

14.16 — Converting constructors and the explicit keyword

👤 **ALEX**¹ ⌚ **FEBRUARY 1, 2025**

In lesson [10.1 -- Implicit type conversion](https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/)², we introduced type conversion and the concept of implicit type conversion, where the compiler will implicitly convert a value of one type to a value of another type as needed if such a conversion exists.

This allows us to do things like this:

```
1  #include <iostream>
2
3  void printDouble(double d) // has a double parameter
4  {
5      std::cout << d;
6  }
7
8  int main()
9  {
10     printDouble(5); // we're supplying an int argument
11
12     return 0;
13 }
```

In the above example, our `printDouble` function has a `double` parameter, but we're passing in an argument of type `int`. Because the type of the parameter and the type of the argument do not match, the compiler will see if it can implicitly convert the type of the argument to the type of the parameter. In this case, using the numeric conversion rules, `int` value `5` will be converted to `double` value `5.0` and because we're passing by value, parameter `d` will be copy initialized with this value.

User-defined conversions

Now consider the following similar example:

```

1  #include <iostream>
2
3  class Foo
4  {
5  private:
6      int m_x{};
7  public:
8      Foo(int x)
9          : m_x{ x }
10     {
11     }
12
13     int getX() const { return m_x; }
14 };
15
16 void printFoo(Foo f) // has a Foo parameter
17 {
18     std::cout << f.getX();
19 }
20
21 int main()
22 {
23     printFoo(5); // we're supplying an int argument
24
25     return 0;
26 }

```

In this version, `printFoo` has a `Foo` parameter but we're passing in an argument of type `int`. Because these types do not match, the compiler will try to implicitly convert `int` value `5` into a `Foo` object so the function can be called.

Unlike the first example, where our parameter and argument types were both fundamental types (and thus can be converted using the built-in numeric promotion/conversion rules), in this case, one of our types is a program-defined type. The C++ standard doesn't have specific rules that tell the compiler how to convert values to (or from) a program-defined type.

Instead, the compiler will look to see if we have defined some function that it can use to perform such a conversion. Such a function is called a **user-defined conversion**.

Converting constructors

In the above example, the compiler will find a function that lets it convert `int` value `5` into a `Foo` object. That function is the `Foo(int)` constructor.

Up to this point, we've typically used constructors to explicitly construct objects:

```

1 | Foo x { 5 }; // Explicitly convert int value 5 to a Foo

```

Think about what this does: we're providing an `int` value (`5`) and getting a `Foo` object in return.

In the context of a function call, we're trying to solve the same problem:

```

1 | printFoo(5); // Implicitly convert int value 5 into a Foo

```

We're providing an `int` value (`5`), and we want a `Foo` object in return. The `Foo(int)` constructor was designed for exactly that!

So in this case, when `printFoo(5)` is called, parameter `f` is copy initialized using the `Foo(int)` constructor with `5` as an argument!

As an aside...

Prior to C++17, when `printFoo(5)` is called, `5` is implicitly converted to a temporary `Foo` using the `Foo(int)` constructor. This temporary `Foo` is then copy constructed into parameter `f`.

In C++17 onward, the copy is mandatorily elided. Parameter `f` is copy initialized with value `5`, and no call to the copy constructor is required (and it will work even if the copy constructor is deleted).

A constructor that can be used to perform an implicit conversion is called a **converting constructor**. By default, all constructors are converting constructors.

Only one user-defined conversion may be applied

Now consider the following example:

```
1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  class Employee
6  {
7  private:
8      std::string m_name{};
9
10 public:
11     Employee(std::string_view name)
12         : m_name{ name }
13     {
14     }
15
16     const std::string& getName() const { return m_name; }
17 };
18
19 void printEmployee(Employee e) // has an Employee parameter
20 {
21     std::cout << e.getName();
22 }
23
24 int main()
25 {
26     printEmployee("Joe"); // we're supplying an string literal argument
27
28     return 0;
29 }
```

In this version, we've swapped out our `Foo` class for an `Employee` class. `printEmployee` has an `Employee` parameter, and we're passing in a C-style string literal. And we have a converting constructor: `Employee(std::string_view)`.

You might be surprised to find that this version doesn't compile. The reason is simple: only one user-defined conversion may be applied to perform an implicit conversion, and this example requires two. First, our C-style string literal has to be converted to a `std::string_view` (using a `std::string_view` converting constructor), and then our `std::string_view` has to be converted into an `Employee` (using the `Employee(std::string_view)` converting constructor).

