C++ 解体新書

隠れた機能を探る

著者: Amir Kirsh

INCREDIBUILD

目次

長年にわたって C++ は「言語への追加」と「標準ライブラリへの追加」という2つの開発を軸に進化してきました。

「言語への追加」には、rvalue リファレンス、ラムダ式、override や final などのコンテキスト キーワード、範囲 for、自動変数宣言、decltype、可変引数テンプレート、コンセプト (C++20) などがあります。

このように言語に新たな機能が追加された場合、コンパイラにもそれを解析し対処する性能が必要になります。

一方の「標準ライブラリへの追加」には、スマートポインタ、declval、threads、アトミック変数、std::optional、std::any などがあります。

自社のドッグフードを食べる

標準ドキュメントでは [library] タグ下にライブラリ部品独自のセクションが作成されます (参照: http://eel.is/c%2B%2Bdraft/library)。

ほぼすべてのライブラリ部品が C++ 言語をベースに作られており、そこに裏技や魔法のようなものはありません。どのライブラリも C++ コアに依存しているため、これを「自社のドッグフードを食べる (単に「ドッグフード」とも)」と言ったりもします (https://en.wikipedia.org/wiki/Eating_your_own_dog_food#Origin_of_the_term)。

規格そのものはライブラリ部品の実装方法については言及していないものの、並列処理の要件など、APIや動作の統一方法の要件については言及しています。例えば、連想コンテナでは、insert と emplace メンバはイテレータとコンテナのリファレンスの有効性には影響を与えず、erase メンバは消去された要素へのイテレータとリファレンスのみ無効にする、といった具合です (参照：http://eel.is/c++draft/associative.reqmts#9)。

この記事では C++ のライブラリ部品の一部を取り上げ、それらをどのように実装するかを紹介し、これらの機能の中に何が隠されているのか、カスタマイズして実装することはできるのか、を確認していきます。なお、カスタマイズしたバージョンを実装するのは、どのように実装されているかを理解するためで、標準ライブラリの一部を置き換えることを推奨するものではありません。

この記事では次の機能について説明します。

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

std::any とは？

JavaScript ではほかのスクリプト言語と同じように次のように記述することができます。

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

上の例で変数「a」は、ある特定の型 (JavaScript では number 型) として定義されますが、これは変化します。

C++ でも同じことができる?

「auto を使えばできる」と考えているのであれば、残念ながらそれは不正解です。

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

とはいえ、std::any を使えば C++ でも似たようなことを実現できます。

何でも入れられる型のアイデアは Kevlin Henney 氏が『C++ Report』誌の 2000 年 7-8 月号で紹介したものです (参照: http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf)。

同氏は初め自分の名前をとってこの型を「Henny」と命名しようとしていたようですが、周囲の説得で最終的に「any」にしたという逸話があります。

これが 2001 年には boost::any (https://scicomp.ethz.ch/public/manual/Boost/1.55.0/any.pdf) として、boost に追加され、C++17 で std::any C++ に導入されました。any を使えばこんなことができます。

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

これを見ると変数「a」は実行時にその型を変えているようですが、これは有効な C++ です。というのも、型を変えているようで実際には変えないからです。

では std::any はどのように格納しているのでしょうか? そして型を変えているように見せている内部メンバとは?

std::any の魔法

自社で any を実装するにあたって最初に考えるのは、void* 型のメンバを格納することです。これで any は取得した引数をコピーするために動的な割り当てができるようになります。

しかし、これには問題があります。

any が格納する値は、それが破棄されたとき (デストラクタ) や別の値を取得したとき (代入演算子) に解放されるからです。また、void* を渡して delete することはできません。そのため何とかして型を記憶しておかなければなりません。

テンプレートは使えない?

any がテンプレート化されていれば any<T> のように型を記憶することができるのでは?

これもダメです。any<int> に const char* を割り当てることはできないし、そもそも any 用に型を用意して保持させたいわけではありません。

では、any を特定の型として管理することなく、実際の型を格納しながら解放できるようにするにはどうすれば良いでしょうか?

