Answer of what is move semantics in cpp by Fred (link) [https://stackoverflow.com/questions/3106110/what-are-move-semantics/3109981#3109981]
I find it easiest to understand move semantics with example code. Let's start with a very simple string class which only holds a pointer to a heap-allocated block of memory:

```
#include <cstring>
#include <algorithm>

class string
{
    char* data;

public:

    string(const char* p)
    {
        size_t size = strlen(p) + 1;
        data = new char[size];
        memcpy(data, p, size);
    }
```

Since we chose to manage the memory ourselves, we need to follow the rule of three. I am going to defer writing the assignment operator and only implement the destructor and the copy constructor for now:

```
    ~string()
    {
        delete[] data;
    }

    string(const string& that)
    {
        size_t size = strlen(that.data) + 1;
        data = new char[size];
        memcpy(data, that.data, size);
    }
```

The copy constructor defines what it means to copy string objects. The parameter `const string& that` binds to all expressions of type string which allows you to make copies in the following examples:

```
string a(x);                                    // Line 1
string b(x + y);                                // Line 2
string c(some_function_returning_a_string());   // Line 3
```

Now comes the key insight into move semantics. Note that only in the first line where we copy `x` is this deep copy really necessary, because we might want to inspect `x` later and would be very surprised if `x` had changed somehow. Did you notice how I just said `x` three times (four times if you include this sentence) and meant the *exact same object* every time? We call expressions such as `x` "lvalues".

The arguments in lines 2 and 3 are not lvalues, but rvalues, because the underlying string objects have no names, so the client has no way to inspect them again at a later point in time. rvalues denote temporary objects which are destroyed at the next semicolon (to be more precise: at the end of the full-expression that lexically contains the rvalue). This is important because during the initialization of `b` and `c`, we could do whatever we wanted with the source string, and *the client couldn't tell a difference*!

C++0x introduces a new mechanism called "rvalue reference" which, among other things, allows us to detect rvalue arguments via function overloading. All we have to do is write a constructor with an rvalue reference parameter. Inside that constructor we can do *anything we want* with the source, as long as we leave it in *some* valid state:

```
    string(string&& that)   // string&& is an rvalue reference to a string
    {
        data = that.data;
        that.data = nullptr;
    }
```

What have we done here? Instead of deeply copying the heap data, we have just copied the pointer and then set the original pointer to null. In effect, we have "stolen" the data that originally belonged to the source string. Again, the key insight is that under no circumstance could the client detect that the source had been modified. Since we don't really do a copy here, we call this constructor a "move constructor". Its job is to move resources from one object to another instead of copying them.

Congratulations, you now understand the basics of move semantics! Let's continue by implementing the assignment operator. If you're unfamiliar with the copy and swap idiom, learn it and come back, because it's an awesome C++ idiom related to exception safety.

```
    string& operator=(string that)
    {
        std::swap(data, that.data);
        return *this;
    }
};
```

Huh, that's it? "Where's the rvalue reference?" you might ask. "We don't need it here!" is my answer :)

Note that we pass the parameter `that` *by value*, so `that` has to be initialized just like any other string object. Exactly how is `that` going to be initialized? In the olden days of C++98, the answer would have been "by the copy constructor". In C++0x, the compiler chooses between the copy constructor and the move constructor based on whether the argument to the assignment operator is an lvalue or an rvalue.

So if you say `a = b`, the *copy constructor* will initialize `that` (because the expression `b` is an lvalue), and the assignment operator swaps the contents with a freshly created, deep copy. That is the very definition of the copy and swap idiom -- make a copy, swap the contents with the copy, and then get rid of the copy by leaving the scope. Nothing new here.

But if you say `a = x + y`, the *move constructor* will initialize `that` (because the expression `x + y` is an rvalue), so there is no deep copy involved, only an efficient move.
`that` is still an independent object from the argument, but its construction was trivial, since the heap data didn't have to be copied, just moved. It wasn't necessary to copy it because `x + y` is an rvalue, and again, it is okay to move from string objects denoted by rvalues.

To summarize, the copy constructor makes a deep copy, because the source must remain untouched.
The move constructor, on the other hand, can just copy the pointer and then set the pointer in the source to null. It is okay to "nullify" the source object in this manner, because the client has no way of inspecting the object again.

