14.13 — Temporary class objects

ALEX[1] DECEMBER 20, 2024

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 that uses list initialization 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:

```cpp
#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:

```cpp
IntPair p { 1, 2 }; // create named object p initialized with { 1, 2 }
IntPair { 1, 2 };   // create temporary object initialized with { 1, 2 }
{ 1, 2 };           // compiler will try to convert { 1, 2 } to temporary object matching expected type (typically a parameter or return type)
```

We'll discuss this last case in more detail in lesson [14.16 -- Converting constructors and the explicit keyword (https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/)](https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/)[2].

A few more examples:

```cpp
std::string { "Hello" }; // create a temporary std::string initialized with "Hello"
std::string {};          // create a temporary std::string using value initialization / default constructor
```

Creating temporary objects via direct initialization Optional

Since we can create temporary objects via direct-list-initialization, you might be wondering whether you can create temporary objects via the other initialization forms. There is no syntax to create temporary objects using copy initialization.

However, you can create temporary objects using direct initialization. For example:

```
Foo (1, 2); //  temporary Foo, direct-initialized with (1, 2) (similar to `Foo { 1, 2 }`)
```

Putting aside the fact that it looks like a function call at first glance, this produces the same result as `Foo { 1, 2 }` (just with no narrowing conversion prevention). Pretty normal right?

We'll now spend the remainder of this section showing you why you probably shouldn't do this.

Author's note

This is mostly here for your reading pleasure, not as something you need to digest, memorize, and be able to explain.

Even if you don't have that much fun reading it, it might help you understand why list initialization is preferred in modern C++!

Now let's look at the case where we don't have any arguments:

```
Foo();     // temporary Foo, value-initialized (identical to `Foo {}`)
```

You probably didn't expect that `Foo()` would create a value-initialized temporary just like `Foo {}` does. And that's probably because this syntax has a completely different meaning when used with a named variable!

```
Foo bar{}; // definition of variable bar, value-initialized
Foo bar(); // declaration of function bar that has no parameters and returns a Foo (inconsistent with `Foo bar{}` and `Foo()`)
```

Ready to get real weird?!?

```
Foo(1);    // Function-style cast of literal 1, returns temporary Foo (similar to `Foo { 1 }`)
Foo(bar);  // Defines variable bar of type Foo (inconsistent with `Foo { bar }` and `Foo(1)`)
```

Wait, what?

- The version with literal `1` in parentheses behaves consistently with all the other versions of this syntax that create temporary objects.
- The version with identifier `bar` in parentheses defines a variable named `bar` (identical to `Foo bar;`). If `bar` is already defined, this will cause a redefinition compile-error.

The compiler knows that literals can't be used as identifiers for variables, so it's able to treat that case consistently with the others.

As an aside...

If you're wondering why `Foo(bar);` behaves identically to `Foo bar;`...

One of the most common uses of parentheses is to group things. For example, in mathematics: `(1 + 2) * 3` produces the result `9`, which is different than `1 + 2 * 3`, which produces the result `7`. If we can do `(1 + 2) * 3`, there's no reason we shouldn't be able to do `(3) * 3`.

For similar reasons, the declaration syntax allows parenthesis-based grouping, and those groups can have a single thing in them. `Foo(bar)` is interpreted as a variable definition consisting of type `Foo` followed by a group that consists only of identifier `bar`. It just looks funny to us, mainly because the parentheses don't serve any useful purpose in this case. But there's no compelling reason to disallow doing so (as that would just make the syntax of the language that much more complicated).

For advanced readers

Let's look at a slightly more complicated case. Consider the statement `Foo * bar();`. By using (or not using) parentheses, we can completely change the meaning of this statement:

- `Foo * bar();` (with no additional parenthesis) groups the `*` with `Foo` by default. `Foo* bar();` is the declaration of a function named `bar` that has no parameters and returns a `Foo*`.
- `Foo (*bar)();` explicitly groups the `*` with `bar`. This defines a function pointer named `bar` that holds the address of a function that takes no parameters and returns a `Foo`.
- `Foo (* bar());` is the same as `Foo * bar();` -- the parentheses are superfluous in this case.

Finally:

- `(Foo *) bar();`. You might expect this to be the same as `Foo* bar()`, but this is actually an expression statement that calls function `bar()`, C-style casts the return value to type `Foo*`, and then discards it!

C++ is so weird sometimes.

Key insight

Parentheses are complex because they are so overloaded, and used in the syntax of vastly different things. This includes function calls, direct-initialization of objects, value initialization of temporaries, C-style casts, groupings of symbols/identifiers, and variable definitions. So when you see parentheses in some syntax... it's not always obvious what you're going to get!

On the other hand, if we see curly braces, we know we're dealing with objects.

Okay, fun over. Back to the boring stuff.

Temporary objects and return by value

When a function returns by value, the object that is returned is a temporary object (initialized using the value or object identified in the return statement).

Here are some examples:

