8.14 — Anonymous objects

BY ALEX ON DECEMBER 27TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In certain cases, we need a variable only temporarily. For example, consider the following situation:

```cpp
#include <iostream>

int add(int x, int y)
{
    int sum = x + y;
    return sum;
}

int main()
{
    std::cout << add(5, 3);

    return 0;
}
```

In the add() function, note that the sum variable is really only used as a temporary placeholder variable. It doesn't contribute much -- rather, its only function is to transfer the result of the expression to the return value.

There is actually an easier way to write the add() function using an anonymous object. An **anonymous object** is essentially a value that has no name. Because they have no name, there's no way to refer to them beyond the point where they are created. Consequently, they have "expression scope", meaning they are created, evaluated, and destroyed all within a single expression.

Here is the add() function rewritten using an anonymous object:

```cpp
#include <iostream>

int add(int x, int y)
{
    return x + y; // an anonymous object is created to hold and return the result of x + y
}

int main()
{
    std::cout << add(5, 3);

    return 0;
}
```

When the expression `x + y` is evaluated, the result is placed in an anonymous object. A copy of the anonymous object is then returned to the caller by value, and the anonymous object is destroyed.

This works not only with return values, but also with function parameters. For example, instead of this:

```cpp
void printValue(int value)
{
    std::cout << value;
}

int main()
{
    int sum = 5 + 3;
    printValue(sum);
    return 0;
}
```

We can write this:

```cpp
int main()
{
    printValue(5 + 3);
    return 0;
}
```

In this case, the expression 5 + 3 is evaluated to produce the result 8, which is placed in an anonymous object. A copy of this anonymous object is then passed to the printValue() function, (which prints the value 8) and then is destroyed.

Note how much cleaner this keeps our code -- we don't have to litter the code with temporary variables that are only used once.

Anonymous class objects

Although our prior examples have been with built-in data types, it is possible to construct anonymous objects of our own class types as well. This is done by creating objects like normal, but omitting the variable name.

```cpp
Cents cents(5); // normal variable
Cents(7); // anonymous object
```

In the above code, `Cents(7)` will create an anonymous Cents object, initialize it with the value 7, and then destroy it. In this context, that isn't going to do us much good. So let's take a look at an example where it can be put to good use:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

void print(const Cents &cents)
{
   std::cout << cents.getCents() << " cents";
}

int main()
{
    Cents cents(6);
    print(cents);

    return 0;
}
```

Note that this example is very similar to the prior one using integers. In this case, our main() function is passing a Cents object (named cents) to function print().

We can simplify this program by using anonymous objects:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

void print(const Cents &cents)
{
   std::cout << cents.getCents() << " cents";
}

int main()
{
    print(Cents(6)); // Note: Now we're passing an anonymous Cents value

    return 0;
}
```

As you'd expect, this prints:

```
6 cents
```

Now let's take a look at a slightly more complex example:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

Cents add(const Cents &c1, const Cents &c2)
{
    Cents sum = Cents(c1.getCents() + c2.getCents());
    return sum;
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    Cents sum = add(cents1, cents2);
    std::cout << "I have " << sum.getCents() << " cents." << std::endl;

    return 0;
}
```

In the above example, we're using quite a few named Cents values. In the add() function, we have a Cents value named sum that we're using as an intermediary value to hold the sum before we return it. And in function main(), we have another Cents value named sum also used as an intermediary value.

We can make our program simpler by using anonymous values:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

Cents add(const Cents &c1, const Cents &c2)
{
    return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
}

int main()
{
    Cents cents1(6);
    Cents cents2(8);
    std::cout << "I have " << add(cents1, cents2).getCents() << " cents." << std::endl; // print anonymous Cents value

    return 0;
}
```

This version of add() functions identically to the one above, except it uses an anonymous Cents value instead of a named variable. Also note that in main(), we no longer use a named “sum” variable as temporary storage. Instead, we use the return value of add() anonymously!

As a result, our program is shorter, cleaner, and generally easier to follow (once you understand the concept).

