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

{

public:

// single parameter constructor, can be used as an implicit conversion

Foo (int foo) : m\_foo (foo)

{

}

int GetFoo () { return m\_foo; }

private:

int 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(int size) class with a constructor that constructs a string of the given size. You have a function print(const MyString&), 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.

<https://stackoverflow.com/questions/121162/what-does-the-explicit-keyword-mean>