will result in a call to the copy constructor. If the copy constructor argument was passed by reference, no problem. c would be a reference to c1, and this would be used to initialize object c2.

But instead, you've defined your copy constructor to have the argument passed by value. How do we pass an argument by value? By using the copy constructor! So your copy constructor call would call itself to try and resolve copying the argument into the parameter. This in turn, would call the copy constructor again to resolve the copying of THAT argument into the parameter.

In other words, each call to the copy constructor results in another call to the copy constructor to copy the argument into the parameter. This sets up an infinite series of recursive copy constructor function calls. Consequently, the compiler disallows this.

deeksha
January 20, 2017 at 3:06 am · Reply

Does something like the below happen?

complex1 c2(c1) --> complex1 c(c1) --> complex1 ctemp(c1) --> complex1 c_temp2(c1) --> complex1 ctemp3(c1) -->
complex1 ctemp4(c1) --> complex1 ctemp5(c1) --> complex1 ctemp6(c1) --> and so on, till program crashes

Alex
January 20, 2017 at 9:21 am · Reply

Essentially, yes.

Sihoo
July 8, 2017 at 3:23 pm · Reply

thank you

Rakesh
January 9, 2017 at 10:07 pm · Reply

Hi,

As i was unable to get a convincing answer, posting it here.

Please tell me the consequences(good/bad) of using a pointer in a copy constructor. Why is a reference used most of the times.

Thanks

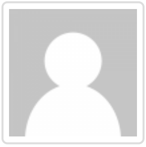
Alex
January 10, 2017 at 10:52 am · Reply

By "using a pointer in a copy constructor", I presume you mean having the copy constructor take a pointer parameter instead of a reference? The short answer is that the C++ specification requires copy constructors to take a reference so it can "copy" objects without having to pass them by address. This makes sense when you consider that references are semantically meant to be aliases for objects -- it makes more sense to pass the actual object to be copied than a pointer to it.

You can certainly create your own constructors that take an object by address, but it wouldn't be a copy constructor.

Rakesh
January 10, 2017 at 5:42 pm · Reply

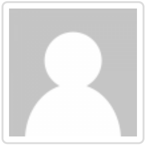
Thanks Alex


Sam
December 12, 2016 at 3:49 pm · Reply

Formatting bug: ctrl+f "Preventing copies"

Alex
December 14, 2016 at 9:33 am · Reply

Fixed, thanks.


Rakesh
December 6, 2016 at 11:22 pm · Reply

Hi,

How can i access the private member of a class without using friend functions or access functions.
Could you please let me know the possible ways to do this.

Soterite
November 12, 2016 at 5:40 pm · Reply

Hi Alex!

Maybe it's just me and I simply cannot grasp something, but shouldn't the copy constructor in your second example look rather like this?:

```
Fraction(const Fraction &original) :
    m_numerator(original.m_numerator), m_denominator(original.m_denominator)
{
    // no need to check for a denominator of 0 here since copy must already be a valid Frac
tion
    std::cout << "Copy constructor called\n"; // just to prove it works
}
```

It seems to me, that the argument of a copy constructor is an original object. The copy constructor is special, because it is called automatically by a compiler in some situations, but in other aspects it behaves like an ordinary one. In fact this code works:

```
class A {
public:
  int someData;
  A(int data) { someData = data; }       // the normal constructor
  A(const A & original) {                // the copy constructor
    cout << "Here I am\n";
    someData = original.someData;
  }
};

A f(A obj) {                             // the copy constructor is going to be called by a
compiler
  return obj;                            // twice
}
```

```

int main()
{
  A first(7);                       // the normal constructor operates
  const A &refFirst = first;        // the const reference is being initialized

  A second(refFirst);               // the copy constructor is called as an ordinary o
ne
  cout << second.someData << "\n";  // and it works similarly to the normal one

  A third = f(second);              // the copy constructor is called twice by a compi
ler
}
```

Please rectify, if I'm wrong with something.