In fact, because cents1 and cents2 are only used in one place, we can anonymize this even further:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

Cents add(const Cents &c1, const Cents &c2)
{
    return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
}
```

```

int main()
{
    std::cout << "I have " << add(Cents(6), Cents(8)).getCents() << " cents." << std::endl; // print anonymous Cents value

    return 0;
}
```

Summary

In C++, anonymous objects are primarily used either to pass or return values without having to create lots of temporary variables to do so. Memory allocated dynamically is also done so anonymously (which is why its address must be assigned to a pointer, otherwise we'd have no way to refer to it).

However, it is worth noting that anonymous objects are treated as rvalues (not lvalues, which have an address) -- therefore, all rules about passing and returning rvalues apply.

It is also worth noting that because anonymous objects have expression scope, they can only be used once. If you need to reference a value in multiple expressions, you should use a named variable instead.

Note: Some compilers, such as Visual Studio, will let you set non-const references to anonymous objects. This is non-standard behavior.

8.15 -- Nested types in classes

Index

8.13 -- Friend functions and classes

C++ TUTORIAL | PRINT THIS POST

116 comments to 8.14 — Anonymous objects

« Older Comments 1 2

Fang
June 25, 2020 at 2:44 pm · Reply

```
1  Cents add(const Cents &c1, const Cents &c2)
2  {
3      return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
4  }
5
6  int main()
7  {
8      std::cout << "I have " << add(Cents(6), Cents(8)).getCents() << " cents." << std::endl; // print anonymous Cents value
9
10     return 0;
11 }
```

In line 8 above, doesn't `add()` return rvalue (since it's not returning a reference type)?

Are we able to chain it with `.getCents()` because member access operator (operator.) also accepts rvalue?

nascardriver
June 26, 2020 at 7:39 am · Reply

Everything correct. There's a way to make member functions not work with rvalues, but by default member functions work with rvalues.

Nguyen
May 13, 2020 at 9:42 am · Reply

Both examples below have the same line of Cents sum = 178. It gives an error to struct but no issue to class. Could you please explain the difference between them? Thank you.

```
//Example of a struct
#include <iostream>
struct Cents
{
    int m_cents;
};

Cents add()
{
    Cents sum = 178;
    return sum;
}

int main()
{
    Cents sum = add();
    std::cout << "I have " << sum.m_cents << " cents." << std::endl;

    return 0;
}

//Example of a class
#include <iostream>
class Cents
{
private:
    int m_cents;

public:
   Cents(int cents) { m_cents = cents; }
   int getCents() { return m_cents; }
};

Cents add()
{
    Cents sum = 178;
    return sum;
}

int main()
{
    Cents sum = add();
    std::cout << "I have " << sum.getCents() << " cents." << std::endl;

    return 0;
}
```

nascardriver
May 14, 2020 at 4:50 am · Reply

You can't initialize a struct like that. You need to use curly braces, because it's an aggregate type. Your class on the other hand has a constructor that takes a single `int`, that's fine.

```
Cents sum{ 178 }; // Works for both
```

Constant_n
May 4, 2020 at 12:54 am · Reply

I guess it works even if we convey only arguments to the function (without Cents objects).

```
std::cout << "I have " << add(6, 8).getCents() << " cents.\n";
```

mesut
April 25, 2020 at 3:20 am · Reply

```
Cents c1{10} // line A
Cents c1 = Cents(10); // line B
```

Hi!
Are line A and B same?

Alex
May 1, 2020 at 2:59 pm · Reply

No. A uses direct initialization, B uses copy initialization.

In this case, the end-result will be the same, but they use different mechanisms and may exhibit non-identical results in other cases.

Kwonk
March 28, 2020 at 3:48 pm · Reply

Hello,

Excuse me if this question is already answered in one of the lessons that is about this topic, but I don't think it is.

```
class Foo
{
    Foo(int);
};

Foo returnFooSomehow();

int main(){
    Foo instance = returnFooSomehow();
    return 0;
};
```

Could elision occur in this case (where an object is copy initialized by an anounymous object returned from a function)?
(I assumed the definitions for Foo::Foo(int) and returnFooSomehow() are unimportant)

nascardriver
March 29, 2020 at 12:57 am · Reply

The definition of `returnFooSomehow` is all that matters.