There are two ways to make this example work:

1. Use a `std::string_view` literal:

```
1 | int main()
2 | {
3 |     using namespace std::literals;
4 |     printEmployee( "Joe"sv); // now a std::string_view literal
5 |
6 |     return 0;
7 | }
```

This works because only one user-defined conversion is now required (from `std::string_view` to `Employee`).

2. Explicitly construct an `Employee` rather than implicitly create one:

```
1 | int main()
2 | {
3 |     printEmployee(Employee{ "Joe" });
4 |
5 |     return 0;
6 | }
```

This also works because only one user-defined conversion is now required (from the string literal to the `std::string_view` used to initialize the `Employee` object). Passing our explicitly constructed `Employee` object to the function does not require a second conversion to take place.

This latter example brings up a useful technique: it is trivial to convert an implicit conversion into an explicit definition. We'll see more examples of this later in this lesson.

Key insight

An implicit conversion can be trivially converted into an explicit definition by using direct list initialization (or direct initialization).

When converting constructors go wrong

Consider the following program:

```

1  #include <iostream>
2
3  class Dollars
4  {
5  private:
6      int m_dollars{};
7
8  public:
9      Dollars(int d)
10         : m_dollars{ d }
11     {
12     }
13
14     int getDollars() const { return m_dollars; }
15 };
16
17 void print(Dollars d)
18 {
19     std::cout << "$" << d.getDollars();
20 }
21
22 int main()
23 {
24     print(5);
25
26     return 0;
27 }

```

When we call `print(5)`, the `Dollars(int)` converting constructor will be used to convert `5` into a `Dollars` object. Thus, this program prints:

\$5

Although this may have been the caller's intent, it's hard to tell because the caller did not provide any indication that this is what they actually wanted. It's entirely possible that the caller assumed this would print `5`, and did not expect the compiler to silently and implicitly convert our `int` value to a `Dollars` object so that it could satisfy this function call.

While this example is trivial, in a larger and more complex program, it's fairly easy to be surprised by the compiler performing some implicit conversion that you did not expect, resulting in unexpected behavior at runtime.

It would be better if our `print(Dollars)` function could only be called with a `Dollars` object, not any value that can be implicitly converted to a `Dollars` (especially a fundamental type like `int`). This would reduce the possibility of inadvertent errors.

The explicit keyword

To address such issues, we can use the **explicit** keyword to tell the compiler that a constructor should not be used as a converting constructor.

Making a constructor **explicit** has two notable consequences:

- An explicit constructor cannot be used to do copy initialization or copy list initialization.
- An explicit constructor cannot be used to do implicit conversions (since this uses copy initialization or copy list initialization).

Let's update the `Dollars(int)` constructor from the prior example to be an explicit constructor:

```

1  #include <iostream>
2
3  class Dollars
4  {
5  private:
6      int m_dollars{};
7
8  public:
9      explicit Dollars(int d) // now explicit
10         : m_dollars{ d }
11     {
12     }
13
14     int getDollars() const { return m_dollars; }
15 };
16
17 void print(Dollars d)
18 {
19     std::cout << "$" << d.getDollars();
20 }
21
22 int main()
23 {
24     print(5); // compilation error because Dollars(int) is explicit
25
26     return 0;
27 }

```

Because the compiler can no longer use `Dollars(int)` as a converting constructor, it can not find a way to convert `5` to a `Dollars`. Consequently, it will generate a compilation error.

For constructors with a separate declaration (inside the class) and definition (outside the class), the `explicit` keyword is used only on the declaration.

Explicit constructors can be used for direct and direct list initialization

An explicit constructor can still be used for direct and direct list initialization:

```

1  // Assume Dollars(int) is explicit
2  int main()
3  {
4      Dollars d1(5); // ok
5      Dollars d2{5}; // ok
6  }

```

Now, let's go back to our prior example, where we made our `Dollars(int)` constructor explicit, and therefore the following generated a compilation error:

```

1  print(5); // compilation error because Dollars(int) is explicit

```

What if we actually want to call `print()` with `int` value `5` but the constructor is explicit? The workaround is simple: instead of having the compiler implicitly convert `5` into a `Dollars` that can be passed to `print()`, we can explicitly define the `Dollars` object ourselves:

```

1  print(Dollars{5}); // ok: explicitly create a Dollars

```

This is allowed because we can still use explicit constructors to list initialize objects. And since we've now explicitly constructed a `Dollars`, the argument type matches the parameter type, so no conversion is required!

