What does the explicit keyword mean?

Asked 13 years, 7 months ago Modified 26 days ago Viewed 1.1m times

3401

What does the `explicit` keyword mean in C++?

c++ constructor explicit c++-faq explicit-constructor

Share Edit Follow Flag

edited Jan 24, 2018 at 22:44
Ronny Brendel
4,469 5 34 54

asked Sep 23, 2008 at 13:58
Skizz
67.2k 10 66 108

133 I just want to point out to anyone new coming along that ever since C++11, `explicit` can be applied to more than just constructors. It's now valid when applied to conversion operators as well. Say you have a class `BigInt` with a conversion operator to `int` and an explicit conversion operator to `std::string` for whatever reason. You'll be able to say `int i = myBigInt;`, but you'll have to cast explicitly (using `static_cast`, preferably) in order to say `std::string s = myBigInt;`. – chris Aug 30, 2012 at 16:52

2 Can't explicit also refer to assignment? (i.e. `int x(5);`) – Eitan Myron Feb 27, 2014 at 0:10

4 @curiousguy, There's no such thing as an explicit implicit conversion. – chris Jun 20, 2018 at 14:08

2 @curiousguy, It's not inherently an implicit conversion. Putting `explicit` there declares an explicit conversion to a type. No implicitness involved in the process. – chris Jun 20, 2018 at 19:37

1 @Milan, Yes, that's exactly it. If you're looking for more information, this answer writes it up more formally. Do note that `bool` is special in this regard. Those answers and searching "explicit conversion operators" will lead you to more writeups about this feature and be better suited than a comment chain. – chris Jan 29, 2021 at 1:08

11 Answers

Sorted by: Highest score (default)

3870

The compiler is allowed to make one implicit conversion to resolve the parameters to a function. What this means is that the compiler can use constructors callable with a **single parameter** to convert from one type to another in order to get the right type for a parameter.

Here's an example class with a constructor that can be used for implicit conversions:

```
class Foo
{
private:
  int m_foo;

public:
  // single parameter constructor, can be used as an implicit conversion
  Foo (int foo) : m_foo (foo) {}

  int GetFoo () { return m_foo; }
};
```

Here's a simple function that takes a `Foo` object:

```
void DoBar (Foo foo)
{
  int i = foo.GetFoo ();
}
```

and here's where the `DoBar` function is called:

```
int main ()
{
  DoBar (42);
}
```

The argument is not a `Foo` object, but an `int`. However, there exists a constructor for `Foo` that takes an `int` so this constructor can be used to convert the parameter to the correct type.

The compiler is allowed to do this once for each parameter.

Prefixing the `explicit` keyword to the constructor prevents the compiler from using that constructor for implicit conversions. Adding it to the above class will create a compiler error at the function call `DoBar (42)`. It is now necessary to call for conversion explicitly with `DoBar (Foo (42))`

The reason you might want to do this is to avoid accidental construction that can hide bugs.
Contrived example:

- You have a `MyString` class with a constructor that constructs a string of the given size. You have a function `print(const MyString&)` (as well as an overload `print (char *string)`), and you call `print(3)` (when you actually intended to call `print("3")`). You expect it to print "3", but it prints an empty string of length 3 instead.

Share Edit Follow Flag

edited Jan 19 at 8:55

community wiki
22 revs, 17 users 57%
Skizz

237 nice write up, you might want to mention multi-arg ctors with default params can also act as single arg ctor, e.g., Object(const char* name=NULL, int otype=0). – maccullt Sep 24, 2008 at 1:23

540 I think it should also be mentioned that one should consider making single argument constructors explicit initially (more or less automatically), and removing the explicit keyword only when the implicit conversion is wanted by design. I think contructors should be explicit by default with an 'implicit' keyword to enable them to work as implicit conversions. But that's not how it is. – Michael Burr Aug 26, 2009 at 17:47

52 @thecoshman: You don't declare a parameter `explicit` -- you declare a constructor `explicit`. But yes: your parameters of type `Foo` have to be constructed `explicitly`, they won't be silently constructed by just plugging their constructor's parameters into the function. – Christian Severin Jun 17, 2011 at 12:12

56 Just an FYI that when calling "print(3)" in your example, the function needs to be "print(const MyString &"). The "const" is mandatory here because 3 is converted to a temporary "MyString" object and you can't bind a temporary to a reference unless it's "const" (yet another in a long list of C++ gotchas) – Larry Jul 10, 2012 at 14:52

