HOW TO USE VOCABULARY TYPES FROM C++17?

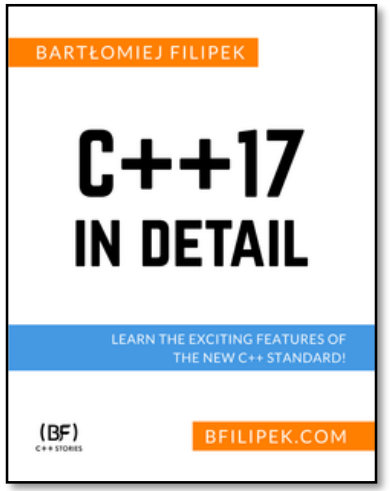
`std::optional`, `std::variant`, `std::any`

17th September 2019

Bartłomiej Filipek, bfilipek.com

About me

- See my coding blog at: www.bfilipek.com
- ~12y coding experience
- Microsoft MVP
- C++ ISO Member
- @Xara.com
  - Text related features for advanced document editors
- Somehow addicted to C++ ☺



C++17 In Detail

Xara Cloud Demo

Microsoft® Most Valuable Professional

cpp-polska.pl
BLOG PROGRAMISTYCZNY

The plan

- `std::optional`
- `std::variant`
- `std::any`
- Summary
- Extras!

std::optional - creation

```
// empty:
std::optional<int> oEmpty;
std::optional<float> oFloat = std::nullopt;

// direct:
std::optional<int> oInt(10);
std::optional oIntDeduced(10); // deduction guides

// make_optional
auto oDouble = std::make_optional(3.0);
auto oComplex = std::make_optional<std::complex<double>>(3.0, 4.0);

// in_place
std::optional<std::complex<double>> o7{std::in_place, 3.0, 4.0};
std::optional<std::vector<int>> oVec(std::in_place, {1, 2, 3}); // will call vector with direct init of {1, 2, 3}

// copy from other optional:
auto oIntCopy = oInt;
```

std::optional – accessing the value

```
// by operator* (or ->)
std::optional<int> oint = 10;
std::cout<< "oint " << *oint << '\n'; // UB if no value!

If (oint.has_value()) { } // simple check!

// by value()
std::optional<std::string> ostr("hello");
try {
   std::cout << "ostr " << ostr.value() << '\n';
}
catch (const std::bad_optional_access& e) {
   std::cout << e.what() << '\n';
}

// by value_or()
std::optional<double> odouble; // empty
std::cout<< "odouble " << odouble.value_or(10.0) << '\n';
```

std::optional – performance & cost

```
struct Range {
   std::optional<double> mMin;
   std::optional<double> mMax;
};

struct RangeCustom {
   bool mMinAvailable;
   bool mMaxAvailable;
   double mMin;
   double mMax;
};
```

32 bytes vs 24 bytes

```
template <typename T>
class optional {
   bool _initialized;
   std::aligned_storage_t<sizeof(t), alignof(T)> _storage;
public: // operations
};
```

```
std::optional<double> od;                  // sizeof = 16 bytes
std::optional<int> oi;                     // sizeof = 8 bytes
std::optional<std::array<int, 10>> oa;     // sizeof = 44 bytes
```

std::optional examples

- Constructing a Query to a Database
- Conversion from a String to an Integer
- Conversion from String, More Generic solution
- Monadic Extensions
- Geometry and Intersections
- Simple optional chaining
- Handling a throwing constructor
- Getting File contents
- Haskell's listToMaybe
- Cleaner interface for map.find
- Configuration of a Nuclear Simulation
- Factory
- Lazy Loading on the stack
- …

https://www.bfilipek.com/2018/06/optional-examples-wall.html

std::variant

- **C.183: Don’t use a union for type punning**:
  - It is undefined behaviour to read a union member with a different type from the one with which it was written. Such punning is invisible, or at least harder to spot than using a named cast. Type punning using a union is a source of errors.
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c183-dont-use-a-union-for-type-punning

std::variant - creation

```
// default initialisation: (the first type has to have a default ctor)
std::variant<int, float> intFloat;
std::cout << intFloat.index() << ", val: " << std::get<int>(intFloat) << '\n';

// std::variant<NotSimple, int> cannotInit; // error
std::variant<std::monostate, NotSimple, int> okInit;

// pass a value:
std::variant<int, float, std::string> intFloatString { 10.5f };

// in_place for complex types
std::variant<std::vector<int>, std::string> vecStr {
   std::in_place_index<0>, { 0, 1, 2, 3 }
};

std::variant<std::string, int, bool> vStrIntBool = "Hello World";
```

