Constructors and member initializer lists

Constructor is a special non-static member function of a class that is used to initialize objects of its class type.

In the definition of a constructor of a class, *member initializer list* specifies the initializers for direct and virtual bases and non-static data members. (Not to be confused with `std::initializer_list`.)

A constructor must not be a coroutine. (since C++20)

Syntax

Constructors are declared using member function declarators of the following form:

class-name **(** *parameter-list*(optional) **)** *except-spec*(optional) *attr*(optional) (1)

Where *class-name* must name the current class (or current instantiation of a class template), or, when declared at namespace scope or in a friend declaration, it must be a qualified class name.

The only specifiers allowed in the *decl-specifier-seq* of a constructor declaration are `friend`, `inline`, `constexpr` (since C++11), `consteval` (since C++20), and `explicit` (in particular, no return type is allowed). Note that cv- and ref-qualifiers are not allowed either: const and volatile semantics of an object under construction don't kick in until the most-derived constructor completes.

The body of a function definition of any constructor, before the opening brace of the compound statement, may include the *member initializer list*, whose syntax is the colon character `:`, followed by the comma-separated list of one or more *member-initializers*, each of which has the following syntax:

class-or-identifier **(** *expression-list*(optional) **)**	(1)	
class-or-identifier braced-init-list	(2)	(since C++11)
parameter-pack **...**	(3)	(since C++11)

1) Initializes the base or member named by *class-or-identifier* using direct initialization or, if *expression-list* is empty, value-initialization
2) Initializes the base or member named by *class-or-identifier* using list-initialization (which becomes value-initialization if the list is empty and aggregate-initialization when initializing an aggregate)
3) Initializes multiple bases using a pack expansion

class-or-identifier - any identifier that names a non-static data member or any type name which names either the class itself (for delegating constructors) or a direct or virtual base.
expression-list - possibly empty, comma-separated list of the arguments to pass to the constructor of the base or member
braced-init-list - brace-enclosed list of comma-separated initializers and nested braced-init-lists
parameter-pack - name of a variadic template parameter pack

Run this code

```
struct S
{
    int n;
    S(int);       // constructor declaration
    S() : n(7) {} // constructor definition:
                  // ": n(7)" is the initializer list
                  // ": n(7) {}" is the function body
};

S::S(int x) : n{x} {} // constructor definition: ": n{x}" is the initializer list

int main()
{
    S s;      // calls S::S()
    S s2(10); // calls S::S(int)
}
```

Explanation

Constructors have no names and cannot be called directly. They are invoked when initialization takes place, and they are selected according to the rules of initialization. The constructors without `explicit` specifier are converting constructors. The constructors with a `constexpr` specifier make their type a *LiteralType*. Constructors that may be called without any argument are default constructors. Constructors that take another object of the same type as the argument are copy constructors and move constructors.

Before the compound statement that forms the function body of the constructor begins executing, initialization of all direct bases, virtual bases, and non-static data members is finished. Member initializer list is the place where non-default initialization of these objects can be specified. For bases and non-static data members that cannot be default-initialized, such as members of reference and const-qualified types, member initializers must be specified. No initialization is performed for anonymous unions or variant members that do not have a member initializer.

The initializers where *class-or-identifier* names a virtual base class are ignored during construction of any class that is not the most derived class of the object that's being constructed.

Names that appear in *expression-list* or *brace-init-list* are evaluated in scope of the constructor:

```
class X
{
    int a, b, i, j;
public:
    const int& r;
    X(int i)
      : r(a) // initializes X::r to refer to X::a
      , b{i} // initializes X::b to the value of the parameter i
      , i(i) // initializes X::i to the value of the parameter i
      , j(this->i) // initializes X::j to the value of X::i
    {}
};
```

Exceptions that are thrown from member initializers may be handled by function-try-block

Member functions (including virtual member functions) can be called from member initializers, but the behavior is undefined if not all direct bases are initialized at that point.

For virtual calls (if the direct bases are initialized at that point), the same rules apply as the rules for the virtual calls from constructors and destructors: virtual member functions behave as if the dynamic type of `*this` is the static type of the class that's being constructed (dynamic dispatch does not propagate down the inheritance hierarchy) and virtual calls (but not static calls) to pure virtual member functions are undefined behavior.

If a non-static data member has a default member initializer and also appears in a member initializer list, then the member initializer is used and the default member initializer is ignored:

```
struct S
{
    int n = 42;   // default member initializer
    S() : n(7) {} // will set n to 7, not 42
};
```

(since C++11)

Reference members cannot be bound to temporaries in a member initializer list:

```
struct A
{
    A() : v(42) {} // Error
    const int& v;
};
```

Note: same applies to default member initializer.

