C++ Under the Hood

Investigating selected features

Written by Amir Kirsh

INCREDIBUILD

Table of Contents

The evolution of C++ over the years revolves around two major developments: additions to the language itself and additions to the standard library.

Additions to the language include things like: rvalue reference, lambda expressions, new contextual keywords like override and final, range-based for, auto variable declarations, decltype, variadic templates, concepts in C++20, and more.

Such additions to the language require the compiler to be able to parse and cope with these new features.

Additions to the standard library include things like: smart pointers, declval, threads, atomic variables, std::optional, std::any and more.

Eating our own dog food

In the standard documents, the library part gets its own section under the [library] tag, For example, see the draft document: http://eel.is/c++draft/library

Almost all library parts are based on the C++ language, without any tricks or hidden magic, in a way that can be implemented on your own. I tend to address this as "we eat our own dog food", since all the libraries rely on the C++ core (see origins of this beautiful term here: https://en.wikipedia.org/wiki/Eating_your_own_dog_food#Origin_of_the_term).

The standard itself doesn't provide any implementation details for library parts (i.e. how to implement), however it does mention requirements to dictate a uniform API and behavior, such as concurrency requirements. For example, for associative containers:
The insert and emplace members shall not affect the validity of iterators and references to the container, and the erase members shall invalidate only iterators and references to the erased elements. , (see: http://eel.is/c++draft/associative.reqmts#9).

This paper will review specific library parts of the C++ language and show how they can be implemented. Specifically, we will check if there is hidden magic inside these features or whether we can implement our own personalized version. Note that implementing our own version is mainly for understanding how things are implemented and not for replacing parts of the standard library, which is not something that is recommended.

The paper will discuss:

- std::any
- std::enable_if
- std::common_type
- std::declval

What is std::any?

In JavaScript, like in other scripting languages, one can write:

```
var a = 3;    // Number
a = "hello"; // String
a = {x: "foo", y: 3.5}; // Object
a = function(msg){alert(msg);}; // function
```

The variable a in the example above is born as one specific type (type Number in JavaScript), but it changes during its lifetime.

Can we do the same in C++?

Well, if you think of 'auto' as a tool for achieving this, you are unfortunately mistaken.

```
auto a = 3;  // int
a = "hello"; // compilation error
              // cannot bind const char* to int
```

However, you can still achieve something similar in C++ by using std::any.

The idea for a type that can hold anything was introduced by Kevlin Henney on the July-August-2000 issue of the "C++ Report" magazine. A copy of the original paper can be found here: http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf

The rumor says that Henney wanted to call this new idea "Henney" after his name but was eventually convinced that "any" is more suitable.

In 2001 it was added into Boost, as boost::any:
https://scicomp.ethz.ch/public/manual/Boost/1.55.0/any.pdf

And then it was added into C++17, as std::any.

With any you can do things like:

```
std::any a = 3; // holding int
a = "hello";  // holding const char*
a = []{std::cout << "I'm a lambda"}; // now holding a lambda!
```

Yes, the variable 'a' seems to change its type at runtime and this is perfectly valid C++.
Well I say that it seems to change its type, since it doesn't change its type actually -- the type is std::any all along.

But how can std::any hold anything? What is the inner member that does the trick? Is it black magic or something that we can do at home?

Under the hood of std::any

A first idea for implementing our own any can be to hold a member of type void* to manage all different types. In either case, any copies the argument that it gets so there is a dynamic allocation that can be held by void*.

However, there is a problem.

There is a need to deallocate the value held by any when it dies (i.e. in the destructor) and when it gets another value (i.e. in the assignment operator). And you cannot perform delete on void*. So we need to remember the type, somehow.

So maybe template can assist here?

If any would be templated it can remember its type, like this: any<T>

Well, this also cannot work since it would not allow assigning const char* to any<int> and the whole idea is not to have a type for any and allow it to hold anything.

So how can we hold the actual type, to allow deallocation, but still without managing any as specific to a certain type?