This not only compiles and runs, it also better documents our intent, as it is explicit about the fact that we meant to call this function with a `Dollars` object.

Note that `static_cast` returns an object that is direct-initialized, so it will consider explicit constructors while performing the conversion:

```
1 | print(static_cast<Dollars>(5)); // ok: static_cast will use explicit constructors
```

Return by value and explicit constructors

When we return a value from a function, if that value does not match the return type of the function, an implicit conversion will occur. Just like with pass by value, such conversions cannot use explicit constructors.

The following programs shows a few variations in return values, and their results:

```
1 | #include <iostream>
2 |
3 | class Foo
4 | {
5 | public:
6 |     explicit Foo() // note: explicit (just for sake of example)
7 |     {
8 |     }
9 |
10 |    explicit Foo(int x) // note: explicit
11 |    {
12 |    }
13 | };
14 |
15 | Foo getFoo()
16 | {
17 |     // explicit Foo() cases
18 |     return Foo{ }; // ok
19 |     return { }; // error: can't implicitly convert initializer list to Foo
20 |
21 |     // explicit Foo(int) cases
22 |     return 5; // error: can't implicitly convert int to Foo
23 |     return Foo{ 5 }; // ok
24 |     return { 5 }; // error: can't implicitly convert initializer list to Foo
25 | }
26 |
27 | int main()
28 | {
29 |     return 0;
30 | }
```

Perhaps surprisingly, `return { 5 }` is considered a conversion.

Best practices for use of `explicit`

The modern best practice is to make any constructor that will accept a single argument `explicit` by default. This includes constructors with multiple parameters where most or all of them have default values. This will disallow the compiler from using that constructor for implicit conversions. If an implicit conversion

is required, only non-explicit constructors will be considered. If no non-explicit constructor can be found to perform the conversion, the compiler will error.

If such a conversion is actually desired in a particular case, it is trivial to convert the implicit conversion into an explicit definition using direct list initialization.

The following **should not** be made explicit:

- Copy (and move) constructors (as these do not perform conversions).

The following **are typically not** made explicit:

- Default constructors with no parameters (as these are only used to convert `{}` to a default object, not something we typically need to restrict).
- Constructors that only accept multiple arguments (as these are typically not a candidate for conversions anyway).

However, if you prefer, the above can be marked as explicit to prevent implicit conversions with empty and multiple-argument lists.

The following **should usually** be made explicit:

- Constructors that take a single argument.

There are some occasions when it does make sense to make a single-argument constructor non-explicit. This can be useful when all of the following are true:

- The constructed object is semantically equivalent to the argument value.
- The conversion is performant.

For example, the `std::string_view` constructor that accepts a C-style string argument is not explicit, because there is unlikely to be a case when we wouldn't be okay with a C-style string being treated as a `std::string_view` instead. Conversely, the `std::string` constructor that takes a `std::string_view` is marked as explicit, because while a `std::string` value is semantically equivalent to a `std::string_view` value, constructing a `std::string` is not performant.

Best practice

Make any constructor that accepts a single argument **explicit** by default. If an implicit conversion between types is both semantically equivalent and performant, you can consider making the constructor non-explicit.

Do not make copy or move constructors explicit, as these do not perform conversions.



Next lesson

14.17 [Constexpr aggregates and classes](#)

3



[Back to table of contents](#)

4



Previous lesson

14.15 [Class initialization and copy elision](#)

5

6



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...



Name*



Email*



Notify me about replies:



POST COMMENT

🚩 Find a mistake? Leave a comment above!?

👤 Avatars from <https://gravatar.com/>⁸ are connected to your provided email address.

221 COMMENTS

Newest ▼



Badger Patcher

🕒 March 30, 2025 2:22 pm PDT

In the subtopic "Converting constructors" you said: "Think about what this does: we're providing an int value (5) and getting a Foo object in return." and "We're providing an int value (5), and we want a Foo object in return. The Foo(int) constructor was designed for exactly that!" but constructors don't return anything! What am I missing?



