C++ Utilities Stuff

Written by:

- Ng Zhia Yang
- Thomas Tan
- Georgie Lee

Last updated: 30 June 2022

This lecture will cover some commonly used classes in C++ you'll encounter in the wild (work) and might find helpful in your projects.

1 `std::pair`

We've probably all heard or used pairs before e.g. in storing a pair of coordinates `(x, y)`. It's so common that C++ has such a class, `std::pair`.

1.1 `SimplePair`, a simple `std::pair` class

Before talking about `std::pair`, it's instructive to talk about a naive implementation and compare and contrast `SimplePair` with what `std::pair` gives us:

```cpp
template <typename T1, typename T2>
struct SimplePair {
  T1 first;
  T2 second;
};
```

Snippet 1: `SimplePair` implementation part 1

At its core, this is all a pair is: a struct that helps us store two types. Here, we just use a simple aggregate template struct with template types `T1` and `T2`, representing the types of the first and second members respectively. We can use it as follows:

```cpp
  SimplePair<int, double> simplePair1{1, 2.0};
  std::cout << simplePair1.first << " " << simplePair1.second << "\n";

  SimplePair<std::string, int> simplePair2{"abc", 1};
  std::cout << simplePair2.first << " " << simplePair2.second << "\n";
```

Snippet 2: `SimplePair` usage

However, it is rather clunky to use this pair template. It would be more convenient to simply define our own `struct`s if this was all there was. So `std::pair` gives us many additional utilities.

For example, it is useful to be able to compare between pairs, so let's implement comparison operators:

```cpp
  bool operator<(const SimplePair& other) const {
    return first == other.first ? second < other.second
                                : first < other.first;
  }

  bool operator==(const SimplePair& other) const = default;
  bool operator!=(const SimplePair& other) const = default;
```

Snippet 3: `SimplePair` implementation part 2

Then, we can do:

```cpp
  SimplePair<int, double> simplePair3{1, 1.0};
  SimplePair<int, double> simplePair4{2, 1.0};
  SimplePair<int, double> simplePair5{2, 2.0};
  SimplePair<int, double> simplePair6{2, 2.0};

  assert(simplePair3 < simplePair4);
  assert(simplePair4 < simplePair5);
  assert(simplePair5 == simplePair6);
  assert(simplePair3 != simplePair6);

  SimplePair<SimplePair<int, int>, int> simplePair7{{1, 1}, 1};
  SimplePair<SimplePair<int, int>, int> simplePair8{{1, 2}, 1};
  SimplePair<SimplePair<int, int>, int> simplePair9{{2, 1}, 1};
  SimplePair<SimplePair<int, int>, int> simplePair10{{2, 1}, 4};

  assert(simplePair7 < simplePair8 && simplePair8 < simplePair9 &&
         simplePair9 < simplePair10);

  std::vector<SimplePair<int, int>> vecOfSimplePairs{
      {6, 9}, {4, 3}, {4, 2}, {1, 1}, {1, 1}, {1, 2}};
  // vecOfSimplePairs is {6,9}, {4,3}, {4,2}, {1,1}, {1,1}, {1,2}

  std::sort(vecOfSimplePairs.begin(), vecOfSimplePairs.end());
  // vecOfSimplePairs is {1,1}, {1,1}, {1,2}, {4,2}, {4,3}, {6,9}
```

Snippet 4: `SimplePair` comparison operators

For the most part, `std::pair` can be used as a drop-in replacement for our `SimplePair` above ie.

```cpp
  std::pair<int, double> stdPair1{1, 2.0};
  std::cout << stdPair1.first << " " << stdPair1.second << "\n";

  std::pair<std::string, int> stdPair2{"abc", 1};
  std::cout << stdPair2.first << " " << stdPair2.second << "\n";

  std::pair<int, double> stdPair3{1, 1.0};
  std::pair<int, double> stdPair4{2, 1.0};
  std::pair<int, double> stdPair5{2, 2.0};
  std::pair<int, double> stdPair6{2, 2.0};

  assert(stdPair3 < stdPair4);
  assert(stdPair4 < stdPair5);
  assert(stdPair5 == stdPair6);
  assert(stdPair3 != stdPair6);

  std::pair<std::pair<int, int>, int> stdPair7{{1, 1}, 1};
  std::pair<std::pair<int, int>, int> stdPair8{{1, 2}, 1};
  std::pair<std::pair<int, int>, int> stdPair9{{2, 1}, 1};
  std::pair<std::pair<int, int>, int> stdPair10{{2, 1}, 4};

  assert(stdPair7 < stdPair8 && stdPair8 < stdPair9 &&
         stdPair9 < stdPair10);

  std::vector<std::pair<int, int>> vecOfStdPairs{
      {6, 9}, {4, 3}, {4, 2}, {1, 1}, {1, 1}, {1, 2}};
  // vecOfStdPairs is {6,9}, {4,3}, {4,2}, {1,1}, {1,1}, {1,2}

  std::sort(vecOfStdPairs.begin(), vecOfStdPairs.end());
  // vecOfStdPairs is {1,1}, {1,1}, {1,2}, {4,2}, {4,3}, {6,9}
```

Snippet 5: `std::pair` usage

1.2 Notable differences with `std::pair`

On top of simply implementing comparison operators, `std::pair` does a lot more for us.

1.2.1 Many other constructors in `std::pair`

See https://en.cppreference.com/w/cpp/utility/pair/pair for a full list and explanations.

1.2.2 `std::pair`'s piecewise constructor

```cpp
template< class... Args1, class... Args2 >
pair( std::piecewise_construct_t,
      std::tuple<Args1...> first_args,
      std::tuple<Args2...> second_args );
```

Snippet 6: `std::pair`'s `std::piecewise_construct_t` ctor overload

Use case 1: you want to construct the pair members in-place i.e. at exactly where they'd be located after the pair is fully constructed. This is in contrast to constructing the pair members somewhere else and then moving (move constructing) them into the pair members.

```cpp
  // Constructs the vectors first then move constructs them into the pair
  // members
  std::pair<std::vector<int>, std::vector<double>> pairOfVecs1(
      {1, 2}, {3.69, 6.9, 4.2});

  // Constructs the vectors in place i.e. together with the pair in memory
  std::pair<std::vector<int>, std::vector<double>> pairOfVecs2(
      std::piecewise_construct,
      std::forward_as_tuple(10, 2),  // 10 elements, all set to 2
      std::forward_as_tuple(
          std::initializer_list<double>{3.69, 6.9, 4.2}));

  assert(pairOfVecs2.first.size() == 10);
  assert(pairOfVecs2.second.size() == 3);
```

Snippet 7: `std::pair` piecewise constructor example

Use case 2: Suppose your `std::pair's` `T1` and `T2` are non-copyable and non-movable types e.g. `std::mutex`. Then, you can't create such a `std::pair` using the other constructors which might rely on moving a temporary (`prvalue`) into the pair. Instead, you must use this constructor overload and create the pair members in-place. Legit example below.

```cpp
struct Q {
  int a;
  double b;
  Q(int a, double b) : a(a), b(b) {}
  bool operator==(const Q& other) const = default;
  Q(const Q& other) = delete;
  Q& operator=(const Q& other) = delete;
  Q(Q&& other) = delete;
  Q& operator=(Q&& other) = delete;
};

// Generates a hash value from a Q object like size_t hashVal =
// QHasher{}(q), required for unordered_map to know how to hash your
// custom class / struct. See
// https://en.cppreference.com/w/cpp/utility/hash
struct QHasher {
  size_t operator()(const Q& q) const {
    return std::hash<int>{}(q.a) ^ std::hash<double>{}(q.b);
  }
};

    // in main
    std::unordered_map<Q, std::pair<int, int>, QHasher> mp;
    // mp.emplace(Q{5, 5.0}, std::make_pair(1, 2)); // won't work because
    // Q is not movable
    mp.emplace(std::piecewise_construct,
               std::forward_as_tuple(5, 5.0),
               std::forward_as_tuple(1, 2));
```

Snippet 8: `std::pair` piecewise constructor example

Use case 2.5: Also notice that `unordered_map`'s `emplace(...)` constructs elements in-place anyway e.g. `unordered_map<int, double> mp; mp.emplace(1, 5.0);` through perfect forwarding into the pair stored in the hashtable node. But when the constructor for your key takes in multiple arguments, we must use pair's piecewise constructor to disambiguate.

1.2.2.1 `std::forward_as_tuple(...)`

Constructs a tuple of references to the arguments in args suitable for forwarding as an argument to a function. The tuple has rvalue reference data members when rvalues are used as arguments, and otherwise has lvalue reference data members. See https://en.cppreference.com/w/cpp/utility/tuple/forward_as_tuple.