答えは次の通りです。
a.テンプレート クラスではなく、単純な通常のクラスを保持する
b. any の内部にテンプレートの内部クラス (ここでは「ホルダー」といいます) を保持する (この「ホルダー」のメンバに実際の型を格納し、any の内部に隠す)
c. any にコンストラクタ テンプレートを保持させて any が任意のパラメータを取得できるようにする

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

ここで目指すのは次のような形です。

```
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;
    }
   // ...
};
```

上のコードでも holder * を保持することができないのでやはりダメです。

クラス ホルダーはテンプレート化されているので、テンプレート パラメータを提供する必要がありますが、holder<T> * のように保持することはできません。

これは型 T が any で定義されていないため、any のコンストラクタでのみ認識され、any 自体には認識されないからです。

これを解決するには、仮想デストラクタを持つ非型テンプレートの基底クラス base_holder を追加して、実際に holder<T> を参照する base_holder に対するポインタを any に格納する必要があります。

```
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;
    }
   // ...
};
```

T は any 内部では未知なので「消去」されます。

std::any の使用方法

残念ながら std::any はそれほど簡単に使うことはできません。内部に格納されている値を取り出すには以下の例のように型を示す必要があります。

```
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
```

実際に格納する型に自動でキャストできればとても便利ですが、std::any によって実際の型は「消去」されるので、プログラマーの手で std::any_cast を実行しなければなりません。

std::any に格納された実際の型は、type_info を返す std::any type() メソッドによって実行時に取得できますが、この情報を使用できるのは if-else 文だけで、多重定義には適していません。

使用方法のまとめ：

**「今まで void* を使っていたところを std::any に置き換えます。
つまり、ほぼないということです。」**

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

まとめ１

C++ に導入された新機能は 2 つのグループに分けることができます。1 つ目は rvalue リファレンス、ラムダ式、override や final といった新しい文脈キーワードなど、言語自体への構文追加です。これにはコンパイラの調整が必要です。2 つ目はスマートポインタ、declval、スレッド、アトミック変数など、標準ライブラリへの追加です。このグループはほとんどの場合 C++ 自体がベースになっているという特徴があります。

まとめ２

規格自体はライブラリへの実装方法について言及していないものの、APIや動作の統一方法の要件については言及しています。この記事の目的はいくつかの言語属性を見直し、どのように実装されるかを確認することです。

次に取り上げるのは、std::enable_if、std::common_type、std::declval などの SFINAE で使われるツールです。

そもそも SFINAE とは?

SFINAE とは「**S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror」の頭文字をとったものです。

これ自体は何も新しいものではなく C++98 から存在しているので、どこかで耳にしたことがあるかもしれません。SFINAE をサポートするための新しい要素がいくつか追加されたことで C++11 以降再び注目を集めています。

SFINAE の考え方はシンプルです。「あるテンプレートの展開において、置き換えに失敗した場合はそこでコンパイルエラーとするのではなくテンプレートを『不適合』とし、対象から除外してコンパイルを継続しよう」というものです。

例:

すべての整数 (short、int、long など) に対して次のようなテンプレート メソッドがあるとします。

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

これには問題が 2 つあります。
a.整数だけでなく何でも受け入れてしまう。
b. ostream に抽出演算子を実装していない「T」を取得した場合、つまり、operator<<(std::ostream& out, const T& t) がコンパイル エラーとなる。

では浮動小数点数型のテンプレート メソッドを例に考えてみましょう。

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

print メソッドを呼び出せるようにしたいのですが、実はこれは 1 つのメソッドではなく、2 つのテンプレート メソッドでできています。しかし、両者は全く同じシグネチャを共有しているためかなり不明確です。

C++20 ではコンセプトでこれを解決できますが、それ以前は、目的のメソッドを適切に解決できるようにテンプレートの引数に制限を加えなければなりませんでした。

この問題には後ほど触れます。

では SFINAE が必要となる別の例を見てみましょう。

一般的なアルゴリズムでキーと値の対応付けを行うために連想コンテナを使う場合を考えてみます。unordered_map を使いたいところですが、提供されたキーがハッシュ関数を持つかどうかは不明です。そこで associative_map という独自の型を作成することにします。キーがハッシュ関数を実装していれば **unordered_map**、そうでなければ **map** で、その場合キーは演算子「<」を持ちます。

ここで必要になるのが古くからある言語ツール「**テンプレートの特殊化**」です。

これは、あるテンプレートクラス (「ベーステンプレート」、継承とは無関係) を宣言した上で、より特殊なケースに特化したバージョンを追加することです。

この例では一般的なケースの「ベーステンプレート」を使用することにします。

```
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;
};
```

3 つ目の引数「typename = void」は少し奇妙です。名前を持たず、デフォルト値として「void」を保持しています。これは実際に送信するわけではなく、後々の特殊化のために存在しています。

とにかく、上記を実装することで現時点では std::map のエイリアスに過ぎませんが、associative_map を作成できます。

ここで std::hash<Key>() というメソッドがある場合、associative_map を unordered_map にしたいので、そのために**特殊化したバージョン**を追加します。

```
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;
};
```

特殊化したテンプレートクラス associative_map が持つテンプレート引数は「Key」と「Value」の 2 つだけです。Key と Value はベーステンプレートが要求する第 3 の引数を補完するために、std::void_t<decltype(std::hash<Key>())> を使っていますが、これは必ずしも存在するわけではありません。