0



Reply



Dietskittles

Don't think of it function-wise as the constructor returning anything, you're right, it doesn't. Think of it language-wise. "in return" is just an English expression used to convey what you end up with, in this context, what you type of object you get after the conversion.

👍 1 ➡ Reply



Seeminglyimplicitexplicitconversion

⌚ November 9, 2024 2:58 am PST

```
1 class Vector2
2 {
3 public:
4     explicit Vector2(double x, double y) : m_x{x}, m_y{y} {}
5
6 private:
7     double m_x{ 0.0 };
8     double m_y{ 0.0 };
9 };
10
11 int main()
12 {
13     Vector2 vec2{ 7, 5 }; // Implicit conversion from int to double
14
15     return 0;
16 }
```

If **"an explicit constructor cannot be used to do implicit conversions (since this uses copy initialization or copy list initialization)"**, why does this work?

It uses direct list initialization, but it still calls the explicit constructor and somewhere in there is an implicit conversion, is it not?

📝 Last edited 7 months ago by [Seeminglyimplicitexplicitconversion](#)

👍 2 ➡ Reply



Alex Author

🔗 Reply to [Seeminglyimplicitexplicitconversion](#)¹⁰ ⌚ November 10, 2024 9:56 pm PST

Explicit constructors can not be used for implicit conversions to the class type. The arguments to an explicit constructor can still be implicitly converted to the parameter type.

👍 2 ➡ Reply



lamb

⌚ July 13, 2024 12:13 am PDT

In section "Converting constructors", you say "So in this case, when `printFoo(5)` is called, parameter `f` is copy initialized using the `Foo(int)` constructor with 5 as an argument!";

I think the process of copy initialization can be described more precisely, otherwise, there may be a misunderstanding that parameter `f` is directly constructed from argument 5 through the constructor(:(, I make such misunderstanding).

According to [copy initialization](https://en.cppreference.com/w/cpp/language/copy_initialization) (https://en.cppreference.com/w/cpp/language/copy_initialization)¹¹, the argument `5` will first be converted to type `Foo` by converting constructor. The result of the conversion is a prvalue temporary (since C++11) (until C++17), and then this temporary object is used to directly initialize `f` (using the copy constructor here). You can see a demo on <https://godbolt.org/z/s9rsW98nf>

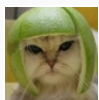
If the function parameter is `{5, 6}` (a list), then copy list initialization will be performed, and copy list initialization will directly call the constructor to initialize `f`, instead of constructing a temporary object first and then copying `f`. You can see a demo on <https://godbolt.org/z/64xW8v4x6>.

It should be noted that the compilation parameters `-std=c++14 -pedantic -fno-elide-constructors` are used in both demos to avoid copy elision.

If I made any mistakes please point them out. Thanks for your time


 Last edited 11 months ago by [lamb](#)

 0  Reply



Alex

Author

 Reply to [lamb](#)¹²  July 14, 2024 6:23 pm PDT

Prior to C++17, it is as you say. As of C++17, the copy is elided, so parameter `f` is directly initialized with the value `5`. No call to the copy constructor is required, and you know it's mandatory elision because it works even if the copy constructor is deleted.

 1  Reply



Sander

 June 1, 2024 9:59 pm PDT

So I'm not understanding why a `std::string_view` constructor that accepts a C-style string is allowed to not be explicit, when one of the arguments to define an explicit constructor is that a value can be made implicitly into a class type to satisfy a function call?

Like what was specified at the beginning of the lecture:

It's entirely possible that the caller assumed this would print 5, and did not expect the compiler to silently and implicitly convert our `int` value to a `Dollars` object so that it could satisfy this function call.

 0  Reply

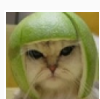


Sander

 Reply to [Sander](#)¹³  June 1, 2024 10:12 pm PDT

I guess it would do two conversions with a C-style string which is disallowed but if you provided the `sv` suffix it'd still implicitly create a class object which you want to avoid right?

 0  Reply



Alex

Author

 Reply to [Sander](#)¹⁴  June 4, 2024 8:28 pm PDT

One of the primary goals of `std::string_view` was to allow functions to accept any kind of string argument (a C-style string, `std::string`, or `std::string_view`).

For this reason, a C-style string and a `std::string` will both implicitly convert to a `std::string_view`, so that either can be used as an argument without having to explicitly cast them.

