8.14 — Anonymous objects

BY ALEX ON DECEMBER 27TH, 2007 | LAST MODIFIED BY ALEX ON SEPTEMBER 13TH, 2017

In certain 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 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:

```
#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:

```
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 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.

```
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:

```
#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:

```
#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:

```
#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; // print anonym
ous 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 an
onymous 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). 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 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