存在する場合は、特殊化したバージョンが自動的に選択されますが、そうでない場合でもエラーとはならず (SFINAE)、ベーステンプレートが使用されます。

C++11 で追加された decltype は std::hash<Key> の型 (存在するかどうかは不明) を特殊化したバージョンで取得するのに使われます。存在しない場合、テンプレートは置換に失敗しますが、エラーは発生せず (SFINAE)、ベーステンプレートに戻ります。

上の例で std::void_t を使用する場合、associative_map<Key, Value> を使うと、あたかも void を送ったかのように、3 つ目のテンプレート パラメータから順にベーステンプレートが補完されます。次に特殊化したテンプレートを検討する際に、3 つ目のパラメータが void でなければ無視します。そこで std::void_t を使って decltype(std::hash<Key>()) が返す値を void に「キャスト」する方法を考えてみます。

std::enable_if

コンパイル時条件式が true の場合のみテンプレートを有効にしたい場合があります。例えば「T が整数の場合のみ」といった具合です。
enable_if 式は元々 boost で使われており、C++11 から標準ライブラリに追加されました。以下は、print での使用例です。

```
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;
}
```

このように、2 つのメソッドはとても似ていますが、SFINAE と std::enable_if を使うことで曖昧さを取り除いて同時にコンパイルすることができます。この例では、戻り値の型として SFINAE を使用していますが、これは void か invalid のどちらかになります。C++ では依存型の前に typename というキーワードを追加するので、戻り値の型宣言が少し長くなります。

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

および

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

std::enable_if 式は、次の 2 つのテンプレート パラメータを取得するテンプレートクラスです。最初のパラメータはコンパイル時に評価されるブーリアン型、2 番目のパラメータのデフォルトは void ですが、ほかの型でも構いません。

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

与えられた boolean 値が true の場合、enable_if::type は T と評価されますが (提供されない場合 T は void)、false の場合は、enable_if::type が定義されていないため置き換えに失敗します。そのため SFINAE が有効です。

このケースでは

```
std::is_integral<T>::value
```

T が整数型なら true、そうでなければ false となります。

```
std::is_floating_point<T>::value
```

T が浮動小数点型なら true、そうでなければ false となります。

例:

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

これは T が浮動小数点型のときのみ有効な型となります。

std::enable_if の裏ワザ

std::enable_if はテンプレートの特殊化をベースとしており、とても簡単に実装できます。

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

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

与えられた boolean 値が true なら、std::enable_if の特殊化したバージョンが選択されて型が定義され、false ならフィールドタイプを定義していないベーステンプレートに戻ります。

ベーステンプレートを実際に実装せずに、不完全なままにしておくこともできます。

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

結果は同じですが、一致するものがない場合のコンパイル エラーは多少不親切になります。整数型に対する print メソッドを削除し、int 型でメソッドを呼び出す場合を考えてみます。実際の structbody では、次のようなエラーメッセージが表示されます。

```
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>'
```

ベーステンプレートのボディが空の場合は次のようなエラーメッセージが表示されます。

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

1 つ目のエラーメッセージの方が少し詳しいのが分かります。

std::enable_if_t

C++14 では次の式が追加されました。

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

C++17 では次の式が追加されました。

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

また、is_integral にも同様の追加がありました。

これらの表現は、enable_if と is_floating_point の後にそれぞれ ::type と ::value を追加しなくてもよい糖衣構文で、例えば次のようなコードになります。

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

これは次のように置き換えることができます。

```
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

std::is_floating_point もテンプレートの特殊化をベースとしており、実装するのは簡単です。

まずは次のように記述してみます。

```
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'
```

上記のコードでも問題なく動作しますが、もっと簡単に記述できます。

```
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
     > {};
```

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

上記はほかの言語の構成要素を使用しています。

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

次も同様です。

```
std::is_same
```

std::is_same の実装は練習問題として残しておきましょう。

std::common_type

テンプレートに関連するもうひとつのツールは std::common_type で、戻り値の型が不明なメソッドを実装できます。

```
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...> はテンプレート引数に共通の型があればそれを返します。例えば、int、bool、long の共通型は long です。

C++14 では戻り値型に auto を使えるので、std::common_type を次のように使うことができます。

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

ただし、この2つは同じではありません。char と char の共通型は当然 char ですが、char の合計を auto で返す場合は int になります。common_type を保持したいけど、実行された操作に応じリスト化したくない場合は、std::common_type が最適といえます。