52 For completeness sake, I am adding that in addition to parameter conversion the *explicit* keyword here will also prevent the use of assignment form of a copy ctor (e.g., Foo myFoo = 42;) and require the explicit forms Foo myFoo = Foo(42); or Foo myFoo(42); – Arbalest Dec 15, 2012 at 0:53

1262

Suppose, you have a class `String`:

```
class String {
public:
    String(int n); // allocate n bytes to the String object
    String(const char *p); // initializes object with char *p
};
```

Now, if you try:

```
String mystring = 'x';
```

The character `'x'` will be implicitly converted to `int` and then the `String(int)` constructor will be called. But, this is not what the user might have intended. So, to prevent such conditions, we shall define the constructor as `explicit`:

```
class String {
public:
    explicit String (int n); //allocate n bytes
    String(const char *p); // initialize sobject with string p
};
```

Share Edit Follow Flag

edited Apr 26, 2018 at 9:36
Azeem
7,629 4 22 36

answered Sep 23, 2008 at 14:09
Eddie
10.7k 1 14 4

18 And it's worth noting that the new generalized initialization rules of C++0x will make `String s = {0};` ill-formed, rather than trying to call the other constructor with a null pointer, as `String s = 0;` would do. – Johannes Schaub - litb Dec 13, 2010 at 8:34

10 Even though this is an old question it seems worth pointing a few things out (or having someone set me straight). By making the int form, or both ctors, 'explicit' you would still have the same bug if you used `String mystring('x')` when you meant `String mystring("x")` wouldn't you? Also, from the comment above I see the improved behavior of `String s = {0}` over `String s = 0` thanks to making the int form of the ctor 'explicit'. But, other than knowing the precedence of the ctors how do you know the intent (i.e. how to spot the bug) of this `String s{0}` ? – Arbalest Aug 5, 2013 at 5:23

Why String mystring = 'x'; is getting converted to int? – InQusitive Mar 22, 2015 at 19:51

12 @InQusitive: `'x'` is being treated as an integer because the `char` data type is just a 1-byte integer. – DavidRR Apr 30, 2015 at 16:55

11 The problem with your example is that it only works with *copy initialization* (using `=`) but not with *direct initialization* (without using `=`): the compiler will still call the `String(int)` constructor without generating an error if you write `String mystring('x');`, as @Arbalest pointed out. The `explicit` keyword is meant for preventing implicit conversions that happen in direct initialization and function resolution. A better solution to your example would be a simple overload of the constructor: `String(char c);`. – Maggyero Aug 11, 2015 at 20:34

180

In C++, a constructor with only one required parameter is considered an implicit conversion function. It converts the parameter type to the class type. Whether this is a good thing or not depends on the semantics of the constructor.

For example, if you have a string class with constructor `String(const char* s)`, that's probably exactly what you want. You can pass a `const char*` to a function expecting a `String`, and the compiler will automatically construct a temporary `String` object for you.

On the other hand, if you have a buffer class whose constructor `Buffer(int size)` takes the size of the buffer in bytes, you probably don't want the compiler to quietly turn `int`s into `Buffer`s. To prevent that, you declare the constructor with the `explicit` keyword:

```
class Buffer { explicit Buffer(int size); ... }
```

That way,

```
void useBuffer(Buffer& buf);
useBuffer(4);
```

becomes a compile-time error. If you want to pass a temporary `Buffer` object, you have to do so explicitly:

```
useBuffer(Buffer(4));
```

In summary, if your single-parameter constructor converts the parameter into an object of your class, you probably don't want to use the `explicit` keyword. But if you have a constructor that simply happens to take a single parameter, you should declare it as `explicit` to prevent the compiler from surprising you with unexpected conversions.

Share Edit Follow Flag

answered Sep 23, 2008 at 16:37
cjm
60.6k 9 123 173

9 `useBuffer` expects an lvalue for his argument, `useBuffer(Buffer(4))` will not work either because of it. Changing it to take a `const Buffer&` or `Buffer&&` or just `Buffer` would make it work. – pqnet Jun 26, 2017 at 13:18

59

The keyword `explicit` accompanies either

- **a constructor of class X that cannot be used to implicitly convert the first (any only) parameter to type X**

> C++ [class.conv.ctor]
>
> 1) A constructor declared without the function-specifier explicit specifies a conversion from the types of its parameters to the type of its class. Such a constructor is called a converting constructor.
>
> 2) An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (8.5) or where casts (5.2.9, 5.4) are explicitly used. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or valueinitialization (8.5).