```
#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; // returns temporary object (initialized using p)
}

// Case 2: Create temporary IntPair and return
IntPair ret2()
{
    return IntPair { 5, 6 }; // returns temporary object (initialized using another
temporary object)
}

// Case 3: implicitly convert { 7, 8 } to IntPair and return
IntPair ret3()
{
    return { 7, 8 }; // returns temporary object (initialized using another temporary
object)
}

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

    return 0;
}
```

In case 1, when we `return p`, a temporary object is created and initialized using `p`.

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.

`static_cast` vs explicit instantiation of a temporary object

In cases where we need to convert a value from one type to another but narrowing conversions aren't involved, we often have the option to use either `static_cast` or explicit instantiation of a temporary object.

For example:

```
#include <iostream>

int main()
{
    char c { 'a' };

    std::cout << static_cast<int>( c ) << '\n'; // static_cast returns a temporary int direct-initialized with value of c
    std::cout << int { c } << '\n';             // explicitly creates a temporary int list-initialized with value c

    return 0;
}
```

`static_cast<int>(c)` returns a temporary `int` that is direct-initialized with the value of `c`. `int { c }` creates a temporary `int` that is list-initialized with the value of `c`. Either way, we get a temporary int initialized with the value of `c`, which is what we want.

Let's show a slightly more complex example:

printString.h:

```
#include <string>
void printString(const std::string &s)
{
    std::cout << s << '\n';
}
```

main.cpp:

```cpp
#include "printString.h"
#include <iostream>
#include <string>
#include <string_view>

int main()
{
    std::string_view sv { "Hello" };

    // We want to print sv using the printString() function

//    printString(sv); // compile error: a std::string_view won't implicitly convert to a std::string

    printString( static_cast<std::string>(sv) ); // Case 1: static_cast returns a temporary std::string direct-initialized with sv
    printString( std::string { sv } );           // Case 2: explicitly creates a temporary std::string list-initialized with sv
    printString( std::string ( sv ) );           // Case 3: C-style cast returns temporary std::string direct-initialized with sv (avoid this one!)