1.2.3 `std::make_pair`

```cpp
template< class T1, class T2 >
constexpr std::pair<V1,V2> make_pair( T1&& t, T2&& u );
```

Snippet 9: `std::make_pair`

Creates a `std::pair` object, deducing the types of `T1` and `T2` from the argument types. It is (was) convenient to use this function to avoid specifying the template types in `std::pair` all the time (pre C++17, where types of template arguments could not be deduced from constructor, but we can do `std::pair p1(5, "i love c++")` now).

```cpp
  auto ezPair1 = std::make_pair(6, 9);
  auto ezPair2 = std::make_pair(4.2, "hi there");
  auto ezPair3 =
      std::make_pair(new double{3.14}, std::string("yo yo yo"));

  static_assert(std::is_same_v<decltype(ezPair1.first), int>);
  static_assert(std::is_same_v<decltype(ezPair1.second), int>);
  static_assert(std::is_same_v<decltype(ezPair2.first), double>);
  static_assert(std::is_same_v<decltype(ezPair2.second), const char*>);
  static_assert(std::is_same_v<decltype(ezPair3.first), double*>);
  static_assert(std::is_same_v<decltype(ezPair3.second), std::string>);
```

Snippet 10: `std::make_pair` usage

1.3 Some use cases of `std::pair` in the standard library

`std::unordered_map<Key,T,Hash,KeyEqual,Allocator>::emplace` returns `true` if insertion takes place, else `false`.

```cpp
    std::unordered_map<int, std::string> mp;
    {
      auto [it, ok] = mp.emplace(1, "uwu");
      assert(it->second == "uwu");
      assert(ok);
    }
    {
      auto [it, ok] = mp.emplace(1, "owo");
      assert(it->second == "uwu");
      assert(!ok);
    }
```

Snippet 11: `std::pair` usage in `emplace(...)`

2 Structured bindings (since C++17)

Syntax: `cv-auto ref-qualifier(optional) [ identifier-list ] = expression;`

2.1 To non-static data members of structs / classes

You might have noticed this syntax: `auto [it, ok] = mp.emplace(...);` This means that the `.first` member of the `std::pair` returned by `mp.emplace(...)` is assigned to `it` and the `.second` member to `ok`.

This is called structured binding for classes / structs and works like this:

```
struct Player {
  inline static int globalCount = 0;
  int id;
  int health, mana;
  Player(int health, int mana)
      : id(++globalCount), health(health), mana(mana) {}
};

  // in main()
  Player p1(100, 100), p2(200, 50);
  auto [p1Id, p1Health, p1Mana] = p1;
  auto [p2Id, p2Health, p2Mana] = p2;
  assert(p1Id == p1.id && p1Health == p1.health && p1.mana == p1.mana);
  assert(p2Id == p2.id && p2Health == p2.health && p2.mana == p2.mana);
```

Snippet 12: Structured bindings for structs / classes

Each identifier in the structured binding identifier-list (inside the `[...]`) becomes the name of a variable that refers to the next member of the struct on the RHS in declaration order.

2.2 To an array

```
int a[2] = {1,2};

auto [x,y] = a; // creates e[2], copies a into e,
                // then x refers to e[0], y refers to e[1]
auto& [xr, yr] = a; // xr refers to a[0], yr refers to a[1]
```

Snippet 13: Structured binding to an array, from
https://en.cppreference.com/w/cpp/language/structured_binding

2.3 To a tuple-like type

The expression `std::tuple_size<E>::value` must be a well-formed integer constant expression, and the number of identifiers in the identifier-list must equal `std::tuple_size<E>::value`.

```
  std::tuple<int, double, std::string> tpl1(5, 0.5, "uwu");
  auto [tpl1First, tpl1Second, tpl1Third] = tpl1;
  tpl1First = 100;
  assert(std::get<0>(tpl1) == 5 && std::get<1>(tpl1) == 0.5 &&
         std::get<2>(tpl1) == "uwu");

  auto& [lvr_tpl1First, lvr_tpl1Second, lvr_tpl1Third] = tpl1;
  lvr_tpl1First = 100;
  lvr_tpl1Second = 3.50;
  assert(std::get<0>(tpl1) == 100 && std::get<1>(tpl1) == 3.50 &&
         std::get<2>(tpl1) == "uwu");
```

Snippet 14: Structured bindings for tuples

“tuple-like” type means the type has template specializations for `std::tuple_size<...>` and `std::tuple_element<...>` e.g. for `std::tuple` and `std::pair` we see these:

Helper classes

`std::tuple_size``<std::tuple>` (C++11)	obtains the size of `tuple` at compile time (class template specialization)
`std::tuple_element``<std::tuple>` (C++11)	obtains the type of the specified element (class template specialization)

Template specializations for `std::tuple`

Helper classes

`std::tuple_size``<std::tuple>` (C++11)	obtains the size of `tuple` at compile time (class template specialization)
`std::tuple_element``<std::tuple>` (C++11)	obtains the type of the specified element (class template specialization)

Template specializations for `std::pair`

So, you can build your own tuple-like class / type by specializing those templates and structured bindings will work for it, just like how they work for `std::tuple` and `std::pair`. We won’t cover these in this lecture, stay tuned for the lectures on template metaprogramming!

3 `std::tuple`

It's essentially a "generalization" of a pair, to `N` types, where `N` is determined and fixed at compile-time but we get elements by using a non-member function `std::get<...>(...)` rather than via member access.

```
std::tuple<int, int, int> tpl1(1, 2, 3);
std::cout << "(1) "                   //
          << std::get<0>(tpl1) << " "  //
          << std::get<1>(tpl1) << " "  //
          << std::get<2>(tpl1) << "\n";

std::tuple<int, double, std::string> tpl2(1, 2.0, "yolo");
std::cout << "(2) "                        //
          << std::get<int>(tpl2) << " "     //
          << std::get<double>(tpl2) << " "  //
          << std::get<std::string>(tpl2) << "\n";

// Output:
// (1) 1 2 3
// (2) 1 2 yolo
```

Snippet 15: `std::get<...>(...)` in action

▼ Why is `std::get<...>(...)` a free function?

If we have a `get<...>()` member function, we run into the issue of having to write `template` before the member function `get<...>()`, when it's a dependent name ie. the meaning or type of object we're invoking the member function `get<...()` on differs from one instantiation to another (of the class or function template).

```
template <typename T1, typename T2>
class BinaryTuple {
  T1 a;
  T2 b;

public:
  BinaryTuple(T1 a, T2 b) : a(std::move(a)), b(std::move(b)) {}

  template <size_t I>
  auto& get() {
    if constexpr (I == 0) {
      return a;
    } else if constexpr (I == 1) {
      return b;
    } else {
      // can't just do static_assert(flag, "no match");
      []<bool flag = false>() {
        static_assert(flag, "no match");
      }
      ();
    }
  }
};

template <size_t I>
void getNoNeedTemplateKeyword(BinaryTuple<int, std::string>& tpl) {
  std::cout << tpl.get<I>() << "\n";  // OK
  std::cout << tpl.get<0>() << "\n";  // OK
  std::cout << tpl.get<1>() << "\n";  // OK
}

template <size_t I, typename T1, typename T2>
void getNeedTemplateKeyword(BinaryTuple<T1, T2>& tpl) {
  std::cout << tpl.template get<I>() << "\n";  // OK

  // If no 'template' keyword, get this error:
  // main.cpp:36:20: error: missing 'template' keyword prior to dependent
  // template name 'get':
  // in std::cout << tpl.get<I>() << "\n";
  // in std::cout << tpl.get<0>() << "\n"; // even this
  // in std::cout << tpl.get<1>() << "\n"; // and this are not allowed
}
```

Snippet 16: Issue with member function `get<...>()`

Regardless of what template parameter `I` we instantiate `template <size_t I> void getNoNeedTemplateKeyword(BinaryTuple<int, std::string>& tpl)` with, the type of `tpl` is always the same and so we don't need `template` keyword behind member function `get<...>()`. But the type of `tpl` in `template <size_t I, typename T1, typename T2> void getNeedTemplateKeyword(BinaryTuple<T1, T2>& tpl)` will differ across instantiations of `getNeedTemplateKeyword(...)` which have different `T1` and / or `T2` template types. So, we'll need `template` keyword behind member function `get<...>()` ie. `tpl.template get<I>()`.

