

12.16 — Anonymous objects

ALEX JULY 25, 2021

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

```
1 #include <iostream>
2 int add(int x, int y)
3 {
4     int sum{ x + y };
5     return sum;
6 }
7
8 int main()
9 {
10    std::cout << add(5, 3) <<
11    '\n';
12    return 0;
13 }
```

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:

```
1 #include <iostream>
2 int add(int x, int y)
3 {
4     return x + y; // an anonymous object is created to hold and return the result of x
5 }
6
7 int main()
8 {
9     std::cout << add(5, 3) << '\n';
10    return 0;
11 }
```

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:

```
1 | #include <iostream>
2 | void printValue(int
3 | value)
4 | {
5 |     std::cout << value;
6 | }
7 | int main()
8 | {
9 |     int sum{ 5 + 3 };
10 |    printValue(sum);
11 |    return 0;
12 | }
```

We can write this:

```
1 | #include <iostream>
2 | void printValue(int
3 | value)
4 | {
5 |     std::cout << value;
6 | }
7 | int main()
8 | {
9 |     printValue(5 + 3);
10 |    return 0;
11 | }
```

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.

```
1 | 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:

```

1  #include <iostream>
2
3  class Cents
4  {
5  private:
6      int m_cents{};
7
8  public:
9      Cents(int cents) { m_cents = cents; }
10
11     int getCents() const { return m_cents; }
12 };
13
14 void print(const Cents& cents)
15 {
16     std::cout << cents.getCents() << "
cents\n";
17 }
18
19 int main()
20 {
21     Cents cents{ 6 };
22     print(cents);
23
24     return 0;
25 }

```

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:

```

1  #include <iostream>
2
3  class Cents
4  {
5  private:
6      int m_cents{};
7
8  public:
9      Cents(int cents) { m_cents = cents; }
10
11     int getCents() const { return m_cents; }
12 };
13
14 void print(const Cents& cents)
15 {
16     std::cout << cents.getCents() << " cents\n";
17 }
18
19 int main()
20 {
21     print(Cents{ 6 }); // Note: Now we're passing an anonymous Cents
value
22
23     return 0;
24 }

```

As you'd expect, this prints:

```
6 cents
```

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

```

1  #include <iostream>
2
3  class Cents
4  {
5  private:
6      int m_cents{};
7
8  public:
9      Cents(int cents) { m_cents = cents; }
10
11     int getCents() const { return m_cents; }
12 };
13
14 Cents add(const Cents& c1, const Cents& c2)
15 {
16     Cents sum{ c1.getCents() + c2.getCents() };
17     return sum;
18 }
19
20 int main()
21 {
22     Cents cents1{ 6 };
23     Cents cents2{ 8 };
24     Cents sum{ add(cents1, cents2) };
25     std::cout << "I have " << sum.getCents() << "
26 cents.\n";
27
28     return 0;
29 }

```

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:

```

1  #include <iostream>
2
3  class Cents
4  {
5  private:
6      int m_cents{};
7
8  public:
9      Cents(int cents) { m_cents = cents; }
10
11     int getCents() const { return m_cents; }
12 };
13
14 Cents add(const Cents& c1, const Cents& c2)
15 {
16     // List initialization looks at the return type of the function
17     // and creates the correct object accordingly.
18     return { c1.getCents() + c2.getCents() }; // return anonymous Cents value
19 }
20
21 int main()
22 {
23     Cents cents1{ 6 };
24     Cents cents2{ 8 };
25     std::cout << "I have " << add(cents1, cents2).getCents() << " cents.\n"; // print anonymous Cents
26     value
27
28     return 0;
29 }

```

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:

```

1  #include <iostream>
2
3  class Cents
4  {
5  private:
6      int m_cents{};
7
8  public:
9      Cents(int cents) { m_cents = cents; }
10
11     int getCents() const { return m_cents; }
12 };
13
14 Cents add(const Cents& c1, const Cents& c2)
15 {
16     return { c1.getCents() + c2.getCents() }; // return anonymous Cents value
17 }
18
19 int main()
20 {
21     std::cout << "I have " << add(Cents{ 6 }, Cents{ 8 }).getCents() << " cents.\n"; // print anonymous
22     Cents value
23
24     return 0;
25 }

```

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.



Next lesson

12.17 Nested types in classes



Back to table of contents



Previous lesson

12.15 Friend functions and classes

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code here````



Name*



Email*



Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies:



POST COMMENT

DP N N FOUT

Newest ▼

©2021 Learn C++