The answer lies in:
a. keeping any a simple regular class, i.e. not a template class
b. having a template inner class inside any - let's call it 'holder' - with a member of this 'holder' to hold the actual type, hiding it inside any.
c. having a template constructor in any that would allow any to get any parameter:

```
template < typename T >
any(T t) : ptr(new holder<T>(std::move(t))) {}
```

So we aim for something like:

```
class any {
    template < typename T >
    struct holder {
        T value;
        holder(T&& t) : value(std::move(t)) {}
    };


    holder * ptr = nullptr; // problem here, holder is templated

public:
    any()  {}
    ~any() { delete ptr; }

    template < typename T >
    any(T t) : ptr(new holder<T>(std::move(t))) {}

    any& operator=(any a) {
        std::swap(ptr, a.ptr);
        return *this;
    }
  // ...
};
```

The code above is still not good. We cannot hold a member like:

```
holder *
```

class holder is templated thus we must provide the template parameter.
But we cannot hold it as

```
holder<T> *
```

as the type T is not defined in any, it is known only in the constructor of any but not in any itself.

The solution is to add a non-templated base class base_holder with a virtual destructor, holding in any a pointer to base_holder which actually points to holder<T>.

This should look like this:

```
class any {

    struct base_holder {
        virtual ~base_holder() {}
        // ...
    };

    template < typename T >
    struct holder: base_holder {
        T value;
        holder(T&& t) : value(std::move(t)) {}
    };


    base_holder * ptr = nullptr; // that's ok now

public:
    any()  {}
    ~any() { delete ptr; }

    template < typename T >
    any(T t) : ptr(new holder<T>(std::move(t))) {}

    any& operator=(any a) {
        std::swap(ptr, a.ptr);
        return *this;
    }
  // ...
};
```

Since T is unknown inside any it is said to be "erased".

Usage of std::any

Unfortunately the usage of std::any is not so simple. In order to extract the value stored inside, you should indicate the type, as shown in the example below:

```
std::any a = 3; // holding int
std::cout << a; // compilation error, std::cout cannot print std::any
std::cout << (int)a; // compilation error, almost..., but not yet
std::cout << std::any_cast<int>(a); // ok - prints 3
```

Allowing automatic casting from std::any to the actual type it holds would have been very useful, but this is not possible as the actual type is "erased" and is unknown for std::any which requires the programmer to perform std::any_cast.

The actual type held inside std::any can be retrieved at runtime using std::any type() method which returns type_info, but the usage of this info is suitable only for if-else statements and not for overloading.

Usage summary:

"Use std::any where in the past you would have used void*. Which is to say, ideally, almost nowhere."

Richard Hodges
https://stackoverflow.com/questions/52715219/when-should-i-use-stdany

SUMMARY OF THINGS 1

The new features introduced into C++ can be divided into two groups. The first group includes syntax additions to the language itself, like: rvalue reference, lambda expressions, new contextual keywords like override and final and more. Such additions require compiler adaptations. The second group includes additions to the standard library, like: smart pointers, declval, threads, atomic variables and more. The special part about the second group is that it is almost always based on the C++ language itself without any tricks or hidden magic.

SUMMARY OF THINGS 2

The standard itself doesn't pose any implementation details for library parts (i.e. how to implement), however it does pose requirements that dictate a uniform API and behavior. Our goal in this paper is to review several language attributes and see how they can be implemented.

The next features that would be discussed are tools used with template SFINAE -- std::enable_if, std::common_type and std::declval.

before we continue, what is SFINAE?

The term SFINAE stands for: **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror

SFINAE is nothing new, it is out there from C++98. So you probably heard about it. The term has regained traction since C++11 as some new elements were added to the language to support SFINAE.

The idea behind SFINAE is simple. When the compiler considers a template as a resolution for a type or a method, calling on the template signature might not fit the actual template parameters provided. In such cases, the template is considered as "not fitting". The compiler continues looking for a better fit.

Example:

Let's assume that we have the following template method for all integral numbers (short, int, long etc.):

```
template<class T>
bool print(const T& t) {
  std::cout "integer number: " << t;
}
```

This method has two problems:
a. it is too greedy, it accepts everything, not only integral numbers
b. in case it gets a T that didn't implement extractor operator into ostream. i.e. operator<<(std::ostream& out, const T& t) -- it would result with a compilation error

Suppose that we do have another template method for floating numbers:

```
template<class T>
bool print(const T& t) {
  std::cout "floating point number: " << t;
}
```

We want to be able to call on our print method, which is actually not a single method but two separate template methods. However, they share exactly the same signature which would result in ambiguity.