This is because when the compiler parses the code, it does not know whether `tpl.get` refers to a member attribute of `tpl` or a template member function, since `tpl` is a template and `get` is template dependent (on `tpl`). So, when writing `tpl.get<0>();`, compiler can interpret it as `tpl.get < 0 > ();` ie. `tpl.get` less than `0` greater than `();`, which is invalid C++ code. So, writing `template` in front of `.get` tells the compiler that `.get` is a template member function and to interpret it as such, and not `<` as the less than operator.

4 `std::optional`

Wraps a value which might or might not be present.

Common use case is as the return value of a function which might fail vs returning `std::pair<T, bool>`, which might be less performant for expensive-to-construct objects.

If `optional<T>` contains a value, it is said to be "engaged"; the value is guaranteed to be allocated as part of the optional object footprint ie. no dynamic memory allocation.

`optional<T>` object does not contain a value ("disengaged") if:

- it's default-initialized,
- initialized with / assigned a value of type `std::nullopt_t` or an optional object that does not contain a value, or
- `reset()` is called on it.

```
std::optional<int> o1{5},  // initialized with value
    o2,                    // default initialized
    o3{std::nullopt};      // initialized with std::nullopt

assert(o1.has_value());
assert(!o2.has_value());
assert(!o3.has_value());

o1.reset();
assert(!o1.has_value());
assert(!o1);  // also works because there's a operator bool()
```

Snippet 17: `std::optional` has value cases

You can call `.emplace(...)` to construct the contained value in-place e.g.

```
o1.emplace(69);
assert(*o1 == 69);  // operator* works as you'd expect

o2.emplace(42);
assert(*o2 == 42);

o2.reset();
assert(o2.value_or(999) == 999);

std::optional<std::pair<int, int>> o4;
o4.emplace(6, 9);
assert(o4->first == 6 &&
       o4->second == 9);  // operator-> works as you'd expect
```

Snippet 18: `std::optional`'s `emplace(...)`

Also, `operator*` and `operator->` work intuitively / similarly to how they'd work for pointers but remember that the contained object is actually stored within the optional and the optional isn't a pointer to the object on the heap or somewhere else.

4.1 Super simplified implementation

In fact, here's (a heavily simplified snippet on) how `std::optional` is implemented in `clang`:

```
template <class T>
struct optional {
  union {
    char __null_state_;
    T __val_;
  };
  bool __engaged_;  // true if this optional object contains a value

  constexpr optional() noexcept : __null_state_(), __engaged_(false) {}

  template <class... _Args>
  constexpr explicit optional(std::in_place_t, _Args&&... __args)
      : __val_(std::forward<_Args>(__args)...), __engaged_(true) {}

  void reset() noexcept {
    if (__engaged_) {
      __engaged_ = false;
    }
  }
};

// Note that this is for `T` which are trivially destructible (means no
// user provided destructor for the class and all its non-static data
// members). Non-trivially destructible `T` require explicitly calling the
// destructor like `__val_.~value_type()` where appropriate and
// implementing a custom destructor for the optional class.
```

Snippet 19: `std::optional` implementation - value storage

As you can see, `union` (discussed in L1) is used to contain both a `__null_state_` and the `__val_` within `std::optional`. `union` also eliminates the need to default construct `__val_` for empty optionals, accommodating the constructor taking no arguments. If we do not have `union` and just stored `T __val_` and `bool __engaged_`, we will inevitably have to default construct `__val_` even for empty optionals ie. `__engaged_ == false`, which is wasteful and incorrect as sometimes `T` might not be default constructible.

If you noticed, there's a `in_place_t` in one of the constructors. This is an empty disambiguation tag used to select the correct constructor to call. For instance, if you wanted to create an `optional` and default construct (pass no arguments) the contained object, you might think to just call `std::optional o1{}`. But this will call the constructor of `std::optional` that takes in no arguments and returns a disengaged `optional` object. So, we instead call `std::optional o1{std::in_place}`, which will call the second constructor and ultimately `__val_()` because `__args` is empty. If we instead wanted to construct the contained object with arguments, we can do so too e.g. `std::optional<std::pair<int, double>> o1{std::in_place, 5, 9.99}` which will invoke `__val_(5, 9.99)`, where `__val_` is of type `std::pair<int, double>`.

Note that `std::in_place` is an object of type `std::in_place_t`, defined in the `<utility>` header.

Cool walkthrough of how to implement `std::optional`.

4.2 Some use cases

Wherever you find yourself checking the existence of a valid value in a variable e.g.

Old way:

```
std::pair<int, bool> age;
std::pair<std::string, bool> name;

parseMessageForAgeAndName(message, age, name);

if (age.second) {
  // age given
}

if (name.second) {
  // name given
}
```

Snippet 20: `std::optional` use case

New way:

```
std::optional<int> age;
std::optional<std::string> name;

parseMessageForAgeAndName(message, age, name);

if (age) { // or age.has_value()
  // age given
}

if (name) { // or name.has_value()
  // name given
}
```

Snippet 21: `std::optional` use case

This conveys the intention more clearly to other readers.

5 `std::variant`

We have seen a couple of utility classes to compose types together, and the last one we'll cover here is `std::variant`. It is an algebraic *sum type*, and holds one of several possible *variants*, which can change at runtime. It is essentially a tagged union that knows what the current "active member" is.

Like unions, variants don't have an additional level of indirection, and the allocated memory for the object is directly in the variant itself.

For example, here we have a variant with an `int` alternative and a `std::string` alternative. We can assign directly to the variant, and it will *essentially* perform overload resolution on the constructors of the variant types:

```
std::variant<int, std::string> v;
v = "hello";

std::cout << "idx = " << v.index() << "\n";
if (std::holds_alternative<int>(v)) {
  std::cout << "int = "           //
            << std::get<int>(v)   //
            << "\n";
} else {
  std::cout << "str = "                   //
            << std::get<std::string>(v)   //
            << "\n";
}
```

```
$ ./main.out | sed -n '1,/<1/d;/1>/q;p'
idx = 1
str = hello
```

Snippet 22: Demonstration of a variant holding an `int` and a `string`

This is the basic of using variant. It holds multiple values with possibly distinct types, and we can use `std::holds_alternative` to check if the type *currently contained* in the variant is the one we specified. In the example above, it did *not* hold the `int` alternative, so we went into the other branch.

We can then use `std::get<T>` to *get* the alternative with the given type, as shown above.

Alternatively, we can also use indices to check and access the variants; the `.index()` method can get the index of the currently active variant, and `std::get` can also be used with that index to get that variant. This indexing method has to be used when the variant holds more than one alternative *of the same type*, since the type-based method would be ambiguous.

Note that if the currently active variant is not the one that is `std::got` (haha), then an exception is thrown.

5.1 Exception safety

If an exception is thrown while the variant is being assigned to, then the variant can become `valueless_by_exception`. This means that the variant does not hold *any* variant, and any accesses to it will be invalid until a new value is assigned.

If the assignment was to the same variant type, then whether the variant remains in a valid state depends on the exception safety guarantee of the assignment operator for the type itself. If the contained type remains valid even when an exception is thrown (eg. on copy-assignment), then the variant remains valid.

On the other hand, if an exception is thrown while the variant is being *switched* from one type to another, then there won't be a valid value. This is due to the fact that the storage for the object is in the variant itself, and so the old object must necessarily be destructed before the new one can be constructed.

For example:

```
struct Foo {
  Foo() {}
  Foo(const Foo&) {
    throw 10;
  }
  Foo& operator=(const Foo&) {
    throw 20;
  }

  int x;
};

std::variant<Foo, int> v1 = 10;
std::variant<Foo, int> v2{};

try {
  v1 = Foo();
} catch (...) {
  std::cout << "oops 1\n";
}

try {
  v2 = Foo();
} catch (...) {
  std::cout << "oops 2\n";
}

std::cout << "v1: idx = " << (int) v1.index();
std::cout << ", valueless = " << v1.valueless_by_exception() << "\n";

std::cout << "v2: idx = " << (int) v2.index();
std::cout << ", valueless = " << v2.valueless_by_exception() << "\n";
```

```
$ ./main.out | sed -n '1,/<2/d;/2>/q;p'
oops 1
oops 2
v1: idx = -1, valueless = 1
v2: idx = 0, valueless = 0
```

Snippet 23: An example of variants being valueless by exception

Here, we see that if the variant contained an existing `Foo` and its copy-assignment threw, then its validity would depend on whether `Foo` itself is valid (ie. its implementation). On the other hand, if an exception is thrown while changing the type, then the variant becomes valueless.

Note that this applies to the `emplace` method as well, not just assignment.

5.2 Default constructibility

Variants can only be default constructed if their first variant is default constructible; this might be a factor to consider when picking the order of the types (which otherwise shouldn't matter).

