# 8.14 — Anonymous variables and objects

BY ALEX ON DECEMBER 27TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 6TH, 2016

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

```
#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, it's 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 variable. An **anonymous variable** is a variable that is given no name. Anonymous variables in C++ have "expression scope", meaning they are destroyed at the end of the expression in which they are created. Consequently, they must be used immediately!

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

```
#include <iostream>

int add(int x, int y)
{
    return 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, unnamed variable. A copy of the anonymous variable is then returned to the caller by value.

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

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

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

We can write this:

```
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 variable. A copy of this anonymous variable is then passed to the printValue() function, which prints the value 8.

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.

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

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:

```
#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 variables:

```
#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)); // 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:

```
#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:

```
#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; // pri
nt 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:

```
#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 variables are primarily used either to pass or return values without having to create lots of temporary variables to do so. However, it is worth noting that anonymous objects are treated as rvalues (not lvalues, which have an address). This means anonymous objects can only be passed or returned by value or const reference. Otherwise, a named variable must be used instead.

It is also worth noting that because anonymous variables 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.

**9.1 -- Introduction to operator overloading**

**Index**

**8.13 -- Friend functions and classes**

**Share this:**

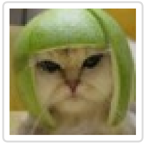
Email | Facebook 3 | Twitter | Google | Pinterest

C++ TUTORIAL | PRINT THIS POST

## 33 comments to 8.14 — Anonymous variables and objects

Zafer
January 29, 2008 at 7:22 pm · Reply

In the last coding example, how were we able to call the Add function directly from the main() function? Why didn't we use Cents.Add instead? Also, how can we define an anonymous variable like Cents(7) for instance? How does the compiler act in terms of returning pointers or references to be able to define an anonymous variable?



**Alex**
January 29, 2008 at 8:50 pm · Reply

In the last example, note that Add() is written as a non-member function. Consequently, it can be called directly. We didn't use Cents.Add() because we didn't write an Add() member function for this version of the Cents class. We certainly could have, but it was easier to show anonymous variables in action this way.

Cents(7) _is_ an anonymous variable because it was never given a variable name. Perhaps I am misunderstanding this part of your question?

Anonymous variables are always dealt with by value [or const reference]. It is not possible to create a pointer or [non-const] reference to an anonymous variable, nor is it possible to use an anonymous variable where a pointer or [non-const] reference parameter or return value is expected.

Josh
July 24, 2009 at 6:21 pm · Reply

"It is not possible to create a pointer or reference to an anonymous variable."

I beg to differ:

```
int main()
{
    int const & a = (2 + 2);
    std::cout << a;
}
<!--formatted-->
```

Alex
March 6, 2016 at 3:35 pm · Reply

I should have said "non-const references". Const references to anonymous objects are allowed, non-const references are not (though Visual Studio will let you do so anyway).

Chris Buck
March 29, 2008 at 11:46 am · Reply

What is a "non-member function" anyway? The line:

```
Cents Add(Cents &c1, Cents &c2)
```

seems pretty confusing to me, as I thought we would have to declare at least the Add() function's prototype somewhere in the class definition and then use Cents::Add() for the actual implementation?
Pls excuse me if I missed it and non-member functions had already been covered.

**Alex**
March 29, 2008 at 1:05 pm · Reply

A non-member function is a function that does not belong to a class. They are the functions that you first learned about before you even knew what a class was!

For example:

```
int Add(int nX, int nY)
{
    return nX + nY;
}
```

This is a simple non-member function. It does not belong to a class. In the same way, the Add() function above is a non-member function, except it adds Cents instead of ints.

We could certainly have written Add() as a normal member function of Cents (in which case, it would have been Cents::Add()), but it would have been somewhat awkward to do so, since we'd have to call it like this:

Cents cResult = cCents1.Add(cCents2);

Probably the best solution would have been to make Add() a static member function of Cents. In that case, we could have called it like this:

Cents cResult = Cents::Add(cCents1, cCents2);

Chris Buck
March 30, 2008 at 12:14 am · Reply

I guessed I was just mixing something up. So 'Cents' was just a simple return value..