- **or a conversion function that is only considered for direct initialization and explicit conversion.**

> C++ [class.conv.fct]
>
> 2) A conversion function may be explicit (7.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (8.5). Otherwise, user-defined conversions are not restricted to use in assignments and initializations.

Overview

Explicit conversion functions and constructors can only be used for explicit conversions (direct initialization or explicit cast operation) while non-explicit constructors and conversion functions can be used for implicit as well as explicit conversions.

```
/*
                                 explicit conversion          implicit conversion

 explicit constructor                    yes                          no

 constructor                             yes                          yes

 explicit conversion function            yes                          no

 conversion function                     yes                          yes

*/
```

Example using structures `X, Y, Z` and functions `foo, bar, baz`:

Let's look at a small setup of structures and functions to see the difference between `explicit` and non-`explicit` conversions.

```
struct Z { };

struct X {
  explicit X(int a); // X can be constructed from int explicitly
  explicit operator Z (); // X can be converted to Z explicitly
};

struct Y{
  Y(int a); // int can be implicitly converted to Y
  operator Z (); // Y can be implicitly converted to Z
};

void foo(X x) { }
void bar(Y y) { }
void baz(Z z) { }
```

Examples regarding constructor:

Conversion of a function argument:

```
foo(2);                     // error: no implicit conversion int to X possible
foo(X(2));                  // OK: direct initialization: explicit conversion
foo(static_cast<X>(2));     // OK: explicit conversion

bar(2);                     // OK: implicit conversion via Y(int)
bar(Y(2));                  // OK: direct initialization
bar(static_cast<Y>(2));     // OK: explicit conversion
```

Object initialization:

```
X x2 = 2;                   // error: no implicit conversion int to X possible
X x3(2);                    // OK: direct initialization
X x4 = X(2);                // OK: direct initialization
X x5 = static_cast<X>(2);   // OK: explicit conversion

Y y2 = 2;                   // OK: implicit conversion via Y(int)
Y y3(2);                    // OK: direct initialization
Y y4 = Y(2);                // OK: direct initialization
Y y5 = static_cast<Y>(2);   // OK: explicit conversion
```

Examples regarding conversion functions:

```
X x1{ 0 };
Y y1{ 0 };
```

Conversion of a function argument:

```
baz(x1);                    // error: X not implicitly convertible to Z
baz(Z(x1));                 // OK: explicit initialization
baz(static_cast<Z>(x1));    // OK: explicit conversion

baz(y1);                    // OK: implicit conversion via Y::operator Z()
baz(Z(y1));                 // OK: direct initialization
baz(static_cast<Z>(y1));    // OK: explicit conversion
```

Object initialization:

```
Z z1 = x1;                  // error: X not implicitly convertible to Z
Z z2(x1);                   // OK: explicit initialization
Z z3 = Z(x1);               // OK: explicit initialization
Z z4 = static_cast<Z>(x1);  // OK: explicit conversion

Z z1 = y1;                  // OK: implicit conversion via Y::operator Z()
Z z2(y1);                   // OK: direct initialization
Z z3 = Z(y1);               // OK: direct initialization
Z z4 = static_cast<Z>(y1);  // OK: explicit conversion
```

Why use `explicit` conversion functions or constructors?

Conversion constructors and non-explicit conversion functions may introduce ambiguity.

Consider a structure `V`, convertible to `int`, a structure `U` implicitly constructible from `V` and a function `f` overloaded for `U` and `bool` respectively.

```
struct V {
  operator bool() const { return true; }
};

struct U { U(V) { } };

void f(U) { }
void f(bool) {  }
```

A call to `f` is ambiguous if passing an object of type `V`.

```
V x;
f(x);  // error: call of overloaded 'f(V&)' is ambiguous
```

The compiler does not know wether to use the constructor of `U` or the conversion function to convert the `V` object into a type for passing to `f`.

If either the constructor of `U` or the conversion function of `V` would be `explicit`, there would be no ambiguity since only the non-explicit conversion would be considered. If both are explicit the call to `f` using an object of type `V` would have to be done using an explicit conversion or cast operation.