If all the variant types are non-default-constructible for some reason, you can use `std::monostate` as the first variant; it's just an empty struct, which uses no extra space and lets the variant be default constructible.

5.3 `std::visit`

You might be wondering if using a `std::variant` needs to be as painful as checking `holds_alternative` all the time. The answer is *only sometimes*.

`std::visit` can simplify the usage of variants by using the visitor pattern (as the name suggests). It takes in some kind of callable object, as well as a variant, then invokes the callable (using overload resolution if necessary) based on the type of the active variant member.

For example, let's make a variant with 3 possible values:

```
std::variant<int, double, std::string> v1, v2, v3;
v1 = 69;
v2 = 3.14159;
v3 = "hello, world!";
```

5.3.1 Custom visitor class

The *simplest* way to use it — in terms of understanding how it works — is to make a custom struct that overloads `operator()` for each of the possible variant types:

```
struct Visitor {
  void operator()(int x) {  //
    std::cout << "int: "    //
              << x << "\n";
  }
  void operator()(double x) {  //
    std::cout << "dbl: "       //
              << x << "\n";
  }
  void operator()(std::string x) {
    std::cout << "str: "  //
              << x << "\n";
  }
};

std::visit(Visitor{}, v1);
std::visit(Visitor{}, v2);
std::visit(Visitor{}, v3);
```

```
$ ./main.out | sed '1,/<3/d;/3>/,$d'
int: 69
dbl: 3.14159
str: hello, world!
```

Snippet 24: Using `std::visit` with a custom visitor class

5.3.2 Templated visitor

Of course, it would be a real pain to have to keep making more of these visitors whenever we want to deal with a different variant type, so we can actually just pass in a templated function.

The equivalent of which is, of course, a lambda taking an `auto` parameter:

```
v3 = 420;
v2 = 2.71828;
v1 = "help i'm stuck in a variant";

auto visitor = [](const auto& x) {
  using x_t = std::decay_t<decltype(x)>;
  if constexpr (std::is_same_v<x_t, int>) {
    std::cout << "int: " << x << "\n";
  } else if constexpr (std::is_same_v<x_t, double>) {
    std::cout << "dbl: " << x << "\n";
  } else if constexpr (std::is_same_v<x_t, std::string>) {
    std::cout << "str: " << x << "\n";
  } else {
    static_assert(std::is_same_v<x_t, void>, "unreachable!");
  }
};

std::visit(visitor, v1);
std::visit(visitor, v2);
std::visit(visitor, v3);
```

```
$ ./main.out | sed '1,/<4/d;/4>/,$d'
str: help i'm stuck in a variant
dbl: 2.71828
int: 420
```

Snippet 25: Using `std::visit` with a custom visitor class

Wait, it looks like we're back to square 0 with manually checking the type, except this time it's done at compile-time using `if constexpr`. Well, yes.

If we're only using one (templated) lambda, then we do need to check like this if we need to differentiate the types. If we don't, then we can make the visitor entirely generic, like this:

```
auto visitor2 = [](auto x) {
  std::cout << "thing: "  //
            << x << "\n";
};
std::visit(visitor2, v1);
std::visit(visitor2, v2);
std::visit(visitor2, v3);
```

```
$ ./main.out | sed '1,/<5/d;/5>/,$d'
thing: help i'm stuck in a variant
thing: 2.71828
thing: 420
```

Snippet 26: Using entirely an generic lambda, without needing to check the types

5.3.3 `overloaded` pattern

Finally, we have what is arguably the best and most flexible way to use `std::visit`, but also the most complicated to understand. It's commonly known as the `overloaded` pattern, and allows us to use `std::visit` like this:

```
v3 = 995;
v2 = 0.0000000001;
v1 = "it's 2am i'm tired";

auto foo = overloaded{
    [](int x) { std::cout << "int: " << x << "\n"; },
    [](double x) { std::cout << "dbl: " << x << "\n"; },
    [](std::string x) { std::cout << "str: " << x << "\n"; },
};

std::visit(foo, v1);
std::visit(foo, v2);
std::visit(foo, v3);
```

```
$ ./main.out | sed '1,/<6/d;/6>/,$d'
str: it's 2am i'm tired
dbl: 1e-10
int: 995
```

Snippet 27: Using `std::visit` with the `overloaded` pattern

In fact, because of the rules of overload resolution, templated functions (eg. auto lambdas) are selected *after* any non-templated ones, we can have a "fallback" visitor, like so:

```
auto foo2 = overloaded{
    [](int x) {
      std::cout << "int: "  //
                << x << "\n";
    },
    [](auto x) {
      std::cout << "???: "  //
                << x << "\n";
    },
};
std::visit(foo2, v1);
std::visit(foo2, v2);
std::visit(foo2, v3);
```

```
$ ./main.out | sed '1,/<7/d;/7>/,$d'
???: it's 2am i'm tired
???: 1e-10
int: 995
```

Snippet 28: Using `overloaded` pattern to have a fallback visitor

So how is this magical `overloaded` gizmo implemented? Like this:

```
template <typename... Xs>
struct overloaded : Xs... {
  using Xs::operator()...;
};

template <typename... Xs>
overloaded(Xs...) -> overloaded<Xs...>;
```

Snippet 29: The implementation of `overloaded`

How this works is that essentially, each lambda that you pass into `overloaded` (as its initialiser) becomes a base class of the final `overloaded` type. It then brings all the `operator()` into scope, so that each lambda in the set can be seen by `std::visit`, as if they were defined in the `overloaded` type itself.

The deduction guide (the second part) is needed in C++17 to tell the compiler how to deduce the template arguments, as part of class template argument deduction (CTAD).

You don't need to worry too much about the details — everyone just copy-pastes this snippet into their project :D

5.3.4 Miscellaneous notes on `std::visit`

A thing to note is that, regardless of which of the methods below you above to use, the return type of each visitor function (ie. all the overloads of `operator()`, all the lambdas, etc.) must have the same return type.

In order to return multiple possible types (eg. a visitor might return the same variant type as the original), you should explicitly specify that the visitors themselves return a `std::variant`.

Furthermore, if the callable that you provide is not able to handle *all* of the possible variant types, then you'll get a compile error:

```
std::variant<std::string, int, double> v;
auto foo = overloaded{
    [](int) { std::cout << "int\n"; },
    [](double) { std::cout << "double\n"; },
};

std::visit(foo, v);
```

```
clang++ -g -std=c++20 -Wpedantic -Wall -Wextra -Wconversion -Wno-unused-var
iable -Wno-unused-but-set-variable   -c -MMD -o broken.o broken.cpp
In file included from broken.cpp:5:
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/variant:16
45:18: error: no type named 'type' in 'std::invoke_result<overloaded<(lambd
a at broken.cpp:19:7), (lambda at broken.cpp:20:7)> &, std::basic_string<ch
ar> &>'
        _Deduced_type>::type;
        ~~~~~~~~~~~~~~~~^~~~
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/variant:16
63:14: note: in instantiation of function template specialization 'std::__d
o_visit<false, true, overloaded<(lambda at broken.cpp:19:7), (lambda at bro
ken.cpp:20:7)> &, std::variant<std::basic_string<char>, int, double> &>' re
quested here
      return __do_visit(std::forward<_Visitor>(__visitor),
             ^
broken.cpp:23:8: note: in instantiation of function template specialization
 'std::visit<overloaded<(lambda at broken.cpp:19:7), (lambda at broken.cpp:
20:7)> &, std::variant<std::basic_string<char>, int, double> &>' requested
here
  std::visit(foo, v);
      ^
In file included from broken.cpp:5:
/usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/variant:10
14:28: error: cannot initialize a member subobject of type 'int (*)(overloa
ded<(lambda at broken.cpp:19:7), (lambda at broken.cpp:20:7)> &, std::varia
nt<std::basic_string<char>, int, double> &)' with an rvalue of type 'void (
*)(overloaded<(lambda at broken.cpp:19:7), (lambda at broken.cpp:20:7)> &,
std::variant<std::basic_string<char>, int, double> &)': different return ty
pe ('int' vs 'void')
...
```

Snippet 30: Using `std::visit` with a non-exhaustive visitor (the error is very long, so we truncated it)

Lastly, you can actually pass more than one variant to `std::visit`; your visitors should now take 2 arguments. Note that you now need to handle the *cartesian product* of all your variant types. For example, if you had `variant<A, B, C>` and `variant<X, Y, Z>`, you need `(A, X)`, `(A, Y)`, etc. — 9 of them.