ここでもやはり、std::common_type に何かしらの裏技があるのか、それとも C++ 言語を使えば難なく実装できるのかを理解したいのです。

std::common_type の背景にあるのは三項演算子です。三項演算子の戻り値の型は true と false の戻り値の型の共通型と定義されています。なので、例えば次の式では型は「double」となります。

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

common_type の実装では、以下の方法で三項演算子を使用することができます。

```
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;
};
```

なお、上記の「false」はあくまで任意の選択です。これは次と同じように動作します。

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

上記の実装では、関係するすべての型がデフォルト コンストラクタを持つことが前提ですが、この前提が崩れる場合でも次のセクションで説明する std::declval を使用することが可能です。

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...>>() );
};
```

前述の例では、std::common_type の詳細をいくつか省略しています。完全な実装方法については cppreference (https://en.cppreference.com/w/cpp/types/common_type#Possible_implementation) を参照してください。

std::declval

decltype はまさに言語の魔法というべきものです。これは、コンパイル時条件式で型を取得でき、先に見たように多くのメタプログラミングのタスクで役立ちますが、ライブラリの機能です。

std::declval は実際に型を作成せず、型を宣言するために使います。

例えば、Bar 型のオブジェクトをパラメータとして取得するメソッド foo の戻り値の型を取得する場合を考えてみましょう。この型を取得するための式は次のようになります。

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

ただし、Bar が空のコンストラクタを持たない式ではコンパイルされません。実際には Bar 型のオブジェクトはいらないけど、Bar 型のパラメータを持つ foo の呼び出しを「想像」したいだけです。

Bar の作成を気にすることなく、このような式を実現したい場合は次のように std::declval を使うことができます。

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

ここでは Bar がどのようにインスタンス化されるかについては特に想定していません。

std::declval のしくみ

std::declval のしくみはとてもシンプルです。これは、実装を伴わない単なるメソッドの宣言ですが、宣言によって戻り値が宣言されているため、次のように decltype で使用することができます。

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

なお、戻り値を T と宣言するだけなら次のようになります。

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

不完全な型や抽象型 (& なしで declval を呼び出す) を使うことはできませんが、std::add_rvalue_reference を追加することで、不完全な型、抽象クラス、rvalue リファレンス、lvalue リファレンスなど考え得るすべてで使えるようになります。

例えば、decltype で調べたいメソッド foo が lvalue リファレンスを取得する場合は次のようになります。

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

このように、参照の縮約ルールに基づいて、とても簡単に declval を使用することができます。

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

詳細は次のリンクをご覧ください。

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

今回取り上げたのは標準ライブラリのほんの一部ですが、その考え方は明確です。C++ は「ドッグフード」とはっきり言うことができます。それは言語の構文そのものをベースにしているため、ライブラリに魔法はないということです。

Incredibuild からの最後のヒントは C++ ビルド時間を短縮して力を引き出す方法です。

技術が大きく進歩したにもかかわらず、開発者は遅い C++ ビルドに悩まされ続けています。

この包括的なガイドでは C++ のビルド時間を短縮するためのすべてを紹介しています。

例えば次のようなものがあります。

- プリコンパイル ヘッダ（PCH）の活用
- 依存関係の低減
- ダイナミックリンク / 共有ライブラリの実装

また、分散処理技術を活用することで C++ のビルド時間を短縮できます。

Incredibuild は主に C++ の長時間におよぶコンパイルを解決するために生まれました。どうすればコンパイル時間を短縮できるかを突き詰めた結果「自分たちの求めるものを実現するためには、自分たちで解決しなければならない」という結論に至りました。ここで紹介したソリューションは、どんなに素晴らしくてもコンパイル時間を短縮するには限界があります。もちろん、誰が悪いという訳ではありません。ただこれらのソリューションには使用する CPU/メモリ/ディスクといった「ガラスの天井」があるのです。Incredibuild ではより少ない労力で、コンパイル時間を実質的により短縮できる方法を探していました。

そして、ほかのコンピューターの CPU を利用することで、実質的なブーストとコンパイル時間の短縮を実現するソリューションを生み出しました。コンパイル時間の短縮といっても、ほんの少しや半分程度ということはなく、90%もの時間短縮を実現します。

そのために独自の分散コンパイル技術を使ってネットワーク上のほかのマシンの CPU サイクルを利用し、ローカルマシンやビルドサーバーを数十コアの仮想スーパーコンピューターに変身させるのです。このガイドで紹介するソリューションは、ビルドを実行しているマシンの境界内で動作しますが、Incredibuild は常にこの境界を超え、ほかのマシンからパワーを集めてローカルマシンを武装しています。

信じられないかも知れませんが、本当です。
詳細は当社のウェブサイトをご覧いただくか、無料版をダウンロードしてください。