```
Foo returnFooSomehow()
{
  // Elision guaranteed
  return 3;
  return Foo{ 3 };

  // Elision not guaranteed, but likely due to compiler optimization.
  Foo foo{ 3 };
  return foo;
}
```

HaudreN
November 14, 2019 at 5:28 am · Reply

```
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents) { m_cents = cents; }

    int getCents() const { return m_cents; }
};

Cents add(const Cents &c1, const Cents &c2)
{
    return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
}

int main()
{

 /*1*/   std::cout << "I have " << add(Cents(6), Cents(8)).getCents() << " cents." << std::endl; // print anonymous Cents value
 //More code here
    return 0;
}
```

Hello. In the following example are the two anonymous objects removed from the stack after /*1*/ is complete or are they removed when the function they are created in ends?

nascardriver
November 14, 2019 at 5:39 am · Reply

They die at the end of the `add` call

```
std::cout << "I have " << add(Cents(6), Cents(8)).getCents()
//                                              ^ dead here
```

The use of a stack isn't standardized. The objects may or may not remain on the stack.

hellmet
October 30, 2019 at 10:16 am · Reply

```
1   ...
2   Cents add(const Cents &c1, const Cents &c2)
3   {
4       return Cents(c1.getCents() + c2.getCents());  // <- This line here (2)
5   }
6
7   int main()
8   {
9       Cents cents1(6);
10      Cents cents2(8);
11      Cents centsSum = add(cents1 + cents2); // <- This line here (1)
12      std::cout << "I have " << centsSum.getCents() << " cents." << std::endl;
13
14      return 0;
15  }
```

In the above snippet of code (excerpt from the lesson), this is what I think is happening.

(1) allocates memory on the 'main' function stack for a Cents object named centsSum. The function call (to operator+) returns a Cent object. This creates an anonymous object at the return location whose values are copied into centsSum and then the anonymous object is promptly destroyed. In this case, at mark (1), there are 2 copies happening, one from function to main's stack, one from stack to the variable centsSum.

If I uniform-initialize, a copy is returned to main's stack which overwrites the memory that is used by centsSum, so effectively, only one copy is made and centsSum is updated. This overwrite doesn't break the memory layout or cause undefined behavior as both destination and source object are of the same 'size' and 'shape' in memory.

Have I gotten this right? Do I need to revisit any chapters? Any wild misconceptions I have here?
Also, How is the copy happening?

nascardriver
October 31, 2019 at 1:47 am · Reply

> Have I gotten this right?
I suspect the lessons still use old rules. Before C++17, 2 copies were created. There were some changes about copy/brace initialization, but I won't look them up because it doesn't matter anymore.
Since C++17, no copies are created, because line 4 creates an object of the same type as the function's return type. `add` constructs the new `Cents` object directly into `main`'s memory.

If you remember where you read about about the copy rules, please let me know so I can update it.

You might also be confused by "anoonymous object" not necessarily meaning that there is an object at all. From a language point of view, line 4 is an anonymous object. It's there, it doesn't have a name, but it's not really there after compilation.

hellmet
October 31, 2019 at 2:24 am · Reply

> If you remember where you read about the copy rules, please let me know so I can update it.
Hmmm, I would say the first 4 lessons of chapter 2 (it's there in some of the explanations, especially 2.4), 7.4 (and the lessons around this lesson).
The copies have been stated, but perhaps a nice summary after chapter 7 (also taking few lessons from chapter 2) on how functions return, how they are actually stored back into main's frame would be great! (since by chapter 7, we learnt C++ only does return by value, learnt about the stack, learnt about scope and user defined datatypes)

> Since C++17, no copies are created, because line 4 creates an object of the same type as the function's return type. `add` constructs the new `Cents` object directly into `main`'s memory.

Does this mean that with and post C++17, the 2-copy method (return by value + copy init) and 1-copy method (return by value + uniform init) behave as 0 copies by writing directly into main's stack? Nice, that means I don't have to worry about expensive copies! Also, how do you know all of this! I couldn't even begin to make a search phrase to hit into Google! From where can I get info like this? Stackoverflow is hostile to beginners like me, so I'm left with not much of resources (apart from asking you) to look up stuff :/ Speaking of, I can't express enough how useful and important the website and your kind, patient, thought out, detailed explanations have been! Absolute Gold, I say!

> From a language point of view, line 4 is an anonymous object. It's there, it doesn't have a name, but it's not really there after compilation.
I was thinking along the same lines. After all, these are, at the end of the day managed by the compiler so that we can code better.

Thank you for the clarification! God this was messing with my head for _so_ long! I'll get back if I have any more questions about the nuances, Thank you again!

nascardriver
October 31, 2019 at 4:07 am · Reply

Thanks for digging up the lessons! What they show isn't related to what's happening here. I just noticed that we're a little too early in the lessons to explain what happens when a value is returned.
When a class-type is returned, a copy has to be made. This copy might involve more than just copying the memory of one object to that of another object, which is what makes the elision of the copy important to the programmer (If it was just a memory copy, the coder wouldn't notice a difference).
You'll learn more about what happens during a return when you get to copy constructors and move semantics. The upcoming chapters will get more difficult, but judging by your interest in the language, I think you're going to make it to the end.

> that means I don't have to worry about expensive copies!
Your compiler usually optimizes returns for you. What happened in C++17 is that compilers have to omit the copy in certain situations.