I hope this example got the main point across. There is a lot more to rvalue references and move semantics which I intentionally left out to keep it simple. If you want more details please see my supplementary answer.

My first answer was an extremely simplified introduction to move semantics, and many details were left out on purpose to keep it simple.
However, there is a lot more to move semantics, and I thought it was time for a second answer to fill the gaps.
The first answer is already quite old, and it did not feel right to simply replace it with a completely different text. I think it still serves well as a first introduction. But if you want to dig deeper, read on :)

Stephan T. Lavavej took the time provide valuable feedback. Thank you very much, Stephan!

Introduction

Move semantics allows an object, under certain conditions, to take ownership of some other object's external resources. This is important in two ways:

1. Turning expensive copies into cheap moves. See my first answer for an example. Note that if an object does not manage at least one external resource (either directly, or indirectly through its member objects), move semantics will not offer any advantages over copy semantics. In that case, copying an object and moving an object means the exact same thing:

   ```
   class cannot_benefit_from_move_semantics
   {
       int a;        // moving an int means copying an int
       float b;      // moving a float means copying a float
       double c;     // moving a double means copying a double
       char d[64];   // moving a char array means copying a char array

      // ...


   };
   ```

2. Implementing safe "move-only" types; that is, types for which copying does not make sense, but moving does. Examples include locks, file handles, and smart pointers with unique ownership semantics. Note: This answer discusses `std::auto_ptr`, a deprecated C++98 standard library template, which was replaced by `std::unique_ptr` in C++11. Intermediate C++ programmers are probably at least somewhat familiar with `std::auto_ptr`, and because of the "move semantics" it displays, it seems like a good starting point for discussing move semantics in C++11. YMMV.

   What is a move?

The C++98 standard library offers a smart pointer with unique ownership semantics called `std::auto_ptr<T>`. In case you are unfamiliar with `auto_ptr`, its purpose is to guarantee that a dynamically allocated object is always released, even in the face of exceptions:

```
{
    std::auto_ptr<Shape> a(new Triangle);
    // ...
    // arbitrary code, could throw exceptions
```

```
    // ...
}   // <--- when a goes out of scope, the triangle is deleted automatically
```

The unusual thing about `auto_ptr` is its "copying" behavior:

```
auto_ptr<Shape> a(new Triangle);

      +---------------+
      | triangle data |
      +---------------+
        ^
        |
        |
        |
  +-----|---+
  |   +-|-+ |
a | p | | | |
  |   +---+ |
  +---------+

auto_ptr<Shape> b(a);

      +---------------+
      | triangle data |
      +---------------+
        ^
        |
        +----------------------+
                               |
  +---------+            +-----|---+
  |   +---+ |            |   +-|-+ |
a | p |   | |          b | p | | | |
  |   +---+ |            |   +---+ |
  +---------+            +---------+
```

Note how the initialization of `b` with `a` does *not* copy the triangle, but instead transfers the ownership of the triangle from `a` to `b`. We also say "`a` is *moved into* `b`" or "the triangle is *moved* from `a` *to* `b`". This may sound confusing, because the triangle itself always stays at the same place in memory.

> To move an object means to transfer ownership of some resource it manages to another object.

The copy constructor of `auto_ptr` probably looks something like this (somewhat simplified):

```
auto_ptr(auto_ptr& source)   // note the missing const
{
```

```
        p = source.p;
        source.p = 0;   // now the source no longer owns the object
    }
```

Dangerous and harmless moves

The dangerous thing about `auto_ptr` is that what syntactically looks like a copy is actually a move. Trying to call a member function on a moved-from `auto_ptr` will invoke undefined behavior, so you have to be very careful not to use an `auto_ptr` after it has been moved from:

```
auto_ptr<Shape> a(new Triangle);   // create triangle
auto_ptr<Shape> b(a);              // move a into b
double area = a->area();           // undefined behavior
```

But `auto_ptr` is not *always* dangerous. Factory functions are a perfectly fine use case for `auto_ptr`:

```
auto_ptr<Shape> make_triangle()
{
    return auto_ptr<Shape>(new Triangle);
}

auto_ptr<Shape> c(make_triangle());      // move temporary into c
double area = make_triangle()->area();   // perfectly safe
```

Note how both examples follow the same syntactic pattern:

```
auto_ptr<Shape> variable(expression);
double area = expression->area();
```

And yet, one of them invokes undefined behavior, whereas the other one does not. So what is the difference between the expressions `a` and `make_triangle()`? Aren't they both of the same type? Indeed they are, but they have different *value categories*.

