9
Essential Utilities

This chapter will introduce some essential classes from the C++ **Utility library**. Some of the metaprogramming techniques presented in the previous chapter will be used in order to work effectively with collections that contain elements of different types.

C++ containers are homogenous, meaning that they can only store elements of one single type. A `std::vector<int>` stores a collection of integers and all objects stored in a `std::list<Boat>` are of type `Boat`. But sometimes, we need to keep track of a collection of elements of different types. I will refer to these collections as **heterogenous collections**. In a heterogeneous collection, the elements may have different types. The following figure shows an example of a homogenous collection of `int`s and a heterogenous collection with elements of different types:

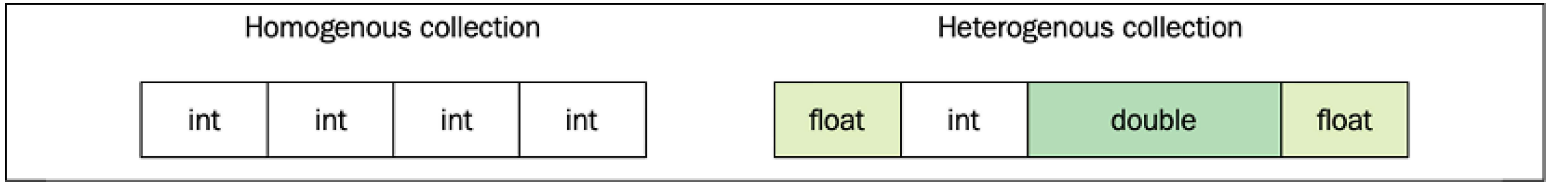

Figure 9.1: Homogenous and heterogenous collections

This chapter will cover a set of useful templates from the C++ Utility library that can be used to store multiple values of various types. The chapter is divided into four sections:

- Representing optional values with `std::optional`
- Fixed size collections using `std::pair`, `std::tuple`, and `std::tie()`
- Dynamically sized collections using the standard containers with elements of type `std::any` and `std::variant`
- Some real-world examples that demonstrate the usefulness of `std::tuple` and `std::tie()`, together with the metaprogramming concepts we covered in *Chapter 8*, *Compile-Time Programming*

Let's begin by exploring `std::optional` and some of its important use cases.

Representing optional values with std::optional

Although quite a minor feature from C++17, `std::optional` is a nice addition to the standard library. It simplifies a common case that couldn't be expressed in a clean and straightforward way prior to `std::optional`. In a nutshell, it is a small wrapper for any type where the wrapped type can be either initialized or uninitialized.

To put it in C++ lingo, `std::optional` is a *stack-allocated container with a max size of one*.

Optional return values

Before the introduction of `std::optional`, there was no clear way to define functions that may not return a defined value, such as the intersection point of two line segments. With the introduction

of `std::optional`, such optional return values can be clearly expressed. What follows is an implementation of a function that returns an optional intersection between two lines:

```
// Prerequisite
struct Point { /* ... */ };
struct Line { /* ... */ };
auto lines_are_parallel(Line a, Line b) -> bool { /* ... */ }
auto compute_intersection(Line a, Line b) -> Point { /* ... */ }
auto get_intersection(const Line& a, const Line& b)
 -> std::optional<Point>
{
 if (lines_are_parallel(a, b))
  return std::optional{compute_intersection(a, b)};
 else
  return {};
}
```

The syntax of `std::optional` resembles that of a pointer; the value is accessed by `operator*()` or `operator->()`. Trying to access the value of an empty optional using `operator*()` or `operator->()` is undefined behavior. It's also possible to access the value using the `value()` member function, which instead will throw an `std::bad_optional_access` exception if the optional contains no value. What follows is a simple example of a returned `std::optional`:

```
auto set_magic_point(Point p) { /* ... */ }
auto intersection = get_intersection(line0, line1);
if (intersection.has_value()) {
```

```
  set_magic_point(*intersection);
}
```

The object held by a `std::optional` is always stack allocated, and the memory overhead for wrapping a type into a `std::optional` is the size of a bool (usually one byte), plus possible padding.

Optional member variables

Let's say we have a class that represents a human head. The head can have a hat of some sort, or no hat at all. By using `std::optional` to represent the hat member variable, the implementation is as expressive as it can be:

```
struct Hat { /* ... */ };
class Head {
public:
  Head() { assert(!hat_); }      // hat_ is empty by default
  auto set_hat(const Hat& h) {
    hat_ = h;
  }
  auto has_hat() const {
    return hat_.has_value();
  }
  auto& get_hat() const {
    assert(hat_.has_value());
    return *hat_;
  }
  auto remove_hat() {
```

```
    hat_ = {};      // Hat is cleared by assigning to {}
  }
private:
  std::optional<Hat> hat_;
};
```

Without `std::optional`, representing an optional member variable would rely on, for example, a pointer or an extra `bool` member variable. Both have disadvantages such as allocating on the heap, or accidentally accessing an optional considered empty without a warning.