```
T fn()
{
  return T(123); // No copy allowed
}

// This is probably what you do with big objects
T fn2()
{
  T t(123);

  // ...

  return t; // Can still cause a copy, but probably won't.
}
```

Unless you explicitly tell your compiler to stop omitting copies, it probably will omit the copies caused by `fn2` even when optimizations are disabled.

hellmet
October 31, 2019 at 4:20 am · Reply

> What they show isn't related to what's happening here. I just noticed that we're a little too early in the lessons to explain what happens when a value is returned. When a class-type is returned, a copy has to be made.
Hmm I thought as much. They mostly deal with inbuilt types, so a trivial copy is erm.. graspable. With class objects, not so much.

> You'll learn more about what happens during a return when you get to copy constructors and move semantics.

Okays!

```
T(123); // No copy allowed
```

This has to write directly into the callers stack then, Okay, noted!

> return t; // Can still cause a copy, but probably won't.
Hmm... Is there any way I can check? Of course, premature optimization is evil, but would be nice to see what's happening!

And thank you very much for your kind words!

nascardriver
October 31, 2019 at 4:30 am · Reply

> Is there any way I can check?
That's difficult with what you know so far. You can try godbolt.org . In the assembly view, look for lines like

```
call    Cents::Cents(const Cents&)
```

those are copies being made.
For example here, C++14 with explicitly disabled copy elision https://godbolt.org/z/BhEuxH
Click on line 16 and press ctrl+f10 to reveal the line in assembly (bottom right).
There's one copy, then another one in the initialization of `c` and another one in main. You can also see the 3 copies being made in the top right panel.

but you won't find any when elision is enabled (https://godbolt.org/z/FbZ9S7).

hellmet
October 31, 2019 at 4:47 am · Reply

Oh My God! So Much detail!
It's _SO_ clear! One copy in the copy init, one copy during the return into main's stack, one copy during the uniform initialization of the centsSum! (hope I got that right)
Hmm I also see that these copies always seem to call the function that takes a const ref as a parameter.
Thank you Very Much! Mind Blown!

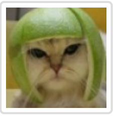
Michael xd
September 17, 2019 at 8:30 am · Reply

Hi Alex.
i still don't understand one thing.Does any anonymous object have a memory adress or not ?
And if it doesn't, how next instruction works.

```
const int& a = 3;
```

I thought const references needs an lvalue to work.

Thank you in advance !



Alex
September 19, 2019 at 1:32 pm · Reply

There are 3 possibilities here:
* The compiler puts it in memory (yes)
* The compiler uses a CPU register (no)
* The compiler optimizes it away completely somehow (no)

In the case of the above, the compiler would create a temporary object with value 3 (in memory) so that a const reference could be set to it.

Nirbhay
August 14, 2019 at 2:07 am · Reply

Hello!

```
Cents add(const Cents &c1, const Cents &c2)
{
    return Cents(c1.getCents() + c2.getCents()); // return anonymous Cents value
}
```

Is 'return Cents(c1.getCents() + c2.getCents());' casting of the result to 'Cents' type?

Thanks.

nascardriver
August 14, 2019 at 2:59 am · Reply

`c1.getCents() + c2.getCents()` returns an `int`. The `Cents(int)` constructor is used to construct a new `Cents`. There's no need to explicitly call the constructor, you can use brace initialization

```
return { c1.getCents() + c2.getCents() };
```

Nirbhay
August 14, 2019 at 3:21 am · Reply

I do not understand the following things:

if we use 'return {c1.getCents() + c2.getCents()};' like you mentioned above then.........

1. it returns 'int'(As anonymous object) but the function add() says that it should return a 'Cents'.

2. how do we call 'getCents()' on this 'int' in this line:

```
std::cout << "I have " << add(Cents(6), Cents(8)).getCents() << " cents." << std::endl; // pr
int anonymous Cents value
```

Does it get called on the anonymous object with int type then?

Thanks.

nascardriver
August 14, 2019 at 3:24 am · Reply

1.