6 `iostream`

The next major topic in this lecture is on the `iostream` library.

There are 2 key concepts in the `iostream` library.

1. The `std::ostream` and `std::istream` classes, which encapsulate output and input character streams.
2. Extensibility of `operator<<` via non-member operator overloading.

6.1 Key concept 1: `std::{i,o}stream` and I/O manipulators.

As we've seen, we can send output to `std::ostream` objects with `operator<<`, and receive input from `std::istream` objects with `operator>>`.

```
int i;
std::cin >> i;   // Passing an lvalue to operator>>
                 // parses input into the given object
std::cout << 3;  // operator<< formats the given object into the output
```

Snippet 31: Example uses of `std::cout` and `std::cin`

However, we also have slightly funnier things that we can do as well:

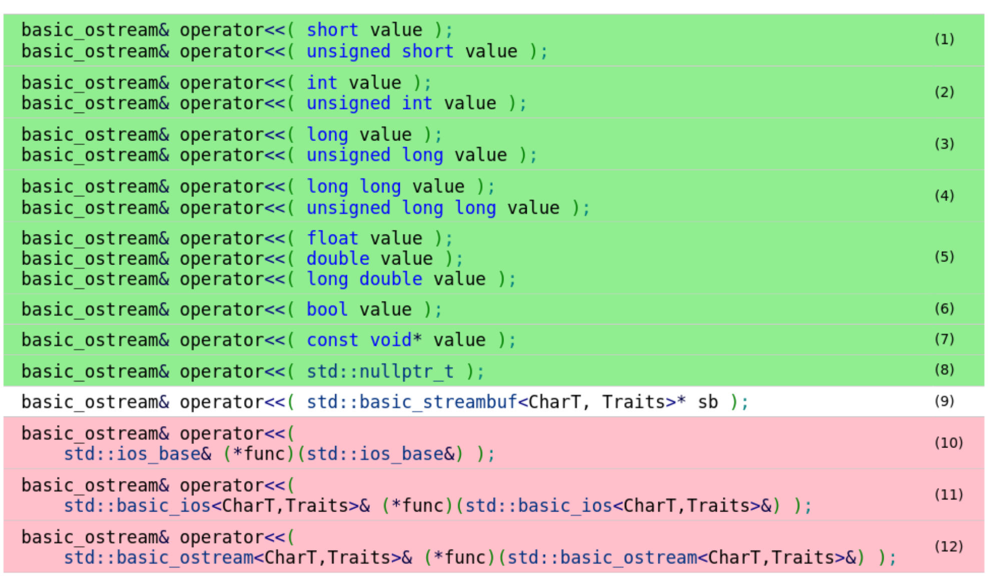
```
std::cout << "Hello world!"
          << std::endl;              // Prints newline and flushes (??)
std::cout << "Hello world part 2!\n"  // Print newline as a normal char
          << std::flush;             // Just flush (??)
```

Snippet 32: What does it mean to print a `std::flush`?

Clearly, "flushing" isn't a printable character, but we can `operator<<` it anyway. To see how this works, let's look at the overloads of `operator<<` that `std::ostream` supports.

Reference:

- `std::ostream::operator<<` member operator overloads
- Non-member operator overloads for characters and string literals



```
basic_ostream& operator<<( short value );
basic_ostream& operator<<( unsigned short value );                                  (1)
basic_ostream& operator<<( int value );
basic_ostream& operator<<( unsigned int value );                                    (2)
basic_ostream& operator<<( long value );
basic_ostream& operator<<( unsigned long value );                                   (3)
basic_ostream& operator<<( long long value );
basic_ostream& operator<<( unsigned long long value );                              (4)
basic_ostream& operator<<( float value );
basic_ostream& operator<<( double value );                                          (5)
basic_ostream& operator<<( long double value );
basic_ostream& operator<<( bool value );                                            (6)
basic_ostream& operator<<( const void* value );                                     (7)
basic_ostream& operator<<( std::nullptr_t );                                        (8)
basic_ostream& operator<<( std::basic_streambuf<CharT, Traits>* sb );               (9)
basic_ostream& operator<<(
    std::ios_base& (*func)(std::ios_base&) );                                       (10)
basic_ostream& operator<<(
    std::basic_ios<CharT,Traits>& (*func)(std::basic_ios<CharT,Traits>&) );         (11)
basic_ostream& operator<<(
    std::basic_ostream<CharT,Traits>& (*func)(std::basic_ostream<CharT,Traits>&) ); (12)
```

`std::ostream::operator<<` member API breakdown

Highlighted in green we have overloads for normal values, and they simply work as expected. Highlighted in red are where the magic happens.

Rather than taking in values, these overloads take in *function pointers*. What `operator<<` will do is to simply call `func(*this)`.

For example, when we do something like `std::cout << std::flush`, we're actually calling the last overload, and so this is the same as doing `std::flush(std::cout)`.

```
std::cout << "Hi" << std::flush  //
          << " there" << std::flush;

// ... is the same as ...
```

```
std::cout << "Hi";
std::flush(std::cout);
std::cout << " there";
std::flush(std::cout);
```

Snippet 33: Fancy I/O manipulators are just function calls

6.2 Key concept 2: Extensibility of `operator<<`

You have already seen this in action when we defined `operator<<` for `SimpleString` and `SimpleVector`. There are of course other standard library types that are printable and all they do is simply overload `operator<<`.

Our previous examples showed how `operator<<` can be extended to print our own data structures, but the same technique can be used to create "modifier" classes, which you might see in some formatting libraries online:

```
struct PrintNumberTwice {
  int n;
};

std::ostream& operator<<(  //
    std::ostream& os,
    PrintNumberTwice n) {
  os << n.n << ' ' << n.n;
  return os;
}
```

```
struct HexNumber {
  int n;
};

std::ostream& operator<<(  //
    std::ostream& os,
    HexNumber n) {
  auto prev_flags = os.flags();
  // Set flag so numbers are printed in hex
  os.setf(os.hex);
  // Prints input in hexadecimal
  os << n.n;
  os.flags(prev_flags);
  return os;
}
```

Snippet 34: Extending `std::ostream` to work with our custom struct

6.3 I/O manipulator overview

Reference: I/O manipulators

Let's now go through some I/O manipulators.

6.3.1 Flushing: `std::flush` and `std::endl`

These are most useful and most common manipulators. `std::flush` does what it says on the tin, and flushes the output stream. `std::endl` is a helper function that prints a newline followed by flushing.

6.3.2 Formatting flags

On top of the manipulators that simply flush, we also have manipulators that set and remove flags.

Reference: fmtflags

We can set flags with manipulators (e.g. `std::hex`) or directly (`os.setf(os.hex)`). `operator<<` can then read out the flags using `os.flags()` as you've seen above, in order to use these flags to affect printing.

This section used to contain a lot of examples for how the various flags worked, but it's probably better to know that they exist and that they're quite extensive, so please read through the list of flags for now, and the next time you feel like you need to do some fancy formatting, you may want to look up if a suitable I/O manipulator solves your problem.

See iomanip for a more comprehensive list of the manipulators, example code, etc.

6.3.3 Monetary and time input/output `std::{get,put}_{money,time}`

See https://gracefu.neocities.org/cpp-get-money.html for a joke article :)

These exist, but honestly they're quite stupid and I've never seen them used nor can I think of any good reason for them to exist.

7 User-defined literals

C++11 added a neat feature that, as the title suggests, allows for user-defined literals. We already have language-defined literals, like these:

```
unsigned long x = 100UL;
float y = 3.14f;
```

Snippet 35: An example of builtin literal suffixes

They allow us to specify the type of literals. For instance, the type of `100UL` is `unsigned long`, `3.14f` is `float`, etc. This lets us alter the "default" types of literals; in this case, `int` and `double`.

As their name would suggest, we can create user-defined ones. The standard library has some of these:

```
using namespace std::literals;

std::complex c = 3.0 + 7i;
std::string s = "hello"s;
std::string_view sv = "world"sv;

std::cout << c << "\n"  //
          << s << "\n"
          << sv << "\n";
```

```
$ ./main.out | sed -n '1,/<1/d;/1>/q;p'
(3,7)
hello
world
```

Snippet 36: An example of standard library UDLs

▼ Literal namespaces

You might have noticed that we had to do `using namespace std::literals` here. This is because each "kind" of literal is in its own namespace. For example, the `chrono` literals (seconds, minutes, etc.) are in `std::literals::chrono_literals`, the `string` one (`s`) is in `...::string_literals`, etc.

Because we cannot qualify (ie. specify the namespace) for literal operators, we need to just use the namespace (technically you can also directly `using` the operator itself).