std::variant – changing the value

```
std::variant<int, float, std::string> intFloatString { "Hello" };
intFloatString = 10; // we're now an int
intFloatString.emplace<2>(std::string("Hello")); // we're now string again

// std::get returns a reference, so you can change the value:
std::get<std::string>(intFloatString) += std::string(" World");
intFloatString = 10.1f;

if (auto pFloat = std::get_if<float>(&intFloatString); pFloat)
    *pFloat *= 2.0f;
```

std::variant – accessing the value

```
std::variant<int, float, std::string> intFloatString;
try {
    auto f = std::get<float>(intFloatString);
    std::cout << "float! " << f << '\n';
}
catch (std::bad_variant_access&) {
    std::cout << "our variant doesn't hold float at this moment...\n";
}


if (const auto intPtr = std::get_if<0>(&intFloatString))
    std::cout << "int!" << *intPtr << '\n';
```

std::variant - visitors

```
// a generic lambda:
auto PrintVisitor = [](const auto& t) { std::cout << t << '\n'; };

std::variant<int, float, std::string> intFloatString { "Hello" };
std::visit(PrintVisitor, intFloatString);
```

```
struct MultiplyVisitor {
   float mFactor;
   MultiplyVisitor(float factor) : mFactor(factor) { }

   void operator()(int& i) const { i *= static_cast<int>(mFactor); }
   void operator()(float& f) const { f *= mFactor; }
   void operator()(std::string& ) const { // nothing to do here...}
};

std::visit(MultiplyVisitor(2.5f), intFloatString );
std::visit(PrintVisitor, intFloatString );
```

std::variant - visitors

```
std::variant<int, float, std::string> myVariant;

std::visit(
overload {
    [](const int& i) { std::cout << "int: " << i; },
    [](const std::string& s) { std::cout << "string: " << s; },
    [](const float& f) { std::cout << "float: " << f; }
   },
   myVariant
);
```

Overload

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

- overload uses three C++17 features:
  - Pack expansions in using declarations - short and compact syntax with variadic templates.
  - Custom template argument deduction rules - this allows the compiler to deduce types of lambdas that are the base classes for the pattern. Without it, we'd have to define a "make" function.
  - Extension to aggregate Initialisation - the overload pattern uses aggregate initialization to init base classes. Before C++17, it was not possible.

std::variant performance

- No dynamic memory allocation
- Extra bits for storing the current type info
  - On GCC 8.1, 32 bit:
    - sizeof string: 32
    - sizeof variant<int, string>: 40
    - sizeof variant<int, float>: 8
    - sizeof variant<int, double>: 16

std::variant examples

- Parsing files
- Config files
- Finite state machines -> Space Game: A std::variant-Based State Machine by Example
- Polymorphism
- Error handling
- … ?

- https://www.bfilipek.com/2018/06/variant.html

std::any

```
std::any a(12);

// set any value:
a = std::string("Hello!");
a = 16;

// reading a value:
// we can read it as int

std::cout << std::any_cast<int>(a) << '\n';
// but not as string:
try {
   std::cout << std::any_cast<std::string>(a) << '\n';
}
catch(const std::bad_any_cast& e) {
   std::cerr<< e.what() << '\n';
}
```

std::any - creation

```
// default initialisation:
std::any a;
assert(!a.has_value());

// initialisation with an object:
std::any a2{10}; // int
std::any a3{MyType{10, 11}};

// in_place:
std::any a4{std::in_place_type<MyType>, 10, 11};
std::any a5{std::in_place_type<std::string>, "Hello World"};

// make_any
std::any a6 = std::make_any<std::string>{"Hello World"};
```

std::any

```
// you can use it in a container:
std::map<std::string, std::any> m;
m["integer"] = 10;
m["string"] = std::string("Hello World");
m["float"] = 1.0f;

for (auto &[key, val] : m) {
    if (val.type() == typeid(int))
        std::cout << "int: " << std::any_cast<int>(val) << '\n';
    else if (val.type() == typeid(std::string))
        std::cout << "string: " << std::any_cast<std::string>(val) << '\n';
    else if (val.type() == typeid(float))
        std::cout << "float: " << std::any_cast<float>(val) << '\n';
}
```

std::any – accessing the value

```
std::any var = 10;

// read access:
auto a = std::any_cast<int>(var);

// read/write access through a reference:
std::any_cast<int&>(var) = 11;

// read/write through a pointer:
int* ptr = std::any_cast<int>(&var);
*ptr = 12;
```

might throw `std::bad_any_cast`

might return `nullptr`

std::any – size and performance

23.8.3 [any.class]:

Implementations should avoid the use of dynamically allocated memory for a small contained value. Example: where the object constructed is holding only an int. Such small object optimisation shall only be applied to types T for which is_nothrow_move_constructible_v<T> is true.