Avoiding empty states in enums

A pattern that can be seen in old C++ code bases is *empty states* or *null states* in `enum`s. Here is an example:

```
enum class Color { red, blue, none };  // Don't do this!
```

In the preceding `enum`, `none` is a so-called null state. The reason for adding the `none` value in the `Color` `enum` is to make it possible to represent an optional color, for example:

```
auto get_color() -> Color; // Returns an optional color
```

However, with this design, there is no way to represent a non-optional color, which makes it necessary for *all* code to handle the extra null state `none`.

A better alternative is to avoid the extra null state, and instead represent an optional color with the type `std::optional<Color>`:

```
enum class Color { red, blue };
auto get_color() -> std::optional<Color>;
```

This clearly indicates that we might not get a color back. But we also know that once we have a `Color` object, there is no way it can be null:

```
auto set_color(Color c) { /* c is a valid color, now use it ... */ }
```

When implementing `set_color()`, we know that the client has passed a valid color.

Sorting and comparing std::optional

The `std::optional` is equally comparable and sortable using the rules shown in the following table:

Two *empty* optional values are considered equal.

```
auto a = std::optional<int>{};
auto b = std::optional<int>{};
auto c = std::optional<int>{4};
```

An empty optional is considered *less* than a non-empty.

```
auto a = std::optional<int>{};
auto b = std::optional<int>{4};
auto c = std::optional<int>{5};
```

```
assert(a == b);
assert(b != c);
```

```
assert(a < b);
assert(b < c);
```

Therefore, if you sort a container of `std::optional<T>`, the empty optional values will end up at the beginning of the container, whereas the non-empty optionals will be sorted as usual, as follows:

```
auto c = std::vector<std::optional<int>>{{3}, {}, {1}, {}, {2}};
std::sort(c.begin(), c.end());
// c is {}, {}, {1}, {2}, {3}
```

If you are in the habit of representing optional values using pointers, designing APIs using out parameters, or adding special null states in enums, it's time to add `std::optional` to your toolbox, as it provides an efficient and safe alternative to these anti-patterns.

Let's continue by exploring collections of a fixed size that can hold elements of different types.

Fixed size heterogenous collections

The C++ Utility library includes two class templates that can be used for storing multiple values of different types: `std::pair` and `std::tuple`. They are both collections with a fixed size. Just like `std::array`, it's not possible to add more values dynamically at runtime.

The big difference between `std::pair` and `std::tuple` is that `std::pair` can only hold two values, whereas `std::tuple` can be instantiated with an arbitrary size at compile time. We will begin with a brief introduc-

tion to `std::pair` before moving on to `std::tuple`.

Using std::pair

The class template `std::pair` lives in the `<utility>` header and has been available in C++ since the introduction of the standard template library. It is used in the standard library where algorithms need to return two values, such as `std::minmax()`, which can return both the smallest and the greatest value of an initializer list:

```
std::pair<int, int> v = std::minmax({4, 3, 2, 4, 5, 1});
std::cout << v.first << " " << v.second;    // Outputs: "1 5"
```

The preceding example shows that the elements of a `std::pair` can be accessed through the members `first` and `second`.

Here, the `std::pair` holds values of the same type, so, it would have been possible to return an array here as well. But what makes `std::pair` even more interesting is that it can hold values of *different* types. This is the reason why we consider this a heterogeneous collection, despite the fact that it can only hold two values.

An example from the standard library where `std::pair` holds different values is the associative container `std::map`. The value type of `std::map` is a pair that consists of the key and the element that the key is associated with:

```
auto scores = std::map<std::string, int>{};
scores.insert(std::pair{"Neo", 12}); // Correct but ineffecient
```

```
scores.emplace("Tri", 45);          // Use emplace() instead
scores.emplace("Ari", 33);
for (auto&& it : scores) { // "it" is a std::pair
  auto key = it.first;
  auto val = it.second;
  std::cout << key << ": " << val << '\n';
}
```

The requirement to explicitly name the `std::pair` type has decreased, and in modern C++, it's common to use initializer lists and structured bindings to hide the fact that we are dealing with values of `std::pair`. The following example expresses the same thing but without mentioning the underlying `std::pair` explicitly:

```
auto scores = std::map<std::string, int> {
  {"Neo", 12},                      // Initializer lists
  {"Tri", 45},
  {"Ari", 33}
};
for (auto&& [key, val] : scores) {      // Structured bindings
  std::cout << key << ": " << val << '\n';
}
```

We will talk more about structured binding later in this chapter.