The full namespace of (for example) `sv` is `std::literals::string_view_literals`, but it also lives in `std::string_view_literals` and `std::literals` via the use of `inline` namespaces.

Here, we used `std::literals`, which brings in all the literals from all the headers you included.

7.1 Defining our own UDLs

By convention, user-defined UDLs (haha…) should be prefixed with `_`, like `"foo"_sv` instead of `"foo"sv` — those without underscores are reserved for the standard library.

Let's revisit `SimpleString` from our previous lecture, and add a literal operator so we can make a `SimpleString` straight from a string literal.

▼ Full code for `SimpleString`

The full code for `SimpleString` is reproduced below just in case (we've made a few small modifications):

```
class SimpleString {
  size_t m_size;
  char* m_buf;

public:
  SimpleString() : m_size{0}, m_buf{new char[1]{'\0'}} {}
  // Create string from string literal
  SimpleString(const char* str)
      : m_size{strlen(str)}, m_buf{new char[m_size + 1]} {
    memcpy(m_buf, str, m_size);
    m_buf[m_size] = '\0';
  }

  SimpleString(const char* str, size_t n)
      : m_size(n), m_buf(new char[m_size + 1]) {
    memcpy(m_buf, str, m_size);
    m_buf[m_size] = 0;
  }

  // Rule of three
  SimpleString(const SimpleString& other)
      : m_size{other.m_size}, m_buf{new char[other.m_size + 1]} {
    memcpy(m_buf, other.m_buf, m_size);
    m_buf[m_size] = '\0';
  }

  SimpleString& operator=(const SimpleString& other) {
    if (this == &other) {
      return *this;
    }

    delete[] m_buf;

    m_size = other.m_size;
    m_buf = new char[m_size + 1];
    memcpy(m_buf, other.m_buf, m_size);
    m_buf[m_size] = '\0';

    return *this;
  }
  ~SimpleString() {
    delete[] m_buf;
  }

  // Basic getters
  size_t size() const {
    return m_size;
  }
  const char* c_str() const {
    return m_buf;
  }

  char& operator[](size_t index) {
    return m_buf[index];
  }
  const char& operator[](size_t index) const {
    return m_buf[index];
  }

  friend std::ostream& operator<<(std::ostream& os,
                                  const SimpleString& str) {
    os << "SimpleString('";
    os.write(str.m_buf, static_cast<std::streamsize>(str.m_size));
    return (os << "')");
  }
  friend auto operator<=>(const SimpleString& lhs,
                          const SimpleString& rhs) {
    size_t smaller_size = std::min(lhs.m_size, rhs.m_size);
    if (auto cmp = memcmp(lhs.m_buf, rhs.m_buf, smaller_size); cmp == 0) {
      return lhs.m_size <=> rhs.m_size;
    } else if (cmp < 0) {
      return std::strong_ordering::less;
    } else {
      return std::strong_ordering::greater;
    }
  }
  friend auto operator==(const SimpleString& lhs,
                         const SimpleString& rhs) {
    return lhs.m_size == rhs.m_size &&
           memcmp(lhs.m_buf, rhs.m_buf, lhs.m_size) == 0;
  }
};
```

There are only a fixed set of allowed signatures for literal operators; in our case, we will choose the `(const char*, size_t)` one. Note that we helpfully get the `size_t` from the compiler so we don't have to compute the length ourselves (obviously, the compiler will know).

All we need to do is define the operator, and it works:

```
SimpleString operator""_ss(  //
    const char* s,
    size_t n) {
  return SimpleString(s, n);
}
```

```
auto ss = "hello"_ss;
std::cout << "ss = "  //
          << ss << "\n";
```

```
$ ./main.out | sed -n '1,/<2/d;/2>/q;p'
ss = SimpleString('hello')
```

7.2 Advanced UDLs

There are many ways to use user-defined literals, not just with strings. With numeric literals, you can create safe, dimensioned unit types, BigInteger-esque things, etc.

7.2.1 Cooked literals

The fixed set of allowed parameter types for literal operators, except the one taking *just* `const char*`, are colloquially known as "cooked" literals; this is because we let the compiler handle the parsing and conversion of the literal into an actual numeric type.

7.2.2 Raw literals

We can also use the `const char*` version, which will be used as a fallback for numeric literals if none of the other overloads match. This gives you the digits as a string literal, which you can then iterate over and do stuff with.

Note that this is not the same as the `const char*, size_t` ones, which are only used for string literals.

7.2.3 Templated UDLs

The most flexible way is to receive the literal as a non-type template parameter. For numeric literals, you can get the actual digits (as characters); these are known as *numeric literal operator templates*:

```
template <char... Nums>
int operator""_bigint() {
  return 69;
}
```

Snippet 37: An example of a *numeric literal operator template*

They are designed this way to circumvent the problem of representation; the largest fundamental integral type in C++ is `unsigned long long`, which is probably too small for "BigInt" class numbers. By not having to pass it as a function argument (ie. value), we just avoid this issue entirely.

Of course, since we can't iterate directly over template parameter packs, this is a little troublesome to use.

For string literals, since C++20 you can also use a non-type template parameter, provided that the type of the NTTP is a class type that is implicitly constructible from a string literal.

Note that for the templated literal operators, the parameter list must be empty.

8 `std::chrono`

C++11 also introduced the `chrono` library, which has a set of useful utilities for handling times; C++20 introduced support for dates as well.

For most cases, `std::chrono` should be preferred over using the old C-style APIs, or using platform specific functions (eg. `clock_gettime` on POSIX).

8.1 Basic concepts

In order to properly use chrono, we should be familiar with its 3 main concepts: *clocks*, *time points*, and *durations*.

8.1.1 Clocks

It should be obvious what clocks are; they provide a source of time measurement. Clocks are defined in terms of an epoch (a starting time) and a tick rate (which also determines their resolution).

The 3 basic clocks are `system_clock`, `steady_clock`, and `high_resolution_clock`.

`system_clock` represents the *wall clock time*; it is not necessarily *monotonic*, ie. the reported time can *go backwards*. This might be caused by timezone changes, user adjustments, etc. Also, if you need to convert to C's `time_t` for any reason, this is the only clock that can do it. Since C++20, the epoch of this clock is specified to be the Unix epoch (1st Jan 1970).

`steady_clock` is a *monotonic* clock, but it is *not* related to "real" wall-clock time. It might be the number of milliseconds since the last reboot, or something else entirely. The only guarantee is that the reported time (number) is strictly non-decreasing, and the time between each "tick" is constant.

`high_resolution_clock` is usually an alias for either `system_clock` or `steady_clock`; it simply asks for the clock with the highest resolution. Unless you really need such a clock, it's usually better to use one of the other clocks for consistent behaviour.

8.1.2 Time points

Again, it should be obvious what these represent — a point in time. They are linked to a specific clock, and are specified as a duration since the epoch of the clock. In terms of the operations you can do on them, you can take their difference to get a duration.

8.1.3 Durations

The last key idea is durations, which are specified as a number of ticks of some time unit; if the unit is in seconds, then a duration of `42` would be 42 seconds.

8.2 Type-safe, dimensional quantities

One key idea that the chrono library uses is that durations should be type-safe, and that the units of a quantity should be part of its type. For example, *seconds* should be a distinct type from *milliseconds*, even if there exists implicit conversions between them (though that's another story).

8.2.1 Safe API design

Suppose we had some function that waited for something to occur, but accepted a timeout:

```
void doAction1(double timeout) {
  // do stuff
}
```

```
doAction1(100);
```

Clearly, this is bad — the actual units of the timeout are not specified anywhere; it might only exist in the documentation, or in a comment that might become outdated.

We can improve this *very slightly* by naming the parameter, and *very slightly more* by documenting the call site:

```
void doAction2(double timeout_ms) {
  // do stuff
}
```

```
doAction2(/* millisecs: */ 100);
```

However, this still relies on comments, which can become outdated, and won't actually prevent compilation if there's a mismatch between the comment and the actual code. In a *type-safe* language, we want to *use* types to help us check the correctness of our code.

8.2.2 Chrono duration types

Enter the duration types; if we look at its declaration:

```
template<
  class Rep,
  class Period = std::ratio<1>
> class duration;
```

Snippet 38: The template declaration of `std::chrono::duration`

It encodes both the underlying type (as `Rep`, for example `int` or `double`), as well as the ratio (`Period`), which encodes the ratio of the duration. The standard defines seconds as having a ratio of 1, so hours would have a ratio of 3600:1 (`std::ratio<3600, 1>`), and milliseconds would have a ratio of 1:1000 (`std::ratio<1, 1000>`). Note that the denominator of a ratio is 1 by default, so it doesn't need to be specified for ratios > 1.