```
return { c1.getCents() + c2.getCents() };
```

Constructs a new `Cents` object, because brace initialization knows the return type.

2.
You're not calling it on an `int`, you're calling it on the `Cents` returned by `add`.

sekhar
May 27, 2019 at 7:14 am · Reply

Hi Alex,

In Summary, the statement "This means anonymous objects can only be passed or returned by value or const reference." w.r.t returned by value looks good but return by const reference is a problem. statement should be modified i believe. Below code causes segmentation file when returning const reference.

#include <iostream>
#include <string>

using namespace std;

class Cents
{
private:
    int m_cents;

public:

    Cents(int val):m_cents(val){   }

    ~Cents() { }

```
    friend const Cents& addCents(const Cents& c1, const Cents& c2);

    int getValue() const
    {
        return m_cents;
    }
};

const Cents& addCents(const Cents& c1, const Cents& c2)
{
    return Cents(c1.getValue() + c2.getValue());
}

int main()
{
    Cents c1(2), c2(3);
    cout << "Total Cents :" << addCents(c1, c2).getValue() << endl;
    return 0;
}
```

Alex
May 30, 2019 at 3:23 pm · Reply

Thanks, fixed.

Paulo Filipe
May 20, 2019 at 2:24 am · Reply

Thanks for this member function Alex

```
int getCents() const { return m_cents; }
```

It made me re-read lesson 8.10 and realize my brain didn't record a damm thing. :)

ConfusedKid
February 20, 2019 at 4:36 am · Reply

Isn't it somewhat confusing?
How does the object "Cents" able to getCents??
I would agree if "Person" is able to getCents and then "Cents" makes it's count less. But it is not alive to give you any "Cents". Jesus

Just an example of how would I see this happening:

```
//! Somewhere in the code
CPerson.getCents();

void CPerson::getCents(int amount_of_desirable_cents)
{
  person->cents.cnt += cents->cnt - amount_of_desirable_cents;
  cents->cnt -= amount_of_desirable_cents;
}
```

I do see some possible behavior that cents can exhibit, ex.: being pressed at some random amount, pushed, affected by some kind of chemical reaction.
But if cents itself doing something with itself, holy crap, correct me if I'm wrong, but how is it possible!!!!!!!!!!!!!
Even we are as a complex object do what we do because of internal mechanisms that push us to eat, drink, talk etc. Or external conditions that impact on us in order to behave somewhat appropriateable.
#EDITION
Or should I think about it as if we say to object like invoke this behavior? object->getMeCents? Wouldn't it be then like Cents->GiveMeCents?

nascardriver
February 20, 2019 at 6:31 am · Reply

Reading this was a wild ride. How about the name "getAmount"?

```
int getAmount() const
{
  return this->m_cents;
}
```

ConfusedKid
February 20, 2019 at 10:53 pm · Reply

Yeap that would be better. As an imperative way of interaction with an object.
But I mostly confused should I use methods (name it) in imperative way, like:

```
Cat->ShutUp();
```

```
Bear->GoFyourSelf();
```

Or in declarative way:

```
Cat->wontTalk();
```

```
Bear->MovesTowardYourHome();
```

nascardriver
February 21, 2019 at 4:43 am · Reply

The first 2 set the state in the animal. The last 2 read the state from the animal an return it.
Of course it's up to you how you name your functions, this is just my interpretation of the names.

ConfusedKid
February 21, 2019 at 6:06 am · Reply

Okay, I meant like if that was the syntax of c++:

```
Cat<-command
Cat->behavior
```

```
Cat<-getLost();      //! imperative
Cat->onAngryOwner(); //! declarative
```

But I get it that both ways are superior. Thanks, mate!

Ryan
February 1, 2019 at 4:44 am · Reply

How could this be improved?

```
#include <iostream>

class Storage
{
private:
    int m_nValue {};
    double m_dValue {};
public:
    Storage(int nValue = 0, double dValue = 0.0) : m_nValue { nValue }, m_dValue { dValue }
    {
    }

    Storage(double dValue = 0.0) : m_dValue { dValue }
    {
    }

    int getInt() const { return m_nValue; }
    double getDouble() const { return m_dValue; }

    friend class Direct;
};

class Vector2d
{
private:
    int m_vector1 {};
    double m_vector2 {};
public:
    Vector2d(int vector1 = 0, double vector2 = 0.0) : m_vector1 { vector1 }, m_vector2 { vector2 }
    {
    }