Thx for the quick answer and this outstanding tutorial.

**Alex**
March 30, 2008 at 10:29 am · Reply

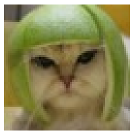
Yes, Cents is a return value.

dano
April 10, 2008 at 7:29 am · Reply

What's really throwin me is line #22. Where I understand the:
Add(cCents1, cCents2) part but the .GetCents is confusing.
Which .GetCents is being called? cCents1.GetCents() or
cCents2.GetCents()?
I'm thinking like: Add(cCents1.GetCents(), cCents2.GetCents())

Thanks for your help.

Dano



**Alex**
April 12, 2008 at 2:16 pm · Reply

The first thing that happens is that Add(cCents1, cCents2) is called. Add() creates a new Cents object, which it returns to the caller. Then, GetCents() is called on the return value of the Add() function.

the joker
July 22, 2009 at 4:44 pm · Reply

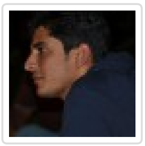
```
Cents Add(Cents &c1, Cents &c2)
```

i can't understand why c1 and c2 are passed by reference ??!!

**E-Barto**
December 6, 2009 at 1:12 am · Reply

If they weren't, you couldn't access them later on in the function:

```
Cents Add(Cents &c1, Cents &c2)
{
    //here
    return Cents(c1.GetCents() + c2.GetCents());
}
```



keshava
November 21, 2009 at 11:26 pm · Reply

nice!!!

CompuChip
May 10, 2010 at 2:15 am · Reply

In fact, I think you could write the main function without any named variables at all:

```
int main()
{
  std::cout << "I have " << Add(Cents(6), Cents(8)).GetCents() << " cents." << std::endl;
  return 0;
}
```

ZNorQ
September 6, 2013 at 3:04 am · Reply

REF: Add(Cents(6), Cents(8)).GetCents()
I also expected this to be true, but it wouldn't run. I fail to see why this wouldn't work. :/

ZNorQ
September 9, 2013 at 2:25 am · Reply

Forget it, I understand why; Add()'s parameters are passed as references, and not by value.
ZNorQ

Bede
February 20, 2016 at 11:34 am · Reply

Yes, that works, but you also have to change the Add() and GetCents() functions so that they work with constants (and const refs):

```
#include <iostream>

class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }
    int GetCents() const { return m_nCents; } // GetCents should work on consts and c
onst refs
};

Cents Add(const Cents &c1, const Cents &c2) // just read-only access to c1 and c2 nec
essary
{
    return Cents(c1.GetCents() + c2.GetCents());
}

int main()
{
    std::cout << "I have " << Add(Cents(6), Cents(8)).GetCents() << " cents." << st
d::endl;
    return 0;
}
```

Just added three times "const".

Chip
June 13, 2010 at 4:16 pm · Reply

Hold your horses. You can write a function of type class? And how this works? :

Cents cCentsSum = Add(cCents1, cCents2);

Shouldn't it be like:

Cents cCentsSum(Add(cCents1, cCents2));

I really want to know how things like this work.

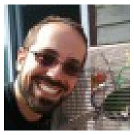
Gammerz
August 4, 2010 at 9:37 am · Reply

```
Cents cCentsSum = Add(cCents1, cCents2);
```

declares an object cCentsSum of type Cents and then initialises it, all on one line. It could have been split into two commands:-

```
Cents cCentsSum;
cCentsSum = Add(cCents1, cCents2);
```

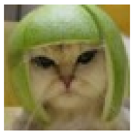
The Add function returns an object of type Cents so this initialisation is valid. Note that the Add function is not a member of the class cents, it's defined outside the class, so is a "normal" function. It's ok to use functions or expressions as initialisation values, we are not restricted to literals.


mslade
December 2, 2010 at 10:24 am · Reply

Great guide. I thought I would add some advice for those new to programming:

If you're using the same expression in anonymous values repeatedly, it should probably be a named variable instead. For example, if you're passing strlen(szName) into the value of 3 different functions, capture it in a named variable and use that three times, instead.

