14.13 — Temporary class objects

Consider the following example:

```
#include <iostream>

int add(int x, int y)
{
    int sum{ x + y }; // stores x + y in a variable
    return sum;       // returns value of that variable
}

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

    return 0;
}
```

In the `add()` function, the variable `sum` is used to store the result of the expression `x + y`. This variable is then evaluated in the return statement to produce the value to be returned. While this might be occasionally useful for debugging (so we can inspect the value of `sum` if desired), it actually makes the function more complex than it needs to be by defining an object that is then only used one time.

In most cases where a variable is used only once, we actually don't need a variable. Instead, we can substitute in the expression used to initialize the variable where the variable would have been used. Here is the `add()` function rewritten in this manner:

```
#include <iostream>

int add(int x, int y)
{
    return x + y; // just return x + y directly
}

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

    return 0;
}
```

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

```
#include <iostream>

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

int main()
{
    int sum{ 5 + 3 };
    printValue(sum);

    return 0;
}
```

We can write this:

```
#include <iostream>

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

int main()
{
    printValue(5 + 3);

    return 0;
}
```

Note how much cleaner this keeps our code. We don't have to define and give a name to a variable. And we don't have to scan through the entire function to determine whether that variable is actually used elsewhere. Because `5 + 3` is an expression, we know it is only used on that one line.

Do note that this only works in cases where an rvalue expression is accepted. In cases where an lvalue expression is required, we must have an object:

```
#include <iostream>

void addOne(int& value) // pass by non-const references requires lvalue
{
    ++value;
}

int main()
{
    int sum { 5 + 3 };
    addOne(sum);   // okay, sum is an lvalue

    addOne(5 + 3); // compile error: not an lvalue

    return 0;
}
```

Temporary class objects

The same issue applies in the context of class types.

Author's note

We'll use a class here, but everything in this lesson is equally applicable to structs that are initialized using aggregate initialization.

The following example is similar to the ones above, but uses program-defined class type `IntPair` instead of `int`:

```
#include <iostream>

class IntPair
{
private:
    int m_x{};
    int m_y{};

public:
    IntPair(int x, int y)
        : m_x { x }, m_y { y }
    {}

    int x() const { return m_x; }
    int y() const { return m_y; }
};

void print(IntPair p)
{
    std::cout << "(" << p.x() << ", " << p.y() << ")\n";
}

int main()
{
    // Case 1: Pass variable
    IntPair p { 3, 4 };
    print(p); // prints (3, 4)

    return 0;
}
```

In case 1, we're instantiating variable `IntPair p` and then passing `p` to function `print()`.

However, `p` is only used once, and function `print()` will accept rvalues, so there is really no reason to define a variable here. So let's get rid of `p`.

We can do that by passing a temporary object instead of a named variable. A **temporary object** (sometimes called an **anonymous object** or an **unnamed object**) is an object that has no name and exists only for the duration of a single expression.

There are two common ways to create temporary class type objects:

```
#include <iostream>

class IntPair
{
private:
    int m_x{};
    int m_y{};

public:
    IntPair(int x, int y)
        : m_x { x }, m_y { y }
    {}

    int x() const { return m_x; }
    int y() const{ return m_y; }
};

void print(IntPair p)
{
    std::cout << "(" << p.x() << ", " << p.y() << ")\n";
}

int main()
{
    // Case 1: Pass variable
    IntPair p { 3, 4 };
    print(p);

    // Case 2: Construct temporary IntPair and pass to function
    print(IntPair { 5, 6 } );

    // Case 3: Implicitly convert { 7, 8 } to a temporary Intpair and pass to
function
    print( { 7, 8 } );

    return 0;
}
```

In case 2, we're telling the compiler to construct an `IntPair` object, and initializing it with `{ 5, 6 }`. Because this object has no name, it is a temporary. The temporary object is then passed to parameter `p` of function `print()`. When the function call returns, the temporary object is destroyed.

In case 3, we're also creating a temporary `IntPair` object to pass to function `print()`. However, because we have not explicitly specified what type to construct, the compiler will deduce the necessary type (`IntPair`) from the function parameter, and then implicitly convert `{ 7, 8 }` to an `IntPair` object.

To summarize:

```
IntPair p { 1, 2 }; // create named object p initialized with value { 1, 2 }
IntPair { 1, 2 };   // create temporary object initialized with value { 1, 2 }
{ 1, 2 };           // compiler will try to convert value { 1, 2 } to temporary
object
```

We'll discuss this last case in more detail in lesson 14.16 -- Converting constructors and the explicit keyword.

It is possibly even more common to see temporary objects used with return values:

```
#include <iostream>

class IntPair
{
private:
    int m_x{};
    int m_y{};

public:
    IntPair(int x, int y)
        : m_x { x }, m_y { y }
    {}

    int x() const { return m_x; }
    int y() const { return m_y; }
};

void print(IntPair p)
{
    std::cout << "(" << p.x() << ", " << p.y() << ")\n";
}

// Case 1: Create named variable and return
IntPair ret1()
{
    IntPair p { 3, 4 };
    return p;
}

// Case 2: Create temporary IntPair and return
IntPair ret2()
{
    return IntPair { 5, 6 };
}

// Case 3: implicitly convert { 7, 8 } to IntPair and return
IntPair ret3()
{
    return { 7, 8 };
}

int main()
{
    print(ret1());
    print(ret2());
    print(ret3());

    return 0;
}
```

The cases in this example are analogous to the cases in the prior example.

A few notes

First, just as in the case of an `int`, when used in an expression, a temporary class object is an rvalue. Thus, such objects can only be used where rvalue expressions are accepted.

Second, temporary objects are created at the point of definition, and destroyed at the end of the full expression in which they are defined. A full expression is an expression that is not a subexpression.