In this way, the type of a duration quantity can express both its numerical representation, as well as its unit.

8.2.3 Converting between duration types

Since we have all the information we need about the type, we can perform conversions between them. For example, this lets us do the following (implicit) conversion:

```
  sc::duration<int, std::milli> ms = 3s;
  std::cout << "3s = " << ms.count() << "\n";
```

```
$ ./main.out | sed '1,/<7/d;/7>/,$d'
3s = 3000
```

Snippet 39: Implicit conversions of `std::chrono`

However, note that not all conversions are allowed implicitly; converting from a floating-point representation to an integer destination will not work. In other words, implicit conversions for durations are only allowed if the conversion can happen without a loss in precision.

```
sc::duration<double, std::ratio<1>> a = 3700ms; // OK
sc::duration<int, std::ratio<1>> b = 3700ms;    // not OK
sc::duration<int, std::milli> a = 3.7s;         // not OK
```

Here, in the first case 3700ms is 3.7s, which can be represented in a `double`, so it works. In the second case, there would be a loss of precision (giving either 3s or 4s, neither of which is correct). In the last case, even though 3.7s would be 3700ms, it is still disallowed because converting from a floating-point representation to an integral one cannot happen implicitly.

8.2.4 Safe API design (part 2)

If we revisit our `doAction` functions, we can rewrite them in terms of chrono types:

```
void doAction3(sc::milliseconds ms) {
  // do stuff
}
```

Snippet 40: An even better API for `doAction`

However, something that might be unexpected is that you *can* pass different types to this function, due to the implicit conversions that we just discussed. For example, both of these calls will succeed:

```
  doAction3(10ms);  // OK
  doAction3(100s);  // also OK!
```

To fully prevent this, you might want to make your own dimensioned unit types that don't have implicit conversions. In this case however, at least the types of each argument would be obvious, and errors can be caught a little easier.

8.3 Chrono literals

`std::chrono` provides a few user-defined-literals for convenience, which you can bring in from `std::chrono_literals`. We've already been using them a lot above; for example:

```
    namespace sc = std::chrono;
    using namespace std::chrono_literals;

    auto one_sec = 1s;
    auto one_min = 1min;
    auto one_hour = 1h;

    std::cout << "1s = " << one_sec.count() << "\n";
    std::cout << "1m = " << one_min.count() << "\n";
    std::cout << "1h = " << one_hour.count() << "\n";
```

```
$ ./main.out | sed -n '1,/<1/d;/1>/q;p'
1s = 1
1m = 1
1h = 1
```

Snippet 41: Using chrono literals

Note that we did `namespace sc = std::chrono` just to save us from all that typing. It's very tedious to keep typing it so often, and it makes code more verbose than it needs to be.

Anyway, note that we can use `.count()` on a `duration` to get the count, in terms of the internal units of representation. However, it's clear that the output isn't very useful — they're all just `1`.

In order to get better output, we should use `duration_cast` to convert the duration into the units of our choice:

```
    using ms = sc::milliseconds;
    std::cout << "1s = " << sc::duration_cast<ms>(one_sec).count()
              << " ms\n";

    std::cout << "1m = " << sc::duration_cast<ms>(one_min).count()
              << " ms\n";

    std::cout << "1h = " << sc::duration_cast<ms>(one_hour).count()
              << " ms\n";
```

```
$ ./main.out | sed -n '1,/<2/d;/2>/q;p'
1s = 1000 ms
1m = 60000 ms
1h = 3600000 ms
```

Snippet 42: Using `duration_cast`

Again, note that we declared some aliases here `ms = sc::milliseconds` just to reduce verbosity.

8.4 Measuring time for benchmarks

One of the most common use cases for `std::chrono` is just to measure the amount of time something takes; we already saw this briefly in previous lectures, but we'll explain them in greater detail now.

The standard pattern for measuring time is to take the timestamp before, then after, and compute the time taken to get a `duration` via subtraction:

```
    auto start = sc::steady_clock::now();

    // do some expensive computation
    system("sleep 2");

    auto duration = sc::steady_clock::now() - start;
    auto ms = sc::duration_cast<sc::milliseconds>(duration);
    std::cout << "2 seconds took " << ms.count() << " ms\n";
```

```
$ ./main.out | sed -n '1,/<3/d;/3>/q;p'
2 seconds took 2007 ms
```

Snippet 43: Measuring time taken with `std::chrono`

Just to illustrate why `duration_cast` is useful, we can try to print the duration directly here:

```
    std::cout << "2 seconds took " << duration.count()  //
              << " (unknown units)\n";
```

```
$ ./main.out | sed -n '1,/<4/d;/4>/q;p'
2 seconds took 2006304516 (unknown units)
```

Snippet 44: Printing the count without `duration_cast`

The units seem to be in nanoseconds here, clearly different from above when we were using the chrono literals. Again, you should not rely on this — use `duration_cast` instead.

8.5 `hh_mm_ss`

There's a neat utility type whose specific purpose is to conveniently decompose any duration into hours, minutes, and seconds.

```
    auto uwu = 69420s + 2.1h + 5min;

    std::cout << "count = " << uwu.count() << "\n";

    auto hms = sc::hh_mm_ss(uwu);
    std::cout << hms.hours().count() << "h, "    //
              << hms.minutes().count() << "m, "  //
              << hms.seconds().count() << "s\n";
```

```
$ ./main.out | sed -n '1,/<5/d;/5>/q;p'
count = 77280
21h, 28m, 0s
```

Snippet 45: Using `hh_mm_ss`

This also illustrates that durations can be composed — you can add and subtract them, as well as multiply and divide by constants:

```
    auto ten_sec = 10s;
    auto five_sec = ten_sec / 2;
    auto a_long_time = ten_sec * 3 + 3h;

    std::cout << "10s = "                              //
              << sc::duration_cast<ms>(ten_sec).count()  //
              << "ms\n";

    std::cout << "5s = "                                //
              << sc::duration_cast<ms>(five_sec).count()  //
              << "ms\n";

    std::cout << "a long time = "
              << sc::duration_cast<ms>(a_long_time).count()  //
              << "ms\n";
```

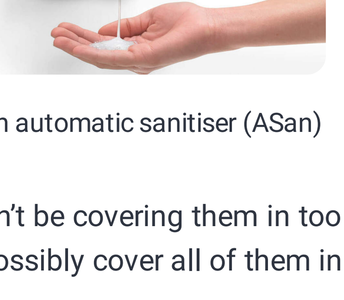
```
$ ./main.out | sed -n '1,/<6/d;/6>/q;p'
10s = 10000ms
5s = 5000ms
a long time = 10830000ms
```

Snippet 46: Composing durations

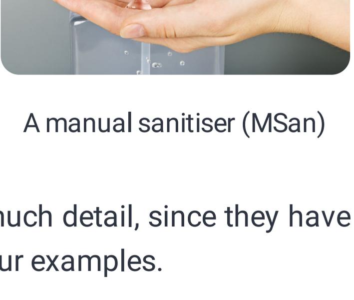
9 Sanitisers

C++ compilers have several useful tools to help us catch bugs when we're writing programs. They operate at runtime, and usually result in a decent amount of overhead. However, the bugs that they expose are typically very hard to catch just by observation or at compile time.

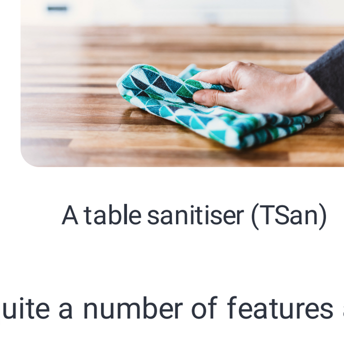
There are a few different types of sanitisers:



An automatic sanitiser (ASan)



A manual sanitiser (MSan)



A table sanitiser (TSan)

We won't be covering them in too much detail, since they have quite a number of features and we can't possibly cover all of them in our examples.

9.1 Address Sanitiser (ASan)

We've already seen bits of ASan before, at least when it blew up on us when we did bad things with heap memory. Indeed, that's the primary class of bugs that ASan catches.