Value categories

Obviously, there must be some profound difference between the expression `a` which denotes

an `auto_ptr` variable, and the expression `make_triangle()` which denotes the call of a function that returns an `auto_ptr` by value, thus creating a fresh temporary `auto_ptr` object every time it is called. `a` is an example of an *lvalue*, whereas `make_triangle()` is an example of an *rvalue*.

Moving from lvalues such as `a` is dangerous, because we could later try to call a member function via `a`, invoking undefined behavior. On the other hand, moving from rvalues such as `make_triangle()` is perfectly safe, because after the copy constructor has done its job, we cannot use the temporary again. There is no expression that denotes said temporary; if we simply write `make_triangle()` again, we get a *different* temporary. In fact, the moved-from temporary is already gone on the next line:

```
auto_ptr<Shape> c(make_triangle());
                                  ^ the moved-from temporary dies right here
```

Note that the letters `l` and `r` have a historic origin in the left-hand side and right-hand side of an assignment. This is no longer true in C++, because there are lvalues which cannot appear on the left-hand side of an assignment (like arrays or user-defined types without an assignment operator), and there are rvalues which can (all rvalues of class types with an assignment operator).

> An rvalue of class type is an expression whose evaluation creates a temporary object. Under normal circumstances, no other expression inside the same scope denotes the same temporary object.

Rvalue references

We now understand that moving from lvalues is potentially dangerous, but moving from rvalues is harmless. If C++ had language support to distinguish lvalue arguments from rvalue arguments, we could either completely forbid moving from lvalues, or at least make moving from lvalues *explicit* at call site, so that we no longer move by accident.

C++11's answer to this problem is *rvalue references*. An rvalue reference is a new kind of reference that only binds to rvalues, and the syntax is `X&&`. The good old reference `X&` is now known as an *lvalue reference*. (Note that `X&&` is *not* a reference to a reference; there is no such thing in C++.)

If we throw `const` into the mix, we already have four different kinds of references. What kinds of expressions of type `X` can they bind to?

```
              lvalue   const lvalue   rvalue   const rvalue
-------------------------------------------------------------
X&            yes
const X&      yes      yes           yes      yes
X&&                                  yes
const X&&                            yes      yes
```

In practice, you can forget about `const X&&`. Being restricted to read from rvalues is not very useful.

> An rvalue reference `X&&` is a new kind of reference that only binds to rvalues.

Implicit conversions

Rvalue references went through several versions. Since version 2.1, an rvalue reference `X&&` also binds to all value categories of a different type `Y`, provided there is an implicit conversion from `Y` to `X`. In that case, a temporary of type `X` is created, and the rvalue reference is bound to that temporary:

```
void some_function(std::string&& r);

some_function("hello world");
```

In the above example, `"hello world"` is an rvalue of type `const char[12]`. Since there is an implicit conversion from `const char[12]` through `const char*` to `std::string`, a temporary of type `std::string` is created, and `r` is bound to that temporary. This is one of the cases where the distinction between rvalues (expressions) and temporaries (objects) is a bit blurry.

Move constructors

A useful example of a function with an `X&&` parameter is the *move constructor* `X::X(X&& source)`. Its purpose is to transfer ownership of the managed resource from the source into the current object.

In C++11, `std::auto_ptr<T>` has been replaced by `std::unique_ptr<T>` which takes advantage of rvalue references. I will develop and discuss a simplified version of `unique_ptr`. First, we encapsulate a raw pointer and overload the operators `->` and `*`, so our class feels like a pointer:

```
template<typename T>
class unique_ptr
{
    T* ptr;

public:

    T* operator->() const
    {
        return ptr;
    }

    T& operator*() const
    {
        return *ptr;
    }
```

The constructor takes ownership of the object, and the destructor deletes it:

```
    explicit unique_ptr(T* p = nullptr)
    {
        ptr = p;
    }

    ~unique_ptr()
    {
        delete ptr;
    }
```