    Vector2d(double vector2 = 0.0) : m_vector2 { vector2 }
34      {
35      }
36
37      int getVectorInt() const { return m_vector1; }
38      double getVectorDouble() const { return m_vector2; }
39
40      friend class Direct;
41  };
42
43  Storage add(const Storage &s1, const Vector2d &v1)
44  {
45      return Storage(s1.getInt() + v1.getVectorInt(), s1.getDouble() + v1.getVectorDouble());
46  }
47
48  class Direct
49  {
50  private:
51      bool m_isIntValue {};
52  public:
53      Direct(bool isIntValue = true) : m_isIntValue { isIntValue }
54      {
55      }
56
57      void print(const Storage &storage)
58      {
59          if (m_isIntValue)
60              std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
61          else
62              std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
63
64      }
65  };
66
67
68
69  int main()
70  {
71      Direct direct(true);
72      direct.print(add(Storage(1, 6.7), Vector2d(2, 2.2)));
73
74
75      return 0;
76  }
```

nascardriver
February 1, 2019 at 8:22 am · Reply

* Line 45, 71, 72: Initialize your variables with uniform initialization. You used direct initialization.
* Line 9, 29, 45: Limit your lines to 80 characters in length for better readability on small displays.
* Line 6, 7, 26, 27, 51: Initialize to a specific value (0, 0.0, false)
* @Storage and @Vector2d are equivalent. Don't repeat yourself.

Dev
January 3, 2019 at 10:26 am · Reply

Hi Alex!
Could tell me one thing? "Temporary objects" are the same as "Anonymous objects"?

Alex
January 6, 2019 at 1:20 pm · Reply

Yep, I use the two synonymously. Anonymous objects have no name, which means they can't live longer than an expression (unless you set a reference to them), which makes them temporary in nature.

Neither are particularly well defined terms, so others may have definitions that vary slightly.

Jeff
October 25, 2018 at 10:56 pm · Reply

Perhaps worth noting here (and hopefully it's addressed later), that returning a copy of the object created on the stack only "magically works" if the compiler-generated copy constructor replicates the object. At least in my limited C++ experience, that copy constructor is a shallow copy, which will fail to replicate non-trivial object structures.

nascardriver
October 25, 2018 at 11:52 pm · Reply

> that copy constructor is a shallow copy
You're right. Shallow and deep copying is covered in lesson 9.15.

Conor
October 25, 2018 at 12:15 am · Reply

I think this is the right section to ask this. Say I have 3 functions called Foo(), Boo() and Goo(). Boo() and Goo() both return char*'s, initially we have Foo(Boo()) where Boo() calls Goo(). My question is, is there a memory leak here as Foo() is void? Or does the memory be freed up again once the function finishes executing?

nascardriver
October 25, 2018 at 3:29 am · Reply

It depends on the creation of the char* in question. It will only leak memory if the char* was dynamically allocated, otherwise you'll return a temporary or don't have a problem.

Conor
October 25, 2018 at 4:20 am · Reply

Goo() creates an IntPtr object from .NET and ToPointer is called on it which is then casted to a char*. So a char* is passed all the way up so I'm guessing its fine as its not dynamically allocated and IntPtr handles its own de-allocation implicitly.

nascardriver
October 25, 2018 at 4:24 am · Reply

Once the object leaves .NET and is casted there should be no way for the runtime to know when the object is last used. Meaning that it's either leaking, because it's never being freed, or the object is destroyed right away, leaving you with a dangling pointer. You do some reading on this, I have no clue about .NET or integration into C++, this is just what I think happens.
If what I said is true, copy the data as soon as you get it and handle the memory yourself.

Conor
October 25, 2018 at 4:34 am · Reply

Ok I will, thank you!

Michael Stef
September 17, 2018 at 4:56 am · Reply

i don't understand this " Memory allocated dynamically is also done so anonymously (which is why its address must be assigned to a pointer, otherwise we'd have no way to refer to it)." does this mean that anonymous objects is stored in the heap as a dynamic allocated object ? or you mean that just it works the same as dynamic allocation as it does not have a reference we can't refer to it

Alex
September 17, 2018 at 8:53 am · Reply

By anonymously, I mean it's not associated with a variable. If you don't assign the address of the allocated memory to something, it will be orphaned/leaked immediately, as there would be no way to deallocate it.

Michael Stef
September 17, 2018 at 10:59 am · Reply

Oh okay thanks i understood it now ... sorry my bad :D

« Older Comments 1 2