Last but not least thank you VERY much for your work. Outstanding job!

Alex
November 13, 2016 at 2:22 pm · Reply

If you don't provide a copy constructor, the compiler will provide one for you. That's why the second example compiles without one.

Your copy constructor above is essentially the same as mine, except mine initializes the class data and yours assigns it. Your won't work with const members.

Did I miss the point of your question?

Soterite
November 14, 2016 at 5:51 pm · Reply

I'm sorry, I'm not a native speaker and maybe I didn't express myself clearly enough. It's all about naming of the argument to the copy constructor. You named it a "copy" and I simply don't understand why:

```
// Copy constructor (private)
   Fraction(const Fraction &copy) :
       m_numerator(copy.m_numerator), m_denominator(copy.m_denominator)
```

I believe it should be called "original".

As I understand it, the copy constructor is given an original object (via const reference) and "produces" (even if it doesn't return) a copy of that object. So it operates just like any other constructor, in particular it can be called explicitly with const reference.

I find it confusing, that you name this argument a "copy", but maybe it is I, who misses some quirks of this constructor :)

Kind regards

Alex
November 14, 2016 at 10:26 pm · Reply

Ah, I named it copy because it's the object being passed in to make a copy of. But I can see where your confusion is, since it isn't the actual copy, the implicit class is. I'll change the name.

Matt
September 26, 2016 at 2:10 am · Reply

In the third full code example, which implements a copy constructor in the class definition, you left in a comment(in main) which I think should be updated to reflect the change. You wrote: "Fraction fCopy(fiveThirds); // Direct initialize -- with what constructor?"

Alex
September 26, 2016 at 12:08 pm · Reply

Agreed, and done. Thanks for the suggestion.

Anddo
September 3, 2016 at 1:00 am · Reply

Typo

```
std::cout << fCopy < 'n';
```

should be

```
std::cout << fCopy << 'n';
```

Alex
September 4, 2016 at 6:21 pm · Reply

Fixed, thanks!

Darren
June 21, 2016 at 4:45 am · Reply

You can pay good money to have your dangling pointer dereferenced.

ganesh
March 17, 2016 at 11:33 pm · Reply

```
class A{
              private:

                   char **fname;
};
```

How to implement in parameterized and copy constructor????

Alex
March 18, 2016 at 1:15 pm · Reply

Have a read of lesson **9.12 -- Shallow vs. deep copying** and see if that answers your question.

ritz301

February 10, 2016 at 1:43 am · Reply

Excellent tutorial! (y)

jonas
February 2, 2016 at 12:21 pm · Reply

may result in memory leak?

Alex
February 2, 2016 at 1:51 pm · Reply

Worse, it can result in dereferencing a dangling pointer, which is a good way to crash your program.

jonas
February 2, 2016 at 11:55 am · Reply

hi Alex, may you explain the dangerousness of self assignment in more details.(....In these cases, the assignment operator doesn't need to do anything (and if the class uses dynamic memory, it can be dangerous if it does))

Alex
February 2, 2016 at 1:50 pm · Reply

See **this faq**.

Mohammed
December 19, 2015 at 5:02 am · Reply

Dear Alex would you check this code and plz tell me the answer of the following question;
When i use the overrided assignment operator iam returning by value, so i think what should happen when the instruction pointer reaches return engdist(feet,inches)
1- 2 argument constructor is called to create a temp object;
2- Copy constructor is called to create a copy which will be returned;