This way, strlen(szName) doesn't have to be calculated three times. It also makes the code more compact and thus more readable.


Alex
March 6, 2016 at 3:41 pm · Reply

Great point. I've updated the article to make this more clear.

Janez
March 14, 2015 at 12:40 pm · Reply

Function Cents Add() was a bit confusing (as you can see from comments). Especially this part: Cents cCentsSum = Add(cCents1, cCents2);
Since you've never mentioned, that this is one of possible ways to call class constructor (as I recall), this confuses a lot of people. Additionally you used "normal" constructor call in Add function (Cents(c1.GetCents() + c2.GetCents())).
Maybe you should mention, that Cents cCentsSum = Add(cCents1, cCents2) is the same as Cents

cCentsSum(Add(cCents1, cCents2)), and Add is non-class function with return type Cents. Cheers!

Methos
July 21, 2015 at 2:48 pm · Reply

So I was writing a program to make examples of this chapter of tutorials in one place and ran into a problem and an annoyance. Both of which I think I've solved, so this is more story than help request, though if there's a more elegant way to go about it, I'd be glad to hear that.

```
DateAndWeather GetToday()
{
               return DateAndWeather(true,
                      GetValue("What month is it?"),
                      GetValue("What day is it?"),
                      GetValue("What year is it?"),
                      GetValue("What was the low temperature on this day?"),
                      GetValue("What was the high temperature on this day?") );
}

void FillArray(DateAndWeather* dPtr, int size)
{
     for (int count=0 ; count < size ; count++)
         dPtr[count] = GetToday();
}
```

GetValue just prints the string and returns the integer response. DateAndWeather is a class of integer values and get/set functions. main() has previously used GetValue to obtain size, and claimed a new DateAndWeather array of that size.

My annoyance is that those questions get asked out of order. It occurs to me that since this is the only way that data comes in in my program, I can just flip the order of the questions and my constructor parameters.

The problem I spent an hour and a half beating my head against my keyboard over is that I included an ID generator in my class, but the IDs were being assigned too high. So for 3 entries, they'd be listed as day 4, 5, and 6. What I ultimately think happened is that the anonymous variable I generated in GetToday went to my constructor, got a new ID, and overwrote the one dPtr[count] got when it was initialized in main. So my new constructor looks like this:

```
DateAndWeather::DateAndWeather (bool garbage, int nMonth , int nDay , int nYear, int nLow ,
int nHigh)
{
     SetDateAndTemps(nMonth, nDay, nYear, nLow, nHigh);
     if (garbage)
     m_nID = s_nIDGenerator++;
}
```

main() doesn't send anything and the prototype in the class declaration sets it to false by default, so the increment only occurs when GetToday's anonymous variable calls it.

I think. It works now, in any case.

**Avneet**
September 4, 2015 at 9:21 pm · Reply

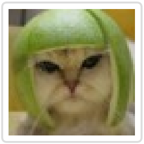
```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    int GetCents() { return m_nCents; }
};

Cents Add(Cents &c1, Cents &c2)
{
    return Cents(c1.GetCents() + c2.GetCents());
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    std::cout << "I have " << Add(cCents1, cCents2).GetCents() << " cents." << std::endl;

    return 0;
}
```

Alex why did you wrote the name Cents when returning from Add(). The program works fine without it. Not a very big question but I would like to know if there is a big reason behind that.

In case you don't understand my question, i am talking about this:

```
return Cents(c1.GetCents() + c2.GetCents()); // okay!
```

```
return (c1.GetCents() + c2.GetCents()); // still okay!
```



Alex

September 5, 2015 at 12:15 am · Reply

Without it, you're relying on the compiler to implicitly convert an int to a Cents.

With it, you're explicitly telling the compiler what kind of object to create and return.

It'll work either way in this case. But in general, leaving something up to the compiler is just one more vector for something to unexpectedly go wrong, so being explicit is usually better.

Mr D

October 9, 2015 at 1:55 pm · Reply

Hi Alex,

The second to last code block example in this lesson has got me really confused!

I don't understand the point of the creation of:

```
Cents Add(Cents &c1, Cents &c2)
```

.