Since a C-style string and a `std::string_view` are essentially different implementations for the same thing (a string) if the user provides a C-style string and it accidentally calls a function taking a `std::string_view`, the function will probably do what the user expected anyway. On the other hand, an `int` and a `Dollar` are not different implementations for the same thing. One is a generic int and the other is a monetary unit.

👍 1 ➡ Reply



LLY

🕒 May 30, 2024 8:05 pm PDT

Hi Alex,

Could you please help to check if my understanding is correct?

```

1  #include <iostream>
2
3  class Dollar
4  {
5  public:
6      Dollar() = default;
7
8      explicit Dollar(double dollar)
9          : m_dollar {dollar}
10     {
11         std::cout << "constructor Dollar(double) called\n";
12     }
13
14     Dollar(const Dollar& dollar)
15         : m_dollar {dollar.m_dollar}
16     {
17         std::cout << "copy constructor called\n";
18     }
19
20     void print()
21     {
22         std::cout << "Dollar value: " << m_dollar << '\n';
23     }
24
25 private:
26     double m_dollar {0.0};
27 };
28
29
30 int main()
31 {
32     //implicit conversion of integer argument to double parameter of constructor, NOT
    implicit conversion of integer to dollar object
33     Dollar d1{4};
34     d1.print();
35
36     Dollar d2 = {Dollar{5}}; //why this work? isn't this copy list initialization.
37     d2.print();
38
39     Dollar d3{Dollar{5}}; //copy constructor, however, copy elision will happen.
40     Hence, constructor(double) is called instead.
41     d3.print();
42
43     //copy list initialization doesn't work, as this is an implicit conversion from
44     initializer list to Dollar object.
45     // Dollar d3 = {5};
46
47     return 0;
48 }

```

I am confused on the `Dollar d2 = {Dollar{5}}` case. I thought this is also copy-list initialization, and should fail since the constructor `Dollar(double)` has been made explicit. However, it still works, could you please explain to me why?

Thanks in advance for your time and help!

👍 1 ➡ Reply




Alex

Author

🗨 Reply to LLY¹⁵ ⌚ June 2, 2024 1:15 pm PDT

In the `d2` case, `Dollar{5}` performs direct-list initialization (where use of an explicit constructor is allowed), and then `d2` is copy-list-initialized with that object. The copy will be elided in C++17.



edit: corrected copy-initialized to copy-list-initialized

 Last edited 1 year ago by Alex

 1  Reply



LLY

 Reply to Alex¹⁶  June 2, 2024 6:11 pm PDT

Thanks for the reply Alex. I must have gap in my understanding of direct-list-initialization and copy-list-initialization. I have thought that whenever an equal sign is used with braced-init-list, it performs copy-list-initialization. In the case of `d2`, it is initialized with `= { Dollar{5} }`, so although `Dollar{5}` is a temporary object, however since it is enclosed in braced-init-list, it will be considered as copy-list-initialization, not copy-initialization. Is this understanding wrong?

Is there any good resources to understand what does list-initialization really mean, and what is the difference between direct vs copy?

Thanks!

 0  Reply



Alex

Author

 Reply to LLY¹⁷  June 5, 2024 1:07 pm PDT

I made a mistake in my response -- `d2` is copy-list-initialized (not copy-initialized). I corrected the prior comment.



Because copy constructors are never marked as explicit, they are always a candidate for copy-initialization and copy-list-initialization.

[cppreference.com](https://en.cppreference.com/w/cpp/language/list_initialization) (e.g. https://en.cppreference.com/w/cpp/language/list_initialization) has articles on each kind of initialization (though it's not always easy to follow).

 0  Reply



LLY

 Reply to Alex¹⁸  June 5, 2024 6:04 pm PDT

Thanks Alex. This really clears up doubts for me. I see that I made a mistake in my original example as well. `Dollar d3 = {5}` doesn't work, because `{5}` first has to be converted to object of type `Dollar` implicitly, which uses the `explicit Dollar(double)` constructor, before it can be used to copy-initialized `d3`. But since the constructor is marked explicit, this throws off an error. This is different from the case of `Dollar d2 = {Dollar{5}}`, where the `Dollar{5}` temporary is already the same type as `d2`, hence only copy-list-initialization is invoked.

Please let me know if this is not correct. Also, all these nuances are really difficult to grasp haha. Thanks!