Conversion constructors and non-explicit conversion functions may lead to unexpected behaviour.

Consider a function printing some vector:

```
void print_intvector(std::vector<int> const &v) { for (int x : v) std::cout << x <<
'\n'; }
```

If the size-constructor of the vector would not be explicit it would be possible to call the function like this:

```
print_intvector(3);
```

What would one expect from such a call? One line containing `3` or three lines containing `0`? (Where the second one is what happens.)

Using the explicit keyword in a class interface enforces the user of the interface to be explicit about a desired conversion.

As Bjarne Stroustrup puts it (in "The C++ Programming Language", 4th Ed., 35.2.1, pp. 1011) on the question why `std::duration` cannot be implicitly constructed from a plain number:

> If you know what you mean, be explicit about it.

Share Edit Follow Flag

edited Jul 12, 2015 at 11:29

answered Jul 10, 2015 at 23:48
Pixelchemist
22.9k 7 44 70

54

This answer is about object creation with/without an explicit constructor since it is not covered in the other answers.

Consider the following class without an explicit constructor:

```
class Foo
{
public:
    Foo(int x) : m_x(x)
    {
    }

private:
    int m_x;
};
```

Objects of class Foo can be created in 2 ways:

```
Foo bar1(10);

Foo bar2 = 20;
```

Depending upon the implementation, the second manner of instantiating class Foo may be confusing, or not what the programmer intended. Prefixing the `explicit` keyword to the constructor would generate a compiler error at `Foo bar2 = 20;`.

It is *usually* good practice to declare single-argument constructors as `explicit`, unless your implementation specifically prohibits it.

Note also that constructors with

- default arguments for all parameters, or
- default arguments for the second parameter onwards

can both be used as single-argument constructors. So you may want to make these also `explicit`.

An example when you would deliberately ***not*** want to make your single-argument constructor explicit is if you're creating a functor (look at the 'add_x' struct declared in this answer). In such a case, creating an object as `add_x add30 = 30;` would probably make sense.

Here is a good write-up on explicit constructors.

Share Edit Follow Flag

edited May 23, 2017 at 11:47
Community Bot
1 1

answered Oct 8, 2013 at 14:43
Gautam
959 12 19

47

The `explicit` keyword makes a conversion constructor to non-conversion constructor. As a result, the code is less error prone.

Share Edit Follow Flag

edited Feb 14, 2013 at 16:40
SaiyanGirl
15.1k 11 39 53

answered Nov 21, 2012 at 2:36
SankararaoMajji
469 4 3

36

The `explicit`-keyword can be used to enforce a constructor to be called *explicitly*.

```
class C {
public:
    explicit C() =default;
};

int main() {
    C c;
    return 0;
}
```

the `explicit`-keyword in front of the constructor `C()` tells the compiler that only explicit call to this constructor is allowed.

The `explicit`-keyword can also be used in user-defined type cast operators:

```
class C{
public:
    explicit inline operator bool() const {
        return true;
    }
};

int main() {
    C c;
    bool b = static_cast<bool>(c);
    return 0;
}
```

Here, `explicit`-keyword enforces only explicit casts to be valid, so `bool b = c;` would be an invalid cast in this case. In situations like these `explicit`-keyword can help programmer to avoid implicit, unintended casts. This usage has been standardized in C++11.

Share Edit Follow Flag

edited Nov 18, 2021 at 22:40
JDługosz
5,333 3 23 41

answered May 14, 2013 at 9:28
Helixirr
863 7 18

18 `C c();` in the first example doesn't mean what you think it means: it's the declaration of a function named `c` that takes no parameters and returns an instance of `C`. – 6502 Feb 19, 2014 at 21:25

2 `explicit operator bool()` is also the C++11 version of safe bool, and can be used implicitly in condition checks (and *only* in condition checks, as far as I'm aware). In your second example, this line would also be valid in `main()`: `if (c) { std::cout << "'c' is valid." << std:: endl; }`. Apart from this, though, it can't be used without explicit casting. – Justin Time - Reinstate Monica Feb 10, 2016 at 18:35

"*constructor to be called explicitly*" no – curiousguy Jun 13, 2018 at 0:20

@JustinTime It's an inane, broken version of the safe bool. The whole idea of explicit implicit conversion is absurd. – curiousguy Jun 13, 2018 at 0:21

@curiousguy True. It seems a bit like a kludge, aimed more at being easily remembered (likely in the hopes of that translating to frequently used) than at following English logic, and designed to not be outright incompatible with previous safe bool implementations (so you're less likely to break something if you swap it in). IMO, at least. – Justin Time - Reinstate Monica Jun 19, 2018 at 7:14

31

Cpp Reference is always helpful!!! Details about explicit specifier can be found here. You may need to look at implicit conversions and copy-initialization too.