As the name suggests, `std::pair` can only hold two values. C++11 introduced a new utility class called `std::tuple`, which is a generalization of `std::pair` that can hold an arbitrary number of elements.

The std::tuple

The `std::tuple` can be used as a fixed-size heterogeneous collection that can be declared to be of any size. In contrast to `std::vector`, for example, its size cannot change at runtime; you cannot add or remove elements.

A tuple can be constructed with its member types explicitly specified like this:

```
auto t = std::tuple<int, std::string, bool>{};
```

Or, we can initialize it using class template argument deduction, as follows:

```
auto t = std::tuple{0, std::string{}, false};
```

This will make the compiler generate a class, which can roughly be viewed like this:

```
struct Tuple {
  int data0_{};
  std::string data1_{};
  bool data2_{};
};
```

As with many other classes in the C++ standard library, `std::tuple` also has a corresponding `std::make_tuple()` function, which deduces the types automatically from the parameters:

```
auto t = std::make_tuple(42, std::string{"hi"}, true);
```

But as stated earlier, from C++17 and onward, many of these `std::make_` functions are superfluous, since C++17 classes can deduce these types from the constructor.

Accessing the members of a tuple

The individual elements of `std::tuple` can be accessed using the free function template `std::get<Index>()`. You may wonder why the members can't be accessed like a regular container with the `at(size_t index)` member function. The reason is that a member function such as `at()` is only allowed to return one type, whereas a tuple consists of different types at different indices. Instead, the function template `std::get()` is used with the index as a template parameter:

```
auto a = std::get<0>(t);    // int
auto b = std::get<1>(t);    // std::string
auto c = std::get<2>(t);    // bool
```

We can imagine the `std::get()` function being implemented something like this:

```
template <size_t Index, typename Tuple>
auto& get(const Tuple& t) {
  if constexpr(Index == 0) {
    return t.data0_;
  } else if constexpr(Index == 1) {
    return t.data1_;
  } else if constexpr(Index == 2) {
```

```
    return t.data2_;
  }
}
```

This means that when we create and access a tuple as follows:

```
auto t = std::tuple(42, true);
auto v = std::get<0>(t);
```

the compiler roughly generates the following code:

```
// The Tuple class is generated first:
class Tuple {
  int data0_{};
  bool data1_{};
public:
  Tuple(int v0, bool v1) : data0_{v0}, data1_{v1} {}
};
// get<0>(Tuple) is then generated to something like this:
auto& get(const Tuple& tpl) { return data0_; }

// The generated function is then utilized:
auto t = Tuple(42, true);
auto v = get(t);
```

Note that this example can merely be thought of as a simplistic way to imagine what the compiler generates when constructing `std::tuple`; the interior of `std::tuple` is very complex. Still, it is important to understand that a `std::tuple` class is basically a simple struct whose members can be accessed by a compile-time index.

The `std::get()` function template can also use the typename as a parameter. It is used like this:

```
auto number = std::get<int>(tuple);
auto str = std::get<std::string>(tuple);
```

This is only possible if the specified type is contained once in the tuple.

Iterating std::tuple members

From a programmer's perspective, it may seem that `std::tuple` can be iterated with a regular range-based `for`-loop, just like any other container, as follows:

```
auto t = std::tuple(1, true, std::string{"Jedi"});
for (const auto& v : t) {
  std::cout << v << " ";
}
```

The reason this is not possible is that the type of `const auto& v` is only evaluated once, and since `std::tuple` contains elements of different types, this code simply does not compile.

The same goes for regular algorithms, as iterators don't mutate the type pointed to; therefore, `std::tuple` does not provide a `begin()` or `end()` member function, nor does it provide a subscript operator, `[]`, for accessing the values. So, we need to come up with some other way to unroll the tuple.

Unrolling the tuple

As tuples cannot be iterated as usual, what we need to do is use metaprogramming to unroll the loop. From the previous example, we want the compiler to generate something like this:

```
auto t = std::tuple(1, true, std::string{"Jedi"});
std::cout << std::get<0>(t) << " ";
std::cout << std::get<1>(t) << " ";
std::cout << std::get<2>(t) << " ";
// Prints "1 true Jedi"
```

As you can see, we iterate every index of the tuple, which means we need the number of types/values contained in the tuple. Then, since the tuple contains different types, we need to write a meta-function that generates a new function for every type in the tuple.