 0  Reply



Robert

Reply to LLY¹⁹ June 3, 2025 9:58 pm PDT

Hi, I would like to leave a list of my comprehension maybe Alex could make me know if I'm wrong :

```
1 Dollar d; //default initialization, call default
2 constructor
3 Dollar d2 {}; //value initialization, call default
4 constructor
5 Dollar d3 {3}; //direct list initialization, call
6 matching constructor
7 Dollar d4 (3); //direct initialization, call matching
8 constructor
9 Dollar dcopy {d}; //direct list initialization, call
10 matching copy constructor
11 Dollar dcopy2 (d) //direct initialization, call matching
    copy constructor
12
13 //Following are valid ONLY if matching constructor is
14 non-explicit:
15 Dollar d5 = 3; //copy initialization, call matching
16 constructor if its non-explicit
17 Dollar d6 = {3} //copy list initialization, call
18 matching constructor if its non-explicit
    Dollar d7 = {d} //copy list initialization, call
    matching copy constructor (usually copy constructor
    won't be explicit)

```



```
void printDollar(Dollar d);

printDollar(3); //needs conversion int -> double ->
Dollar, so its invalid
printDollar(3.0) //if matching constructor
Dollar(double) is non-explicit will be called
printDollar(Dollar {3}) //temporary Dollar object is
called using direct list initialization and calling
matching constructor, then will be used copy
constructor

```

👍 0

Reply



Phargelm

April 21, 2024 9:21 am PDT

Despite that copy constructors do not perform conversions it seems that making copy constructor `explicit` prevents creating copy at all (when there's no copy elision). I've tried the next code sample:

```

1  #include <iostream>
2
3  class IntPair
4  {
5  private:
6      int m_x{};
7      int m_y{};
8
9  public:
10     explicit IntPair(const IntPair& intPair) // explicit copy constructor
11         : m_x { intPair.m_x }, m_y { intPair.m_y }
12     {
13         std::cout << "Copy!\n";
14     }
15
16     IntPair(int x, int y)
17         : m_x { x }, m_y { y }
18     {}
19
20     int x() const { return m_x; }
21     int y() const { return m_y; }
22 };
23
24 void print(IntPair p)
25 {
26     std::cout << "(" << p.x() << ", " << p.y() << ")\n";
27 }
28
29 int main()
30 {
31     IntPair intPair = {4, 8}; // no errors probably because of copy elision.
32     print(intPair); // compile error "no matching function for call to
33     'IntPair::IntPair(IntPair&)' in gcc
34
35     return 0;
36 }

```

So making a copy constructor `explicit` has side effects at least on my machine. Removing `explicit` keyword leads to compiling and output:

```

1 | Copy!
2 | (4, 8)

```


May it have any practical consequences? Let's say we can prohibit objects copying of some class because it can be expensive and thus force the user to pass it by reference always?

 Last edited 1 year ago by Phargelm

 1  Reply



Phargelm

 Reply to [Phargelm](#)²⁰  April 21, 2024 9:30 am PDT

I see that we can use `= delete` to achieve this goal instead of making copy constructor `explicit`.

 0  Reply



Alex Author

Reply to Phargelm²¹ April 25, 2024 1:32 pm PDT

Making the copy constructor explicit means it can't be used for implicit copies, but can still be used for explicit copies.

Given this line:

```
1 | IntPair i2 { intPair };
```

This works if the copy constructor is explicit, but not if it is deleted.

👍 2 ➡ Reply



Jacob

February 26, 2024 5:21 pm PST

Hi Alex.

Referring to the section labeled "Best practices for use of explicit": In the second sentence you say:

If a conversion is required, only **non-explicit** constructors will be considered. If **no non-explicit constructor can be found** to perform the conversion, the compiler will error. (Emphasis mine)

I'm so confused! I would think just the opposite: If you have declared the constructor explicit, then only an explicit conversion should be considered while a non-explicit conversion should earn a compile error.

What am I missing. (The screws I know about. ;-)

👍 0 ➡ Reply



Alex Author

Reply to Jacob²² February 27, 2024 8:39 pm PST

I think maybe the context that we're talking about implicit conversions didn't carry over from the prior sentence. Rewritten as: "If an implicit conversion is required, only non-explicit constructors will be considered. If no non-explicit constructor can be found to perform the conversion, the compiler will error."

As you'll read, explicit constructors can still be used for explicit conversions.

Make more sense now?