Quick look

> The explicit specifier specifies that a constructor or conversion function (since C++11) doesn't allow implicit conversions or copy-initialization.

Example as follows:

```
struct A
{
    A(int) { }      // converting constructor
    A(int, int) { } // converting constructor (C++11)
    operator bool() const { return true; }
};

struct B
{
    explicit B(int) { }
    explicit B(int, int) { }
    explicit operator bool() const { return true; }
};

int main()
{
    A a1 = 1;      // OK: copy-initialization selects A::A(int)
    A a2(2);       // OK: direct-initialization selects A::A(int)
    A a3 {4, 5};   // OK: direct-list-initialization selects A::A(int, int)
    A a4 = {4, 5}; // OK: copy-list-initialization selects A::A(int, int)
    A a5 = (A)1;   // OK: explicit cast performs static_cast
    if (a1) cout << "true" << endl; // OK: A::operator bool()
    bool na1 = a1; // OK: copy-initialization selects A::operator bool()
    bool na2 = static_cast<bool>(a1); // OK: static_cast performs direct-initialization

//  B b1 = 1;      // error: copy-initialization does not consider B::B(int)
    B b2(2);       // OK: direct-initialization selects B::B(int)
    B b3 {4, 5};   // OK: direct-list-initialization selects B::B(int, int)
//  B b4 = {4, 5}; // error: copy-list-initialization does not consider B::B(int,int)
    B b5 = (B)1;   // OK: explicit cast performs static_cast
    if (b5) cout << "true" << endl; // OK: B::operator bool()
//  bool nb1 = b2; // error: copy-initialization does not consider B::operator bool()
    bool nb2 = static_cast<bool>(b2); // OK: static_cast performs direct-initialization
}
```

Share Edit Follow Flag

answered Aug 20, 2016 at 12:45
selfboot
1,361 17 23

1 `explicit operator bool()` vs. `if` is a special case. There is no way to reproduce it with user defined `Bool`, `explicit operator Bool()` and a function called `If`. – curiousguy Jun 13, 2018 at 0:07

23

It is always a good coding practice to make your one argument constructors (including those with default values for `arg2`, `arg3`,...) as already stated. Like always with C++: if you don't - you'll wish you did...

Another good practice for classes is to make copy construction and assignment private (a.k.a. disable it) unless you really need to implement it. This avoids having eventual copies of pointers when using the methods that C++ will create for you by default. An other way to do this is derive from `boost::noncopyable`.

Share Edit Follow Flag

edited May 11, 2020 at 12:46
NAND
655 8 22

answered Oct 1, 2009 at 22:00
fmuecke
8,668 1 19 25

25 This post is written in 2009. Today you don't declare them as private, but rather say `= delete`. – v010dya Oct 2, 2015 at 7:08

8

Constructors append implicit conversion. To suppress this implicit conversion it is required to declare a constructor with a parameter explicit.

In C++11 you can also specify an "operator type()" with such keyword http://en.cppreference.com/w/cpp/language/explicit With such specification you can use operator in terms of explicit conversions, and direct initialization of object.

P.S. When using transformations defined BY USER (via constructors and type conversion operator) it is allowed only one level of implicit conversions used. But you can combine this conversions with other language conversions

- up integral ranks (char to int, float to double);
- standart conversions (int to double);
- convert pointers of objects to base class and to void*;

Share Edit Follow Flag

answered Jan 23, 2015 at 9:26
Konstantin Burlachenko
4,893 2 40 35

0

Other answers are missing one important factor which I am going to mention here.

Along with "delete" keyword, "explicit" allows you to control the way compiler is going to generate special member functions - default constructor, copy constructor, copy-assignment operator, destructor, move constructor and move-assignment.

Refer https://docs.microsoft.com/en-us/cpp/cpp/explicitly-defaulted-and-deleted-functions

Share Edit Follow Flag

answered Apr 19 at 13:07
Manojkumar Khotele
925 11 23