    return 0;
}
```

Let's say the code in header file `printString.h` isn't code we can modify (e.g. because it's distributed with some 3rd party library we're using, and has been written to be compatible with C++11, which doesn't support `std::string_view`). So how do we call `printString()` with `sv`? Because a `std::string_view` won't implicitly convert to a `std::string` (for efficiency reasons), we can't just use `sv` as an argument. We must use some explicit form of conversion.

In case 1, `static_cast<std::string>(sv)` invokes the static_cast operator to cast `sv` to a `std::string`. This returns a temporary `std::string` that has been direct-initialized using `sv`, which is then used as the argument for the function call.

In case 2, `std::string { sv }` creates a temporary `std::string` that is list-initialized using `sv`. Since this is an explicit construction, the conversion is allowed. This temporary is then used as the argument for the function call.

In case 3, `std::string ( sv )` use a C-style cast to cast `sv` to a `std::string`. Although this works here, C-style casting can be dangerous in general and should be avoided. Notice how similar this looks to the prior case!

Best practice

As a quick rule of thumb: Prefer `static_cast` when converting to a fundamental type, and a list-initialized temporary when converting to a class type.

Prefer `static_cast` when to create a temporary object when any of the following are true:

- We need to performing a narrowing conversion.
- We want to make it really obvious that we're converting to a type that will result in some different behavior (e.g. a `char` to an `int`).
- We want to use direct-initialization for some reason (e.g. to avoid list constructors taking precedence).

Prefer creating a new object (using list initialization) to create a temporary object when any of the following are true:

- We want to use list-initialization (e.g. for the protection against narrowing conversions, or because we need to invoke a list constructor).
- We need to provide additional arguments to a constructor to facilitate the conversion.

Related content

We cover list constructors in lesson [16.2 -- Introduction to std::vector and list constructors (https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/)][3].

Next lesson
14.14 Introduction to the copy constructor

4

Back to table of contents

5

Previous lesson
14.12 Delegating constructors

6

7

Robert
June 3, 2025 11:56 am PDT

Hi Alex, `printString( std::string ( sv ) ); // Case 3: C-style cast returns temporary std::string direct-initialized with sv (avoid this one!)`
This line why the compiler is not failing here ? Above I saw: Foo(bar); // Defines variable bar of type Foo (inconsistent with `Foo { bar }` and `Foo(1)`) . The c-style cast shouldnt be `(std::string) sv` ?

0 Reply

Chad V
May 16, 2025 6:32 am PDT

The best practice at the bottom is worded
"Prefer static_cast **when to create** a temporary object when any of the following are true:"

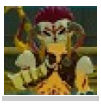
It likely is intended as
"Prefer static_cast **to create** a temporary object when any of the following are true:"

0 Reply

beretkaaaa
May 6, 2025 2:40 am PDT

English isn't my first language so I might be wrong, sorry in advance.
In the last Best practice block there is a sentence "We need to performing a narrowing conversion.", wouldn't "We need to perform" be correct here?

0 Reply



Sebby Teh
March 28, 2025 11:08 am PDT

Thank you for using curly braces for initializing objects, it makes it so much clearer and safer, it also avoids confusion with functions.

0 Reply

hatola
March 25, 2025 3:14 pm PDT

one of the best and most interesting lessons so far!

0 Reply

Jack

March 6, 2025 1:40 pm PST

why exactly are we passing in the address of the c-style string in the header file?

0 Reply

Bassem

February 27, 2025 10:57 am PST

Typo
We need to performing a narrowing conversion.
to performing -> to perform

0 Reply

jorge98

February 8, 2025 5:33 pm PST

after 1 week of procrastinating i finally finished this lesson :)

1 Reply

EmtyC

December 21, 2024 4:26 am PST

I wasn't ready for the () things :>
Sir Alex, how about adding a section or lesson in the chapter on functions (20) about parenthesis in c++.
Keep up the great work :)

1 Reply

chemumu

November 21, 2024 9:07 am PST

How come `std::string ( sv )` is a C-style cast and not a call to the constructor of std::string? I thought a C-style cast is something like `(std::string) sv`.

If it is a C-style cast, can one tell if `std::string ( x )` is a C-style cast or a call to a constructor without knowing exactly what x is?

0 Reply

Alex Author

Reply to chemumu [11] November 25, 2024 3:55 pm PST

Per https://en.cppreference.com/w/cpp/language/direct_initialization, `std::string ( sv )` is a function-style cast, which is a variant of the C-style cast. Per

https://en.cppreference.com/w/cpp/language/explicit_cast, this performs a static_cast. And per https://en.cppreference.com/w/cpp/language/static_cast, the result object is direct-initialized (using the constructor).

So essentially, for class types, it is both -- a C-style cast that calls the constructor using direct initialization. The end result is effectively identical to the case where we're direct initializing a variable, just a little more roundabout way of getting there.

1 Reply

chemumu

Reply to Alex [12] November 27, 2024 8:38 am PST

I see, thanks!

Are these examples function-style casts too?

```
1  class Foo {
2  public:
3      Foo(int value) : m_value{value} {}
4
5  private:
6      const int m_value{};
7  };
8
9  int main() {
10     const Foo foo_1(1);       // function-style cast?
11     const Foo foo_2 = Foo(2); // function-style cast?
12
13     return 0;
14 }
```

I can now see why this should be avoided - nothing prevents me from defining `foo_1` as `const Foo foo_1(1.5);`, but I guess it's unavoidable in some situations, e.g., when trying to define a vector of n elements with `const std::vector<int> v(10);`

0 Reply

Alex Author

Reply to chemumu [13] November 29, 2024 11:32 pm PST

Named variables with a parenthesis initializer use direct initialization.

And yes, direct initialization doesn't prevent narrowing conversions, but your compiler will probably warn you.

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/delegating-constructors/
7. https://www.learncpp.com/temporary-class-objects/
8. https://www.learncpp.com/site-news/happy-holidays-to-everyone/
9. https://www.learncpp.com/cpp-tutorial/basic-inheritance-in-c/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/temporary-class-objects/#comment-604381
12. https://www.learncpp.com/cpp-tutorial/temporary-class-objects/#comment-604522
13. https://www.learncpp.com/cpp-tutorial/temporary-class-objects/#comment-604580
14. https://g.ezoic.net/privacy/learncpp.com