In C++20, we can solve this with concepts. But before C++20, there was a need to add restrictions to the template arguments in order to allow proper resolution of the desired method.

We will solve these two print methods issues later.

Let's take a look at another example that requires SFINAE.

Suppose that we want to use an associative container to map between a key and a value in some generic algorithm. We often prefer to use unordered_map, but we are not sure whether the key provided has a hash function. Some keys implement hash function while others don't. Hence, we want to create our own type, let's call it associative_map which would be an **unordered_map** if the key has implemented a hash function or a **map** if not, in which case the key needs to have an operator <.

To achieve that let's meet another old language tool: **template specialization**.

Template Specialization allows us to declare a template class (called the "base template", not related to inheritance) and then add a specialized version for a more specific case.

In our example, we are going to use a "base template" for the generic case:

```
template <typename Key, typename Value, typename = void>
struct associative_map: std::map<Key, Value> {
  // inheriting all public constructors from base
  using std::map<Key, Value>::map;
};
```

Note that the 3rd template argument: typename = void -- is a bit strange -- it doesn't have a name, it is not used (which goes together with not having a name) and it has a default value 'void'. The user is not actually going to send it. It is there for the specialization that will come soon.

Anyway, the above implementation allows the creation of an associative_map which at the moment is just an alias for, well, std::map.

But in case there is a method std::hash<Key>() we want our associative_map to be an unordered_map! To achieve that we would add the **specialized version**:

```
template <typename Key, typename Value>
struct associative_map<Key, Value,
std::void_t<decltype(std::hash<Key>())>> : std::unordered_map<Key,
Value> {
  using std::unordered_map<Key, Value>::unordered_map;
};
```

The specialized version of the template class associative_map has only two template arguments: Key and Value, in order to complete the missing third template argument required by the base template, we use std::void_t<decltype(std::hash<Key>())> which doesn't always exist.

If it exists then the specialized version is picked automatically. If not, it is not an error -- it is SFINAE, and the base template is used.

The use of the keyword decltype (added in C++11) is for getting in the specialized version the type of std::hash<Key> which may not exist. If it doesn't exist, the template fails substitution but without an error (SFINAE) and we get back to the base template in this case.

A word about the use of std::void_t in the above example. When we use associative_map<Key, Value> the third template parameter is first being completed from the base template as if we sent void. Then when the specialized version is being considered if the third parameter is not void, it is not a match and the specialized version is ignored. Thus we wish to "cast" the value returned by decltype(std::hash<Key>()) to void, using std::void_t.

std::enable_if

There are cases where we want to enable a template only if some compile time expression is true. For example only if T is an integer.
The expression enable_if was first born in boost and then added into the standard library in C++11. Here is a usage example for our print example:

```
template<class T>
typename std::enable_if<std::is_integral<T>::value>::type
print(const T& t) {
    std::cout << "integer: " << t << std::endl;
}

template<class T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(const T& t) {
    std::cout << "floating: " << t << std::endl;
}
```

As you can see we have two very similar methods, but they can compile together without ambiguity due to SFINAE and use of std::enable_if. We have used the SFINAE as the return type in this example, which is either void or invalid. The language requires adding the keyword typename before a dependent type, which makes the type declaration for the return value a bit long:

```
typename std::enable_if<std::is_integral<T>::value>::type
```

and

```
typename std::enable_if<std::is_floating_point<T>::value>::type
```

The expression std::enable_if is a template class that gets two template parameters, the first one is a boolean expression that must be evaluated in compile time, the second parameter is by default void but can be any other type:

```
template< bool B, class T = void >
struct enable_if;
```

If the boolean expression provided is evaluated to true, then enable_if::type is evaluated to T (with T being void if not provided).
However, when the boolean expression provided is evaluated to false, then enable_if::type is not defined, which results in substitution failure, useful for SFINAE.

In our case:

```
    std::is_integral<T>::value
```
is true if T is an integral type and false otherwise.

```
    std::is_floating_point<T>::value
```
is true if T is a floating point type and false otherwise.

Thus for example,

```
    typename std::enable_if<std::is_floating_point<T>::value>::type
```
is a valid type -- void -- only if T is a floating point type.

Under the hood of std::enable_if

The implementation of std::enable_if is quite straightforward and based on template specialization.

