Move constructors

A move constructor of class T is a non-template constructor whose first parameter is `T&&`, `const T&&`, `volatile T&&`, or `const volatile T&&`, and either there are no other parameters, or the rest of the parameters all have default values.

Syntax

class_name (*class_name* &&)	(1)	(since C++11)
class_name (*class_name* &&) = `default;`	(2)	(since C++11)
class_name (*class_name* &&) = `delete;`	(3)	(since C++11)

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.

Explanation

1. Typical declaration of a move constructor.
2. Forcing a move constructor to be generated by the compiler.
3. Avoiding implicit move constructor.

The move constructor is typically called when an object is initialized (by direct-initialization or copy-initialization) from rvalue (xvalue or prvalue) (until C++17) xvalue (since C++17) of the same type, including

- initialization: `T a = std::move(b);` or `T a(std::move(b));`, where b is of type T;
- function argument passing: `f(std::move(a));`, where a is of type T and f is `void f(T t)`;
- function return: `return a;` inside a function such as `T f()`, where a is of type T which has a move constructor.

When the initializer is a prvalue, the move constructor call is often optimized out (until C++17) never made (since C++17), see copy elision.

Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc.) rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For example, moving from a `std::string` or from a `std::vector` may result in the argument being left empty. However, this behavior should not be relied upon. For some types, such as `std::unique_ptr`, the moved-from state is fully specified.

Implicitly-declared move constructor

If no user-defined move constructors are provided for a class type (`struct`, `class`, or `union`), and all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there are no user-declared destructors;
- the implicitly-declared move constructor is not defined as *deleted* due to conditions detailed in the next section, (until C++14)

then the compiler will declare a move constructor as a non-explicit `inline public` member of its class with the signature `T::T(T&&)`.

A class can have multiple move constructors, e.g. both `T::T(const T&&)` and `T::T(T&&)`. If some user-defined move constructors are present, the user may still force the generation of the implicitly declared move constructor with the keyword `default`.

The implicitly-declared (or defaulted on its first declaration) move constructor has an exception specification as described in dynamic exception specification (until C++17) exception specification (since C++17)

Deleted implicitly-declared move constructor

The implicitly-declared or defaulted move constructor for class T is defined as *deleted* if any of the following is true:

- T has non-static data members that cannot be moved (have deleted, inaccessible, or ambiguous move constructors);
- T has direct or virtual base class that cannot be moved (has deleted, inaccessible, or ambiguous move constructors);
- T has direct or virtual base class with a deleted or inaccessible destructor;
- T is a union-like class and has a variant member with non-trivial move constructor;

• T has a non-static data member or a direct or virtual base without a move constructor that is not trivially copyable.	(until C++14)
The deleted implicitly-declared move constructor is ignored by overload resolution (otherwise it would prevent copy-initialization from rvalue).	(since C++14)

Trivial move constructor

The move constructor for class T is trivial if all of the following is true:

- it is not user-provided (meaning, it is implicitly-defined or defaulted);
- T has no virtual member functions;
- T has no virtual base classes;
- the move constructor selected for every direct base of T is trivial;
- the move constructor selected for every non-static class type (or array of class type) member of T is trivial;

- T has no non-static data members of volatile-qualified type. (since C++14)

A trivial move constructor is a constructor that performs the same action as the trivial copy constructor, that is, makes a copy of the object representation as if by `std::memmove`. All data types compatible with the C language (POD types) are trivially movable.

Implicitly-defined move constructor

If the implicitly-declared move constructor is neither deleted nor trivial, it is defined (that is, a function body is generated and compiled) by the compiler if odr-used. For `union` types, the implicitly-defined move constructor copies the object representation (as by `std::memmove`). For non-union class types (`class` and `struct`), the move constructor performs full member-wise move of the object's bases and non-static members, in their initialization order, using direct initialization with an xvalue argument. If this satisfies the requirements of a constexpr constructor, the generated move constructor is `constexpr`.

Notes

To make strong exception guarantee possible, user-defined move constructors should not throw exceptions. For example, `std::vector` relies on `std::move_if_noexcept` to choose between move and copy when the elements need to be relocated.

If both copy and move constructors are provided and no other constructors are viable, overload resolution selects the move constructor if the argument is an rvalue of the same type (an xvalue such as the result of `std::move` or a prvalue such as a nameless temporary (until C++17)), and selects the copy constructor if the argument is an lvalue (named object or a function/operator returning lvalue reference). If only the copy constructor is provided, all argument categories select it (as long as it takes a reference to const, since rvalues can bind to const references), which makes copying the fallback for moving, when moving is unavailable.

A constructor is called a 'move constructor' when it takes an rvalue reference as a parameter. It is not obligated to move anything, the class is not required to have a resource to be moved and a 'move constructor' may not be able to move a resource as in the allowable (but maybe not sensible) case where the parameter is a const rvalue reference (const T&&).

Example

Run this code

```
#include <string>
#include <iostream>
```

```cpp
#include <iomanip>
#include <utility>

struct A
{
    std::string s;
    int k;
    A() : s("test"), k(-1) { }
    A(const A& o) : s(o.s), k(o.k) { std::cout << "move failed!\n"; }
    A(A&& o) noexcept :
           s(std::move(o.s)),       // explicit move of a member of class type
           k(std::exchange(o.k, 0)) // explicit move of a member of non-class type
    { }
};

A f(A a)
{
    return a;
}

struct B : A
{
    std::string s2;
    int n;
    // implicit move constructor B::(B&&)
    // calls A's move constructor
    // calls s2's move constructor
    // and makes a bitwise copy of n
};

struct C : B
{
    ~C() { } // destructor prevents implicit move constructor C::(C&&)
};

struct D : B
{
    D() { }
    ~D() { }          // destructor would prevent implicit move constructor D::(D&&)
    D(D&&) = default; // forces a move constructor anyway
};

int main()
{
    std::cout << "Trying to move A\n";
    A a1 = f(A()); // return by value move-constructs the target from the function parameter
    std::cout << "Before move, a1.s = " << std::quoted(a1.s) << " a1.k = " << a1.k << '\n';
    A a2 = std::move(a1); // move-constructs from xvalue
    std::cout << "After move, a1.s = " << std::quoted(a1.s) << " a1.k = " << a1.k << '\n';

    std::cout << "Trying to move B\n";
    B b1;
    std::cout << "Before move, b1.s = " << std::quoted(b1.s) << "\n";
    B b2 = std::move(b1); // calls implicit move constructor
    std::cout << "After move, b1.s = " << std::quoted(b1.s) << "\n";

    std::cout << "Trying to move C\n";
    C c1;
    C c2 = std::move(c1); // calls copy constructor

    std::cout << "Trying to move D\n";
    D d1;
    D d2 = std::move(d1);
}
```

Output:

```
Trying to move A
Before move, a1.s = "test" a1.k = -1
After move, a1.s = "" a1.k = 0
Trying to move B
Before move, b1.s = "test"
After move, b1.s = ""
```

```
Trying to move C
move failed!
Trying to move D
```