👍 1 ➡ Reply



Jacob

February 21, 2024 5:59 pm PST

I am writing this comment just after playing with the second example, the Employee that wouldn't compile because there were two conversions involved: string->string_view and then string_view->Employee. But what if I skip the string_view business altogether? So I tried this:

```

1  #include <iostream>
2  #include <string>
3  // #include <string_view>
4
5  class Employee
6  {
7  private:
8      std::string m_name{};
9
10 public:
11     Employee(std::string name)
12         : m_name{ name }
13     {
14     }
15
16     const std::string& getName() const { return m_name; }
17 };
18
19 void printEmployee(Employee e) // has an Employee parameter
20 {
21     std::cout << e.getName();
22 }
23
24 int main()
25 {
26     printEmployee("Joe"); // we're supplying an string literal argument
27
28     return 0;
29 }

```

Well, that barfed the same way:

```

1  $ c++ -o str-Emp str-Emp.cpp
2  str-Emp.cpp: In function 'int main()':
3  str-Emp.cpp:26:19: error: could not convert '(const char*)"Joe"' from 'const char*' to
4  'Employee'
5      26 |         printEmployee("Joe"); // we're supplying an string literal argument
6          |                         ^~~~~
7          |                         |
8          |                         const char*

```

Wo Hoppem here? Wild guess: There are still multiple implicit conversions:

1. Literal string->std::string
2. std::string->Employee

The string_view business was a red herring to me.

Have I grokked correctly, O Old One? (Not ready to disincorporate just yet >;-)

👍 0 ➡ Reply



Alex

Author

🔁 Reply to [Jacob](#) ²³ 🕒 February 23, 2024 2:07 pm PST

Yup, it's a double-conversion in the same way.

Also, who you calling old? Even though it's true. :(

👍 1 ➡ Reply



Jacob

Reply to [Alex](#)²⁴ February 26, 2024 5:15 pm PST

Alex, many years ago (I ain't no spring chicken neither) it was a truism that all IT students & professionals had read these two Robert Heinlein novels: "The Moon is a Harsh Mistress" with its sentient computer named Mike, and "Stranger in a Strange Land" which introduces grokking and the wise, discorporated Martian Old Ones. Though I assume and hope you are not ready to discorporate.

1

Reply



Alex

Author

Reply to [Jacob](#)²⁵ February 27, 2024 8:37 pm PST

Sadly, I haven't read either of those novels. I know the term grokking from conventional use, but didn't realize where it originated.

I am not planning on discorporating any time soon! There is still much materialization left for me here.

1

Reply



Estelyen

January 30, 2024 3:57 am PST

This chapter is not listed in the site index for the word 'explicit'. I think it should be.

0

Reply



Alex

Author

Reply to [Estelyen](#)²⁶ January 31, 2024 10:49 pm PST

Fixed. Thanks for pointing this out.

2

Reply



BLANK28

January 15, 2024 1:06 pm PST

Hey Alex,

I didn't get this: "Do not make copy or move constructors explicit, as these do not perform conversions."

Are you saying that copy and move constructors are not used for implicit conversions by the compiler ??

0

Reply



Alex

Author

Reply to [BLANK28](#)²⁷ January 17, 2024 9:04 pm PST

Correct. Copy and move constructors are used for copying/moving one object to another object of the

same type. Since both objects have the same type, this isn't considered to be a conversion.



1



Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/implicit-type-conversion/>
3. <https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/>
4. <https://www.learncpp.com/>
5. <https://www.learncpp.com/cpp-tutorial/class-initialization-and-copy-elision/>
6. <https://www.learncpp.com/convertng-constructors-and-the-explicit-keyword/>
7. <https://www.learncpp.com/cpp-tutorial/overloading-the-assignment-operator/>
8. <https://gravatar.com/>
9. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-608909>
10. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-603972>
11. https://en.cppreference.com/w/cpp/language/copy_initialization
12. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-599537>
13. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-597840>
14. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-597841>
15. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-597757>
16. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-597864>
17. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-597884>
18. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-598004>
19. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-598042>
20. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-596065>
21. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-596066>
22. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-594066>
23. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-593899>
24. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-593959>
25. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-594064>
26. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-593028>
27. <https://www.learncpp.com/cpp-tutorial/convertng-constructors-and-the-explicit-keyword/#comment-592392>
28. <https://g.ezoic.net/privacy/learncpp.com>