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
     using type = T;
};
```

If the boolean expression provided is true the specialized version of std::enable_if is picked, which defines type. In case the boolean expression is false we fall back to the base template which doesn't define the field type.

There is an option to waive the actual implementation of the base template and keep it incomplete:

```
template<bool B, class T = void>
struct enable_if;
```

The result would be the same, albeit with a less friendly compilation error when there is no match. Suppose that we remove the print method for integral types and still call the method with an int. With actual struct body the error message would be something like:

```
template argument deduction/substitution failed:
In substitution of:
'template<class T> typename
enable_if<std::is_floating_point<T>::value>::type print(const T&)
[with T = int]'
error: no type named 'type' in 'struct enable_if<false, void>'
```

With an empty body for the base template the error message would be something like:

```
template argument deduction/substitution failed:
In substitution of 'template<class T> typename
enable_if<std::is_floating_point<_Tp>::value>::type print(const T&)
[with T = int]':
error: invalid use of incomplete type 'struct enable_if<false, void>'
```

The first error message is a bit more verbose.

std::enable_if_t

C++14 added the following expression:

```
template<bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

C++17 added the following expression:

```
template<class T>
inline constexpr bool is_floating_point_v =
is_floating_point<T>::value;
```

And the same was added also for is_integral.

These expressions are mainly syntactic sugaring to avoid the need for adding ::type or ::value after enable_if and is_floating_point respectively, so for example the following code:

```
template<class T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(const T& t) {
    std::cout << "floating: " << t << std::endl;
}
```

Can be replaced with:

```
template<class T>
typename std::enable_if_t<std::is_floating_point_v<T>>
print(const T& t) {
    std::cout << "floating: " << t << std::endl;
}
```

std::is_floating_point

The implementation of std::is_floating_point is also quite simple and based on template specialization.

A first thought might be:

```
template<class T>
struct is_floating_point {
  static constexpr bool value = false;
};

template<>
struct is_floating_point<double> {
  static constexpr bool value = true;
};

template<>
struct is_floating_point<float> {
  static constexpr bool value = true;
};

// and the same for `long double'
```

Above works fine, but there is a shorter way.

```
template< class T >
struct is_floating_point
     : std::integral_constant<
         bool,
         std::is_same<float, typename std::remove_cv<T>::type>::value
||
         std::is_same<double, typename
std::remove_cv<T>::type>::value  ||
         std::is_same<long double, typename
std::remove_cv<T>::type>::value
     > {};
```

See: https://en.cppreference.com/w/cpp/types/is_floating_point

Above is using other building blocks of the language:

```
std::integral_constant<bool, true>
std::integral_constant<bool, false>
```

As well as:

```
std::is_same
```

I leave the implementation of std::is_same as an exercise for the reader.

std::common_type

Another tool that relates to templates is std::common_type which enables implementing a method with an unknown return type:

```
template<typename T, typename... Ts>
std::common_type_t<T, Ts...> sum(T t1, Ts... ts) {
  return t1 + sum(ts...);
}

template<typename T>
T sum(T t1) {
  return t1;
}
```

std::common_type_t<T, Ts...> returns the common type for the template arguments if there exists one. For example the common type for int, bool and long is: long.

C++14 allows auto return type, so we could replace the usage of std::common_type with the following:

```
template<typename T, typename... Ts>
auto sum(T t1, Ts... ts) {
  return t1 + sum(ts...);
}
```

However the two are not the same, as the common type for char and char is obviously char while the return type of summing up two chars with auto return type is int! In case one prefers to preserve the common_type and not lift it up according to the operation performed, std::common_type would be the right tool.

And again we want to understand whether there is some kind of magic in std::common_type or it can be implemented by using the C++ language without any hidden tricks.

The idea behind the implementation of std::common_type is the ternary operator (shortened if expression: ?:). The language defines the return type of the ternary operator to be the common type between the true return type and the false return type. So for example the type of this expression:

```
test_expression ? 3 : 2.5;
```

is double.