Now comes the interesting part, the move constructor:

```
    unique_ptr(unique_ptr&& source)   // note the rvalue reference
    {
        ptr = source.ptr;
        source.ptr = nullptr;
    }
```

This move constructor does exactly what the `auto_ptr` copy constructor did, but it can only be supplied with rvalues:

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);                 // error
unique_ptr<Shape> c(make_triangle());   // okay
```

The second line fails to compile, because `a` is an lvalue, but the parameter `unique_ptr&& source` can only be bound to rvalues. This is exactly what we wanted; dangerous moves should never be implicit. The third line compiles just fine, because `make_triangle()` is an rvalue. The move constructor will transfer ownership from the temporary to `c`. Again, this is exactly what we wanted.

> The move constructor transfers ownership of a managed resource into the current object.

Move assignment operators

The last missing piece is the move assignment operator. Its job is to release the old resource and acquire the new resource from its argument:

```
    unique_ptr& operator=(unique_ptr&& source)   // note the rvalue reference
    {
        if (this != &source)    // beware of self-assignment
        {
            delete ptr;         // release the old resource

            ptr = source.ptr;   // acquire the new resource
            source.ptr = nullptr;
        }
        return *this;
    }
};
```

Note how this implementation of the move assignment operator duplicates logic of both the destructor and the move constructor. Are you familiar with the copy-and-swap idiom? It can also be applied to move semantics as the move-and-swap idiom:

```
    unique_ptr& operator=(unique_ptr source)   // note the missing reference
    {
        std::swap(ptr, source.ptr);
        return *this;
    }
};
```

Now that `source` is a variable of type `unique_ptr`, it will be initialized by the move constructor; that is, the argument will be moved into the parameter. The argument is still required to be an rvalue, because the move constructor itself has an rvalue reference parameter. When control flow reaches the closing brace of `operator=`, `source` goes out of

scope, releasing the old resource automatically.

> The move assignment operator transfers ownership of a managed resource into the current object, releasing the old resource.
> The move-and-swap idiom simplifies the implementation.

Moving from lvalues

Sometimes, we want to move from lvalues. That is, sometimes we want the compiler to treat an lvalue as if it were an rvalue, so it can invoke the move constructor, even though it could be potentially unsafe.
For this purpose, C++11 offers a standard library function template called `std::move` inside the header `<utility>`.
This name is a bit unfortunate, because `std::move` simply casts an lvalue to an rvalue; it does *not* move anything by itself. It merely *enables* moving. Maybe it should have been named `std::cast_to_rvalue` or `std::enable_move`, but we are stuck with the name by now.

Here is how you explicitly move from an lvalue:

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);              // still an error
unique_ptr<Shape> c(std::move(a));   // okay
```

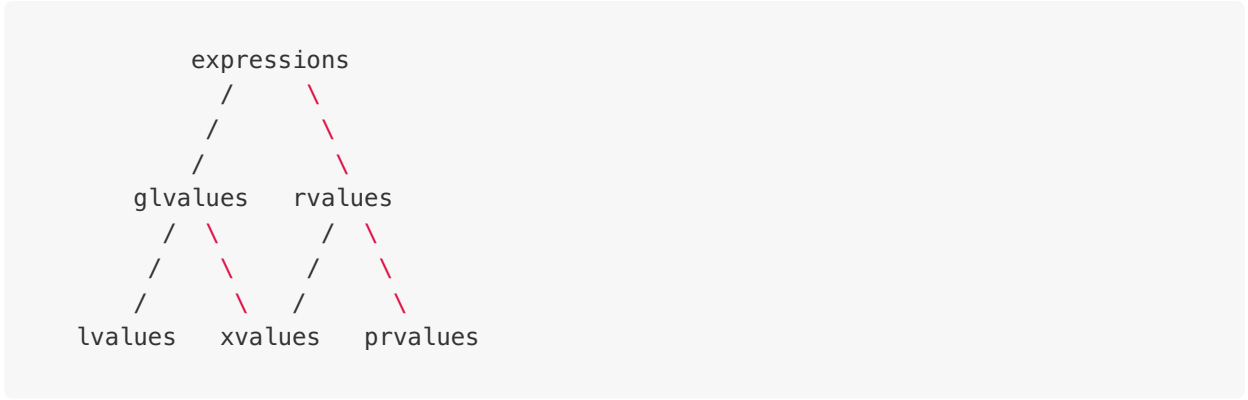
Note that after the third line, `a` no longer owns a triangle. That's okay, because by *explicitly* writing `std::move(a)`, we made our intentions clear: "Dear constructor, do whatever you want with `a` in order to initialize `c`; I don't care about `a` anymore. Feel free to have your way with `a`."