To compile with ASan, pass `-fsanitize=address` to the compiler.

```
// asan.cpp

#include <iostream>

int main() {
  auto xs = new int[10];
  xs[1] = 300;

  // access out of bounds
  xs[10] = 69;

  // use-after-free
  delete[] xs;
  std::cout << "xs[0] = " << xs[0] << "\n";
}
```

Snippet 47: A few examples on types of things that ASan can catch

```
$ ASAN_OPTIONS=halt_on_error=0:print_legend=0 ./asan.out 2>&1 | fold -w 75
setarch: failed to execute ASAN_OPTIONS=halt_on_error=0:print_legend=0: No
such file or directory
```

We get a nice trace of where each error occurred to help us in debugging. Note that we passed some flags to ASan here so that it spams less output, and also to make it continue printing errors after the first one.

9.2 Memory Sanitiser (MSan)

A close cousin of ASan is MSan, and it catches uninitialised memory reads. Note that if we run this program with ASan:

```
#include <iostream>
int main() {
  int x;
  std::cout << "x = " << x << "\n";
}
```

```
$ ./asan-2.out
x = 0
```

Snippet 48: ASan cannot catch uninitialised memory use

We don't get any flagged errors. This is where MSan comes into play:

```
#include <iostream>
int main() {
  int x;
  std::cout << "x = " << x << "\n";

  auto y = new int;
  std::cout << "y = " << *y << "\n";
}
```

```
$ ./msan.out
x = 0
y = 0
```

Snippet 49: ASan cannot catch uninitialised memory use

▼ Why do we need both ASan and MSan?

Since both of these sanitisers operate on memory, it might beg the question of why we need two separate ones; note that it is generally not possible to run two sanitisers at once.

The reason is that each memory sanitiser generally allocates what is known as *shadow memory*; it mirrors each real byte of your program memory with one or more bits of "shadow" memory, which the sanitiser uses to keep track of stuff that you've done to it.

For example, ASan would use this shadow memory to track allocation status, size, etc. while MSan would use it to track whether a location has been written to when trying to read from it.

One reason why they cannot coexist is that it would require *a lot* of virtual memory, which might not be possible.

9.3 Undefined Behaviour Sanitiser (UBSan)

UBSan helps to catch instances of undefined behaviour in your program. It is not exhaustive of course — there's way too many kinds of UB in C++, and some would literally be impossible to catch, even at runtime.

```
#include <iostream>

void f(int x) {
  std::cout << "x = " << x << "\n";
}
int main() {
  // calling a function pointer through a wrong type
  auto ff = (void (*)(double)) f;
  ff(3.1);

  // unaligned access
  auto p = new uint64_t[2];
  p[0] = 69420;
  p[1] = 42069;

  std::cout << "p[0.5] = "                          //
            << *((uint64_t*) ((uint32_t*) &p[0] + 1))  //
            << "\n";
}
```

```
$ ./ubsan.out 2>&1 | fold -w 75
ubsan.cpp:9:3: runtime error: call to function f(int) through pointer to in
correct function type 'void (*)(double)'
/__w/ccc-2/ccc-2/lectures/l07/sanitisers/ubsan.cpp:3: note: f(int) defined
here
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan.cpp:9:3 in
ubsan.cpp:17:16: runtime error: load of misaligned address 0x000000d62fe4 f
or type 'uint64_t' (aka 'unsigned long'), which requires 8 byte alignment
0x000000d62fe4: note: pointer points here
  2c 0f 01 00 00 00 00 00  55 a4 00 00 00 00 00 00  00 00 00 00 00 00 00 00
  11 e0 00 00 00 00 00 00
              ^
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan.cpp:17:16 in
x = 0
p[0.5] = 180684979175424
```

9.4 Thread Sanitiser (TSan)

The last sanitiser that we'll talk about is TSan, which is very useful for catching data races in multi-threaded programs. Note that it cannot catch *race conditions*, which can also be caused by logic errors.

Just as a brief introduction, a data race happens when there are two unordered accesses to a memory location, and at least one of those accesses is a write.

Because multi-threaded programs are inherently non-deterministic, sometimes TSan can miss errors and you might need a couple of runs to catch all possible errors.

Of course, just because TSan doesn't report anything, doesn't mean that your program is free of data races! Take CS3211 in semester 2 for more multi-threaded fun :D

Here's a simple example of a program that has a data race:

```
#include <iostream>
#include <thread>

int x;
int main() {
  auto t1 = std::thread([]() {  //
    x = 10;
  });
  auto t2 = std::thread([]() {  //
    std::cout << "x = " << x << "\n";
  });
  t1.join();
  t2.join();
}
```

Snippet 50: An example of a program that has a data race

```
$ ./tsan.out 2>&1 | fold -w 75
==================
WARNING: ThreadSanitizer: data race (pid=1337)
  Read of size 4 at 0x00000382453c by thread T2:
    #0 main::$_1::operator()() const /__w/ccc-2/ccc-2/lectures/l07/sanitise
rs/tsan.cpp:10:28 (tsan.out+0x4c063e)
    #1 void std::__invoke_impl<void, main::$_1>(std::__invoke_other, main::
$_1&&) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bit
s/invoke.h:60:14 (tsan.out+0x4c05dd)
    #2 std::__invoke_result<main::$_1>::type std::__invoke<main::$_1>(main:
:$_1&&) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bi
ts/invoke.h:95:14 (tsan.out+0x4c04fd)
    #3 void std::thread::_Invoker<std::tuple<main::$_1> >::_M_invoke<0ul>(s
td::_Index_tuple<0ul>) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../i
nclude/c++/9/thread:244:13 (tsan.out+0x4c04a5)
    #4 std::thread::_Invoker<std::tuple<main::$_1> >::operator()() /usr/bin
/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/thread:251:11 (tsa
n.out+0x4c0445)
    #5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<main::$_1>
 > >::_M_run() /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c
++/9/thread:195:13 (tsan.out+0x4c0279)
    #6 <null> <null> (libstdc++.so.6+0xd6de3)
  Previous write of size 4 at 0x00000382453c by thread T1:
    #0 main::$_0::operator()() const /__w/ccc-2/ccc-2/lectures/l07/sanitise
rs/tsan.cpp:7:7 (tsan.out+0x4bfd8b)
    #1 void std::__invoke_impl<void, main::$_0>(std::__invoke_other, main::
$_0&&) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bit
s/invoke.h:60:14 (tsan.out+0x4bfd4d)
    #2 std::__invoke_result<main::$_0>::type std::__invoke<main::$_0>(main:
:$_0&&) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/bi
ts/invoke.h:95:14 (tsan.out+0x4bfc6d)
    #3 void std::thread::_Invoker<std::tuple<main::$_0> >::_M_invoke<0ul>(s
td::_Index_tuple<0ul>) /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../i
nclude/c++/9/thread:244:13 (tsan.out+0x4bfc15)
    #4 std::thread::_Invoker<std::tuple<main::$_0> >::operator()() /usr/bin
/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/thread:251:11 (tsa
n.out+0x4bfbb5)
    #5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<main::$_0>
 > >::_M_run() /usr/bin/../lib/gcc/x86_64-linux-gnu/9/../../../../include/c
++/9/thread:195:13 (tsan.out+0x4bf9e9)
    #6 <null> <null> (libstdc++.so.6+0xd6de3)
  Location is global 'x' of size 4 at 0x00000382453c (tsan.out+0x0000038245
3c)
  Thread T2 (tid=1337, running) created by main thread at:
    #0 pthread_create <null> (tsan.out+0x4496dd)
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, st
d::default_delete<std::thread::_State> >, void (*)()) <null> (libstdc++.so.
6+0xd70a8)
    #2 main /__w/ccc-2/ccc-2/lectures/l07/sanitisers/tsan.cpp:9:13 (tsan.ou
t+0x4bf42f)
  Thread T1 (tid=1337, finished) created by main thread at:
    #0 pthread_create <null> (tsan.out+0x4496dd)
    #1 std::thread::_M_start_thread(std::unique_ptr<std::thread::_State, st
d::default_delete<std::thread::_State> >, void (*)()) <null> (libstdc++.so.
6+0xd70a8)
    #2 main /__w/ccc-2/ccc-2/lectures/l07/sanitisers/tsan.cpp:6:13 (tsan.ou
t+0x4bf41d)
SUMMARY: ThreadSanitizer: data race /__w/ccc-2/ccc-2/lectures/l07/sanitiser
s/tsan.cpp:10:28 in main::$_1::operator()() const
==================
x = 10
ThreadSanitizer: reported 1 warnings
```

10 References

- cppreference `std::pair`
- cppreference `std::forward_as_tuple`
- cppreference `std::tuple`
- cppreference `std::optional`
- cppreference `iomanip`
- cppreference `variant`
- cppreference User-defined literals
- youtube Meeting C++ 2019 - Howard Hinnant - Design Rationale for the `chrono` Library
- youtube CppCon 2014: Kostya Serebryany "Sanitize your C++ code"