The implementation of common_type can use the ternary operator in the following manner:

```
template<typename T, typename... Ts>
struct common_type {
  using type = decltype( false ?
T() : typename common_type<Ts...>::type() );
};

template<typename T>
struct common_type<T> {
  using type = T;
};
```

Note that the 'false' above is just an arbitrary choice. It would work the same with:

```
  using type = decltype( true ?
T() : typename common_type<Ts...>::type() );
```

We assume in the above implementation that all involved types have default constructor, to avoid this assumption we can use std::declval, discussed in the next section:
template<typename T, typename... Ts>
struct common_type;

```
template<typename... Ts>
using common_type_t = typename common_type<Ts...>::type;

template<typename T, typename... Ts>
struct common_type {
  using type = decltype( false ?
std::declval<T>() : std::declval<common_type_t<Ts...>>() );
};
```

The aforementioned example ignores some of the details of std::common_type. For a complete implementation suggestion, take a look at the cppreference:
https://en.cppreference.com/w/cpp/types/common_type#Possible_implementation

std::declval

The keyword decltype is indeed a language magic that we cannot implement on our own. It allows getting the type of an expressions at compile time, which is useful in many meta programming tasks as seen earlier.

However, std::declval on the other hand, is a library feature. The use of std::declval is for declaring types without the need to actually create them.

For example, let's assume that we want to get the return type of the method foo that gets as a parameter an object of type Bar. The expression for getting this type would be:

```
decltype(foo(Bar()))
```

However, if Bar doesn't have an empty constructor the expression, this will not compile. Even though we actually don't really need an object of type Bar, we just wish to "imagine" a call to foo with a parameter of type Bar.

To allow such an expression without dealing with how Bar can be created std::declval was invented, and can be used in our case like this:

```
decltype(foo(std::declval<Bar>()))
```

Now we do not assume anything about how Bar can be instantiated, it can even be an abstract class!

How the trick of std::declval works?

Well it is quite simple. It's just a declaration of a method without implementation, however the declaration declares the return value and thus making it usable for decltype:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

Note that if we would just declare the return value as T, like this:

```
template<class T>
T declval() noexcept;
```

We wouldn't be able to use incomplete types or abstract types (without adding & when calling declval). The addition of std::add_rvalue_reference allows all possible usages, including incomplete types, abstract classes, rvalue reference and lvalue reference.

For example, if the method foo that we want to investigate with decltype gets lvalue reference:

```
int foo(Bar&);
```

we can still use declval very simply based on reference collapsing rules:

```
decltype(foo(std::declval<Bar&>()))
```

You can read more on that here:

https://stackoverflow.com/questions/20303250/is-there-a-reason-declval-returns-add-rvalue-reference-instead-of-add-lvalue-ref

Please note that we have covered just a few parts of the standard library. But the idea is clear. We can clearly say that C++ “is eating its own dog food” in the sense that there is no magic in the library, since things are based on the language syntax itself.

One last hack from Incredibuild to get your juices flowing: Reducing C++ build times

Despite the massive technological advancements, developers are still struggling with slow C++ builds.

This comprehensive guide lays down **EVERYTHING you need to know to reduce your C++ build times.**

Among the various hacks and solutions to unlock faster C++ build times presented in this guide, you can find:

- Utilizing Precompiled Headers (PCH)
- Reducing Dependencies
- Implementing Dynamic Linking / Shared Libraries

Also, you can utilize a distributed processing technology to reduce your C++ build times:

Since Incredibuild was created to specifically solve C++ long compilation time, it is no wonder that we are very acquainted with the various possibilities out there to reduce compilation time. We examined each and every one of them before coming to the conclusion that we have to create our own solution in order to really get what we're looking for. Eventually, all of these solutions, however amazing they may be, are limited in their ability to reduce compilation time. It's not their fault, of course. They have a 'glass ceiling': the CPU/RAM/disk operations they use. We were looking for a more substantial effect on compilation time, with less effort.

That's why our solution exploits CPU power from other computers, to obtain a real boost and reduce compilation time. And when we say reduce the compilation time, we don't mean a small fraction of that time, or even a half (which can still leave you with quite a bit of waiting time on your hand). We mean potentially reducing 90% of that time, making compilation time a non-issue.

So how do we do that? Using our unique distributed compilation technology, we enable users to utilize the idle CPU cycles of other machines in their network, thus transforming each local machine or build server into a virtual supercomputer with dozens of cores. While the solutions mentioned in this guide work within the boundaries of the machine running the build, we are constantly breaking these boundaries, arming the local machine with superpowers taken from other machines.

Sounds a bit wild? It really is.
To learn more visit our website or download our free version.