> `std::move(some_lvalue)` casts an lvalue to an rvalue, thus enabling a subsequent move.

Xvalues

Note that even though `std::move(a)` is an rvalue, its evaluation does *not* create a temporary object. This conundrum forced the committee to introduce a third value category. Something that can be bound to an rvalue reference, even though it is not an rvalue in the traditional sense, is called an *xvalue* (eXpiring value). The traditional rvalues were renamed to *prvalues* (Pure rvalues).

Both prvalues and xvalues are rvalues. Xvalues and lvalues are both *glvalues* (Generalized lvalues). The relationships are easier to grasp with a diagram:



Note that only xvalues are really new; the rest is just due to renaming and grouping.

> C++98 rvalues are known as prvalues in C++11. Mentally replace all occurrences of "rvalue" in the preceding paragraphs with "prvalue".

Moving out of functions

So far, we have seen movement into local variables, and into function parameters. But moving is also possible in the opposite direction. If a function returns by value, some object at call site (probably a local variable or a temporary, but could be any kind of object) is initialized with the expression after the `return` statement as an argument to the move constructor:

```
unique_ptr<Shape> make_triangle()
{
    return unique_ptr<Shape>(new Triangle);
}          \-----------------------------/
                  |
                  | temporary is moved into c
                  |
                  v
unique_ptr<Shape> c(make_triangle());
```

Perhaps surprisingly, automatic objects (local variables that are not declared as `static`) can also be *implicitly* moved out of functions:

```
unique_ptr<Shape> make_square()
{
    unique_ptr<Shape> result(new Square);
    return result;   // note the missing std::move
```

```
}
```

How come the move constructor accepts the lvalue `result` as an argument? The scope of `result` is about to end, and it will be destroyed during stack unwinding. Nobody could possibly complain afterwards that `result` had changed somehow; when control flow is back at the caller, `result` does not exist anymore! For that reason, C++11 has a special rule that allows returning automatic objects from functions without having to write `std::move`. In fact, you should *never* use `std::move` to move automatic objects out of functions, as this inhibits the "named return value optimization" (NRVO).

> Never use `std::move` to move automatic objects out of functions.

Note that in both factory functions, the return type is a value, not an rvalue reference. Rvalue references are still references, and as always, you should never return a reference to an automatic object; the caller would end up with a dangling reference if you tricked the compiler into accepting your code, like this:

```
unique_ptr<Shape>&& flawed_attempt()   // DO NOT DO THIS!
{
    unique_ptr<Shape> very_bad_idea(new Square);
    return std::move(very_bad_idea);   // WRONG!
}
```

> Never return automatic objects by rvalue reference. Moving is exclusively performed by the move constructor, not by `std::move`, and not by merely binding an rvalue to an rvalue reference.

Moving into members

Sooner or later, you are going to write code like this:

```
class Foo
{
    unique_ptr<Shape> member;

public:

    Foo(unique_ptr<Shape>&& parameter)
    : member(parameter)   // error
    {}
};
```

Basically, the compiler will complain that `parameter` is an lvalue. If you look at its type, you see an rvalue reference, but an rvalue reference simply means "a reference that is bound to an rvalue"; it does *not* mean that the reference itself is an rvalue! Indeed, `parameter` is just an ordinary variable with a name. You can use `parameter` as often as you like inside the body of the constructor, and it always denotes the same object. Implicitly moving from it would be dangerous, hence the language forbids it.

> A named rvalue reference is an lvalue, just like any other variable.

The solution is to manually enable the move:

```
class Foo
{
    unique_ptr<Shape> member;

public:

    Foo(unique_ptr<Shape>&& parameter)
    : member(std::move(parameter))   // note the std::move
    {}
};
```

You could argue that `parameter` is not used anymore after the initialization of `member`. Why is there no special rule to silently insert `std::move` just as with return values? Probably because it would be too much burden on the compiler implementors. For example, what if the constructor body was in another translation unit? By contrast, the return value rule simply has to check the symbol tables to determine whether or not the identifier after the `return` keyword denotes an automatic object.