However what actually happens is that the 2 argument is called but the copy constructor is not called, does this mean that the temp object is the one which returned to e3

```
#include<iostream>
using namespace std;
class engdist
{
    int feet;
    float inches;
    int serial_number;
    static int count;
public:
    engdist():feet(0),inches(0.0F)
    {
    serial_number=++count;
    cout<<"Constructing dist # "<<serial_number<<" of total "<<count<<endl;
    }
    engdist(int f,float i):feet(f),inches(i)
    {
        serial_number=++count;
    cout<<"Constructing dist # "<<serial_number<<" of total "<<count<<endl;
    }
    engdist(engdist& e):feet(e.feet),inches(e.inches)
    {
        cout<<"Overrided copy constructor called"<<endl;
        serial_number=++count;
        cout<<"Constructin object # "<<serial_number<<" of total "<<count<<endl;
    }
    engdist(float x)
    {
        feet=x;
        inches=(x-feet)*12.0;
        serial_number=++count;
    cout<<"Constructing dist # "<<serial_number<<" of total "<<count<<endl;
    }
    void showdist()const
    {
        cout<<"Dist # "<<serial_number<<" = "<<feet<<"-"<<inches<<endl;
    }
    ~engdist()
    {
        cout<<"dESTROYING dist # "<<serial_number<<endl;
        count--;
        cout<<"dist left "<<count<<endl;
    }
    static void showtotal()
    {
        cout<<count<<endl;
    }
    engdist operator =(const engdist&);
};
int engdist::count=0;
engdist engdist::operator =(const engdist& e3)
{
    cout<<"Overrided assignment operator invoked"<<endl;
    feet=e3.feet;
    inches=e3.inches;
    return engdist(feet,inches);
}
int main()
{
    engdist e1(15,4.5F);
    e1.showdist();
    engdist e2(23.5F);
    e2.showdist();
    engdist::showtotal();
    engdist e3=e2;
    e3.showdist();
    e3.showtotal();
    engdist e4;
    e4=e3;
    e4.showdist();
    engdist::showtotal();
    e1.~engdist();
    e2.~engdist();
    e3.~engdist();
    e4.~engdist();
    getchar();
    getchar();
    return 0;
```

```
}
```

Mohammed
December 19, 2015 at 5:33 am · Reply

Guys if some one knows the answer please say it cuz it is driving me crazy
and i have an interview after 1 week and one of the topics is OOP in C++

Alex
December 20, 2015 at 1:38 pm · Reply

I presume you're talking about what happens when we execute "e4=e3". In this case, your overloaded assignment operator is called. Because the assignment operator is returning by value, I would expect the copy constructor to be called at that point. However, compilers are allowed to optimize away copy constructors (even if they have other side effects!), and I think that what may be happening here.

Ankit
November 5, 2015 at 2:55 am · Reply

Hi Alex,

I am working on project and I got a half written code from my previous mate. He has declared copy constructors as private. Is there any specific use or advantage of using copy constructor as private? I am not able to get why someone will declare a copy contructor as private??

Regards
Ankit Dimri

Alex
November 7, 2015 at 11:00 am · Reply

Making the copy constructor private ensures that a copy can't be made of the object. This can be desirable in some cases, such as when you want to ensure only one of an object exists (e.g. a log file).

Siva
October 12, 2015 at 4:32 am · Reply

why copy constructor returns this pointer while overloaded assignment operator returns. Even though both updates in this pointer. Anyone please explain this.

Thanks in Advance,
Siva

Alex
October 12, 2015 at 1:33 pm · Reply

The copy constructor doesn't return anything (because it doesn't need to).

The assignment operator returns *this so you can chain multiple assignments together (a = b = c = d)

Marko
April 8, 2015 at 1:33 pm · Reply

Hi Alex,

I want to thank you for this brilliant tutorial.
It is helping me a lot, I have c/java background and this is all what I needed.
It is going into my head like it already was there, don't really have a sense that I am learning something new, I am reading this easier than a note I would have to write to myself.
Unfortunatly I dont have a money, cause I am still studying :(, but if I get a job, I will deffinitly remembre to this, and pay my debt to society (you :)).