You could have just written it like:

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    int GetCents() { return m_nCents; }
};

//Cents Add(Cents &c1, Cents &c2)
//{
//  Cents cTemp(c1.GetCents() + c2.GetCents());
//  return cTemp;
//}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = (cCents1.GetCents() + cCents2.GetCents());
    std::cout << "I have " << cCentsSum.GetCents() << " cents." << std::endl;

    return 0;
}
```

I believe in this example it's the first time in the tutorial that you've instantiated a class, and then passed another object (2 actually) of the same class to it as the argument. Maybe we need some more explaining about this! If you covered it earlier and i forgot about it, apologies!

Alex
October 9, 2015 at 4:12 pm · Reply

You're right, I _could_ have written it as you note, however the point of the lesson was to show you a simple example of where an anonymous object could be used instead of a temporary variable.

It probably would have made more sense to write function Add() as an overloaded version of operator+. I'll do that when I rewrite the lesson.

If you replace function Add() with the following:

```
Cents operator+(Cents &c1, Cents &c2)
{
    return Cents(c1.GetCents() + c2.GetCents());
}
```

I think you'll agree it's nice to be able to write:

```
Cents cCentsSum = cCents1 + cCents2; // I don't have to know how Cents is implemented
```

Instead of having to use the GetCents() access function to do this:

```
Cents cCentsSum = (cCents1.GetCents() + cCents2.GetCents()); // I have to know that I can construct a Cents from the integer parts of other Cents
```

> I believe in this example it's the first time in the tutorial that you've instantiated a class, and then passed another object (2 actually) of the same class to it as the argument.

No such thing happened! 🙂 The function Add() (or the overloaded operator+ in this comment) is not a member function, so I didn't pass anything to a class I just instantiated.

Function Add() (or the overloaded operator+ in this comment) simply takes two objects of the same class type, and returns an (anonymous) object of that type back to the caller.

Mr D
October 10, 2015 at 2:23 pm · Reply

Thanks Alex,

You said:

[quote]Function Add() (or the overloaded operator+ in this comment) simply takes two objects of the same class type, and returns an (anonymous) object of that type back to the caller.[/quote]

I sort of get that, but then i don't understand how this relates to the code in lesson 9.2:

```
Cents operator+(const Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

Here, the two objects are NOT of the same class type, so why place the word Cents before operator+? I really don't understand what this means (the word "Cents" before operator+)! 🙁

Or is the answer as simple as: whatever the return type will be, the function type must be the same?

Alex
October 10, 2015 at 6:27 pm · Reply

Yes, the type before the function name is the type that the function will return! So in this case:

```
Cents operator+(const Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

the Cents before operator+ indicates the the function will return a Cents object (which it does using the return statement)

This doesn't have anything to do with the function parameters, which could be of any defined type (e.g. you could write an operator+ that took an int and a double and returned a Cents if you wanted -- though it would be kind of silly).

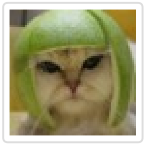
Mr D
October 12, 2015 at 2:58 pm · Reply

Thanks Alex, i'm slowly but slowly getting my head around it!

RS
December 15, 2015 at 12:58 am · Reply

Alex, thank you so very much for this tutorial! After years of fearing pointers, and OOP, suffering from one bad teacher and another, I am able to feel good about these topics.
Any chance we might see a comprehensive quiz for this and the next couple of chapters soon?

My goal is to get the whole tutorial done by the end of the holidays, and move on to other challenges that may help me find a job. Phew!



Alex
December 15, 2015 at 4:45 pm · Reply

Probably not soon. 🙁 I'm adding comprehensive quizzes as I rewrite the lessons (because those lessons add new material, and I want the quizzes to reflect that new material as well as the existing stuff). Chapter 8 is going to take a while to get through, due to the holidays and the sheer amount of material covered in this chapter.

RS
December 17, 2015 at 11:13 am · Reply

I hope you get to them sometime. 🙂 I really appreciate your lessons. In the meanwhile, I'll probably get my partner to assign me quizzes, and keep checking back to any new quizzes that might magically appear 😉