Delegating constructor

If the name of the class itself appears as *class-or-identifier* in the member initializer list, then the list must consist of that one member initializer only; such a constructor is known as the *delegating constructor*, and the constructor selected by the only member of the initializer list is the *target constructor*

In this case, the target constructor is selected by overload resolution and executed first, then the control returns to the delegating constructor and its body is executed.

Delegating constructors cannot be recursive.

(since C++11)

```
class Foo
{
public:
    Foo(char x, int y) {}
    Foo(int y) : Foo('a', y) {} // Foo(int) delegates to Foo(char, int)
};
```

Inheriting constructors

See using declaration.

Initialization order

The order of member initializers in the list is irrelevant: the actual order of initialization is as follows:

1) If the constructor is for the most-derived class, virtual bases are initialized in the order in which they appear in depth-first left-to-right traversal of the base class declarations (left-to-right refers to the appearance in base-specifier lists)
2) Then, direct bases are initialized in left-to-right order as they appear in this class's base-specifier list
3) Then, non-static data member are initialized in order of declaration in the class definition.
4) Finally, the body of the constructor is executed

(Note: if initialization order was controlled by the appearance in the member initializer lists of different constructors, then the destructor wouldn't be able to ensure that the order of destruction is the reverse of the order of construction)

Example

Run this code

```
#include <fstream>
#include <string>
#include <mutex>

struct Base
{
    int n;
};

struct Class : public Base
{
    unsigned char x;
    unsigned char y;
    std::mutex m;
    std::lock_guard<std::mutex> lg;
    std::fstream f;
    std::string s;

    Class(int x) : Base{123}, // initialize base class
        x(x),     // x (member) is initialized with x (parameter)
        y{0},     // y initialized to 0
        f{"test.cc", std::ios::app}, // this takes place after m and lg are initialized
        s(__func__), // __func__ is available because init-list is a part of constructor
        lg(m),    // lg uses m, which is already initialized
        m{}       // m is initialized before lg even though it appears last here
    {}            // empty compound statement

    Class(double a) : y(a + 1),
        x(y), // x will be initialized before y, its value here is indeterminate
        lg(m)
    {} // base class initializer does not appear in the list, it is
       // default-initialized (not the same as if Base() were used, which is value-init)

    Class()
    try // function-try block begins before the function body, which includes init list
      : Class(0.0) // delegate constructor
    {
        // ...
    }
    catch (...)
    {
        // exception occurred on initialization
    }
};

int main()
{
    Class c;
    Class c1(1);
    Class c2(0.1);
}
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 194 (https://cplusplus.github.io/CWG/issues/194.html)	C++98	the declarator syntax of constructors only allows at most one function specifier (e.g. a constructor cannot be declared `inline explicit`)	multiple function specifiers are allowed
CWG 257 (https://cplusplus.github.io/CWG/issues/257.html)	C++98	it was unspecified whether an abstract class should provide member initializers for its virtual base classes	specified as not required and such member initializers are ignored during execution
CWG 1696 (https://cplusplus.github.io/CWG/issues/1696.html)	C++98	reference members could be initialized to temporaries (whose lifetime would end at the end of constructor)	such initialization is ill-formed

References

- C++20 standard (ISO/IEC 14882:2020):
  - 11.4.4 Constructors [class.ctor]
  - 11.10.2 Initializing bases and members [class.base.init]
- C++17 standard (ISO/IEC 14882:2017):
  - 15.1 Constructors [class.ctor]
  - 15.6.2 Initializing bases and members [class.base.init]
- C++14 standard (ISO/IEC 14882:2014):

        - 12.1 Constructors [class.ctor]
        - 12.6.2 Initializing bases and members [class.base.init]
    - C++11 standard (ISO/IEC 14882:2011):
        - 12.1 Constructors [class.ctor]
        - 12.6.2 Initializing bases and members [class.base.init]
    - C++98 standard (ISO/IEC 14882:1998):
        - 12.1 Constructors [class.ctor]
        - 12.6.2 Initializing bases and members [class.base.init]

See also

- copy elision
- converting constructor
- copy assignment
- copy constructor
- default constructor
- destructor
- explicit
- initialization
    - aggregate initialization
    - constant initialization
    - copy initialization
    - default initialization
    - direct initialization
    - list initialization
    - reference initialization
    - value initialization
    - zero initialization
- move assignment
- move constructor
- new