ty::ItIsProbablyObviouslyThatIAmNotNativeEnglishWriter(LOVE YOU)

rohitnitp
February 19, 2012 at 10:37 pm · Reply

how it come while defining copy constructor you used object of the same class to access private data member which belongs to that object only..since now this object as it is already created how can we access private data directly to anywhere else even if it copy constructor of the same class..it will violate the idea of data encapsulation and data hiding..i tried to access m_nCents via GetLength() at that point but compiler has given me error..why is it so,please explain..

saurabh
February 5, 2012 at 4:23 pm · Reply

Here is answer to why dont we use pass by pointer in copy constructor so see following scenario

```
class ABC
{
public:
  ABC(){} /*Simple Constructor*/
  ABC(ABC * b) { }/*Copy Constructor*/
};
int main(int argc, char* argv[]){
  ABC * d1 = NULL;
  ABC    d2 = d1; /*Calling Copy constructor*/
  return 0;
}
```

See above example , if we take pointer type object in copy constructor
two flaws are here
1) We can assign NULL value to simple object of that class
2) see this line ABC d2 = d1;
here d2 is ABC type while d1 is ABC* type means as per rule we are violating basic rules by allowing to assign simple type with pointer type.
If there is any pointer type member variable in side ABC class means DEEP Copy scenario like int * m_iVal; then it will crash out during calling copy constructor by passing NULL object, so stop such mistake at design time we do use const reference object of same class type in copy constructor as a parameter.Hoping this will clear you.

subbaiahtp
August 25, 2011 at 2:40 am · Reply

```
class A
{

```

```
private:

char *fname;
int age;

};

class B
{
private:
A aobj;
char *lname;
int marks;
};
```

Please let me know the B's copy constructor implementation?

subrat
August 26, 2011 at 10:36 pm · Reply

```
class A {
private:
    char *fname;
    int age;
public:
    /*A() {
        std::cout<<"zero ctr An"<fname = new char[strlen(fname)+1];
        this->age=age;
      std::cout<<"1 cstr---a...."<<this<<std::endl;
    }
    A(const A& val){
        age=val.age;
        fname = new char[strlen(val.fname)+1];
        strcpy(fname, val.fname);
        std::cout<<"Cpy Cstr---A...."<<this<<std::endl;
    }

    ~A(){
        delete[] fname;
        std::cout<<"Destr---A..."<<this<lname=new char[strlen(lname)+1];
          this->marks= marks;
          std::cout<<"constrr B......."<<this<<std::endl;
      }
      B(const B& val):aobj(val.aobj) {
        lname=new char[strlen(val.lname)+1];
        strcpy(lname, val.lname);
        marks= val.marks;
        std::cout<<"copy Cstr---B::::"<<this<<std::endl;

    }

    ~B(){
        delete[] lname;
        std::cout<<"Destr---B:::::"<<this<<std::endl;
    }
};
```

Sanjeev

August 22, 2010 at 11:38 pm · Reply

Hi,
For the class A can I have copy constructor like A(const A*) instead of A(const A&). What will be the difference. Can anybody please explain?

HYA
June 6, 2012 at 5:21 am · Reply

Hi Sanjeev
In that case you have to write any object intialization like this A a, aa1 = &a; ... which is a bit confusing ...but you can implement both copy contructors as

```
 class Aclass
{
    int i ;
public:
    Aclass(){   i=37;   }

    Aclass (const Aclass* a){
        this->i= a->i ;
        cout<i= a.i ;
        cout<<"Copy contructor with reference called";
    }
    int Geti() {return i;}
};

int _tmain(int argc, _TCHAR* argv[])
{
    Aclass aa;
    Aclass aa1 = &aa ;
    Aclass aa2 = aa;

    cout << aa1.Geti();
}
```

HYA
June 6, 2012 at 5:24 am · Reply

```
class Aclass
{
    int i ;
public:
    Aclass(){   i=37;   }

    Aclass (const Aclass* a){
        this->i= a->i ;
        cout<i= a.i ;
        cout<<"Copy contructor with reference called";
    }
    int Geti() {return i;}
};
```