Compiler	Sizeof(std::any)
GCC 8.1 (Coliru) 16	16
Clang 7.0.0 (Wandbox)	32
MSVC 2017 15.7.0 32-bit	40
MSVC 2017 15.7.0 64-bit	64

std::any - examples

- Useful when passing between layers/systems/libraries
  - *„The general gist is that std::any allows passing ownership of arbitrary values across boundaries that don't know about those types."*

```
struct property {
   property();
   property(const std::string &, const std::any &);
   std::string name;
   std::any value;
};

typedef std::vector<property> properties;
```

Summary

- Std::optional
  - Adds „null” state to your type,
  - no dynamic allocation
  - .has_value(), value(), operator*
- Std::variant
  - Typesafe union,
  - no dynamic allocations
  - Std::get<>, std::visit
- Std::any
  - Typesafe void*,
  - Might perfom dynamic allocation
  - std::any_cast<>

Quiz Time

- Why is std::optional better than unique_ptr? (for storing nullable types?)
  - Unique_ptr also nullable, but requires dynamic memory allocation, cannot copy easily

Quiz Time

- What's the sizeof of std::variant?
  - Depends ☺
  - In general, it's the max size of all variant types + place for the discriminator value

Quiz Time

- What's std::monostate?
  - Default constructible, empty object
  - Helpful when your types in variant cannot be default constructible
  - Variant<monostate, TypeA, TypeB> var;

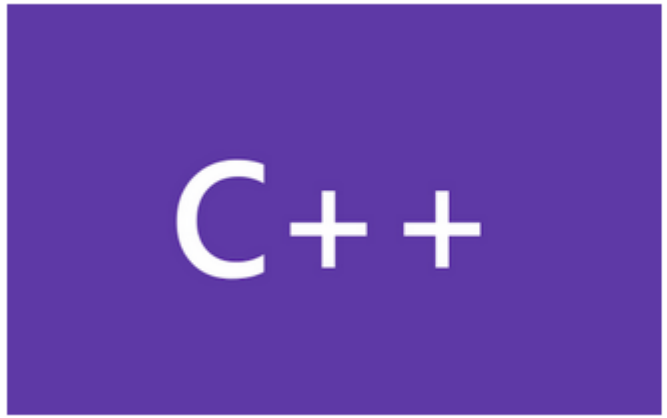
Quiz Time

- Can std::variant get into an invalid state?
  - When old value is destroyed and then during the initialisation of the new value we throw exception
  - See example: http://coliru.stacked-crooked.com/a/413ce70317dbd2e5

Quiz Time

- Can you use std::optional with T* (pointer) or bool?
  - Yes, but it's confusing

```
std::optional<int*> opi{ new int(10) };
if (opi && *opi)
{
    std::cout << **opi;
    delete *opi;
}
if (opi)
    std::cout << "opi is still not empty!";
```



Open Sourcing MSVC's STL

Stephan T. Lavavej - MSFT September 16, 2019

Today at CppCon 2019, we (the MSVC team) announced that we're releasing our implementation of the C++ Standard Library (also known as the STL) as open source. https://github.com/microsoft/STL is our new repository, containing all of our product source code, a new CMake build system,

https://devblogs.microsoft.com/cppblog/open-sourcing-msvcs-stl/


```
#include <concepts>
```

C++20 Concepts Are Here in Visual Studio 2019 version 16.3

xiangfan September 10, 2019

C++20 Concepts are now available for the first time in Visual Studio 2019 version 16.3 Preview 2. This includes both the compiler and standard library support, but not the intellisense support.

https://devblogs.microsoft.com/cppblog/c20-concepts-are-here-in-visual-studio-2019-version-16-3/

50% off

https://leanpub.com/cpp17indetail/c/cppkrk

https://cpp-polska.pl/

cpp-polska.pl

BLOG PROGRAMISTYCZNY

http://cpp-polska.pl/slack