You can also pass `parameter` by value. For move-only types like `unique_ptr`, it seems there is no established idiom yet. Personally, I prefer pass by value, as it causes less clutter in the interface.

Special member functions

C++98 implicitly declares three special member functions on demand, that is, when they are needed somewhere: the copy constructor, the copy assignment operator and the destructor.

```
X::X(const X&);              // copy constructor
X& X::operator=(const X&);   // copy assignment operator
X::~X();                     // destructor
```

Rvalue references went through several versions. Since version 3.0, C++11 declares two additional special member functions on demand: the move constructor and the move assignment operator. Note that neither VC10 nor VC11 conform to version 3.0 yet, so you will have to implement them yourself.

```
X::X(X&&);                  // move constructor
X& X::operator=(X&&);       // move assignment operator
```

These two new special member functions are only implicitly declared if none of the special member functions are declared manually. Also, if you declare your own move constructor or move assignment operator, neither the copy constructor nor the copy assignment operator will be declared implicitly.

What do these rules mean in practice?

> If you write a class without unmanaged resources, there is no need to declare any of the five special member functions yourself, and you will get correct copy semantics and move semantics for free. Otherwise, you will have to implement the special member functions yourself. Of course, if your class does not benefit from move semantics, there is no need to implement the special move operations.

Note that the copy assignment operator and the move assignment operator can be fused into a single, unified assignment operator, taking its argument by value:

```
X& X::operator=(X source)    // unified assignment operator
{
    swap(source);            // see my first answer for an explanation
    return *this;
}
```

This way, the number of special member functions to implement drops from five to four. There is a tradeoff between exception-safety and efficiency here, but I am not an expert on this issue.

Forwarding references (previously known as *Universal references*)

Consider the following function template:

```
template<typename T>
```

```
void foo(T&&);
```

You might expect `T&&` to only bind to rvalues, because at first glance, it looks like an rvalue reference. As it turns out though, `T&&` also binds to lvalues:

```
foo(make_triangle());   // T is unique_ptr<Shape>, T&& is unique_ptr<Shape>&&
unique_ptr<Shape> a(new Triangle);
foo(a);                 // T is unique_ptr<Shape>&, T&& is unique_ptr<Shape>&
```

If the argument is an rvalue of type `X`, `T` is deduced to be `X`, hence `T&&` means `X&&`. This is what anyone would expect.
But if the argument is an lvalue of type `X`, due to a special rule, `T` is deduced to be `X&`, hence `T&&` would mean something like `X& &&`. But since C++ still has no notion of references to references, the type `X& &&` is *collapsed* into `X&`. This may sound confusing and useless at first, but reference collapsing is essential for *perfect forwarding* (which will not be discussed here).

> T&& is not an rvalue reference, but a forwarding reference. It also binds to lvalues, in which case `T` and `T&&` are both lvalue references.

If you want to constrain a function template to rvalues, you can combine SFINAE with type traits:

```
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_rvalue_reference<T&&>::value, void>::type
foo(T&&);
```

Implementation of move

Now that you understand reference collapsing, here is how `std::move` is implemented:

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& t)
{
    return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

As you can see, `move` accepts any kind of parameter thanks to the forwarding reference `T&&`, and it returns an rvalue reference. The `std::remove_reference<T>::type` meta-function call is necessary because otherwise, for lvalues of type `X`, the return type would be `X& &&`, which would collapse into `X&`. Since `t` is always an lvalue (remember that a named rvalue reference is an lvalue), but we want to bind `t` to an rvalue reference, we have to explicitly cast `t` to the correct return type.
The call of a function that returns an rvalue reference is itself an xvalue. Now you know where xvalues come from ;)

> The call of a function that returns an rvalue reference, such as `std::move`, is an xvalue.

Note that returning by rvalue reference is fine in this example, because `t` does not denote an automatic object, but instead an object that was passed in by the caller.

```
[1]: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4164.pdf
[2]: http://en.cppreference.com/w/cpp/language/sfinae
```