If we start with a function that generates the call for a specific index, it will look like this:

```
template <size_t Index, typename Tuple, typename Func>
void tuple_at(const Tuple& t, Func f) {
 const auto& v = std::get<Index>(t);
 std::invoke(f, v);
}
```

We can then combine it with a generic lambda, as you learned in *Chapter 2, Essential C++ Techniques:*

```
auto t = std::tuple{1, true, std::string{"Jedi"}};
auto f = [](const auto& v) { std::cout << v << " "; };
tuple_at<0>(t, f);
tuple_at<1>(t, f);
tuple_at<2>(t, f);
// Prints "1 true Jedi"
```

With the function `tuple_at()` in place, we can then move on to the actual iteration. The first thing we need is the number of values in the tuple as a compile-time constant. Fortunately, this value can be obtained by the type trait `std::tuple_size_v<Tuple>`. Using `if constexpr`, we can then unfold the iteration by creating a similar function, which takes different actions, depending on the index:

- If the index is equal to the tuple size, it generates an empty function
- Otherwise, it executes the lambda at the passed index and generates a new function with 1 added to the index

This is how the code will look:

```
template <typename Tuple, typename Func, size_t Index = 0> void tuple_for_each(const Tuple& t, const Func& f) {
 constexpr auto n = std::tuple_size_v<Tuple>;
 if constexpr(Index < n) {
  tuple_at<Index>(t, f);
  tuple_for_each<Tuple, Func, Index+1>(t, f);
```

```
  }
}
```

As you can see, the default index is set to zero so that we don't have to specify it when iterating. This `tuple_for_each()` function can then be called like this, with the lambda directly in place:

```
auto t = std::tuple{1, true, std::string{"Jedi"}};
tuple_for_each(t, [](const auto& v) { std::cout << v << " "; });
// Prints "1 true Jedi"
```

Quite nice; syntactically, it looks pretty similar to the `std::for_each()` algorithm.

Implementing other algorithms for tuples

Expanding upon `tuple_for_each()`, different algorithms iterating a tuple can be implemented in a similar manner. Here is an example of how `std::any_of()` for tuples is implemented:

```
template <typename Tuple, typename Func, size_t Index = 0>
auto tuple_any_of(const Tuple& t, const Func& f) -> bool {
 constexpr auto n = std::tuple_size_v<Tuple>;
 if constexpr(Index < n) {
  bool success = std::invoke(f, std::get<Index>(t));
  if (success) {
   return true;
  }
  return tuple_any_of<Tuple, Func, Index+1>(t, f);
 } else {
```

```
    return false;
  }
 }
```

It can be used like this:

```
auto t = std::tuple{42, 43.0f, 44.0};
auto has_44 = tuple_any_of(t, [](auto v) { return v == 44; });
```

The function template `tuple_any_of()` iterates through every type in the tuple and generates a lambda function for the element at the current index, which it then compares with `44`. In this case, `has_44` will evaluate to `true`, as the last element, a `double` value, is `44`. If we add an element of a type that is not comparable with `44`, such as `std::string`, we will get a compilation error.

Accessing tuple elements

Prior to C++17, there were two standard ways of accessing elements of a `std::tuple`:

- For accessing single elements, the function `std::get<N>(tuple)` was used.
- For accessing multiple elements, the function `std::tie()` was used.

Although they both worked, the syntax for performing such a simple task was very verbose, as shown in the following example:

```
// Prerequisite
using namespace std::string_literals;  // "..."s
```

```
auto make_saturn() { return std::tuple{"Saturn"s, 82, true}; }
int main() {
 // Using std::get<N>()
 {
  auto t = make_saturn();
  auto name = std::get<0>(t);
  auto n_moons = std::get<1>(t);
  auto rings = std::get<2>(t);
  std::cout << name << ' ' << n_moons << ' ' << rings << '\n';
  // Output: Saturn 82 true   }
  // Using std::tie()
 {
  auto name = std::string{};
  auto n_moons = int{};
  auto rings = bool{};
  std::tie(name, n_moons, rings) = make_saturn();
  std::cout << name << ' ' << n_moons << ' ' << rings << '\n';
 }
}
```

In order to be able to perform this common task elegantly, structured bindings were introduced in C++17.

Structured bindings

Using structured bindings, multiple variables can be initialized at once using `auto` and a bracket declaration list. As with the `auto` keyword in general, you can apply control over whether the variables should be mutable references, forward references, const references, or values by using the corresponding modifier. In the following example, a structured binding of `const` references is being constructed:

```
const auto& [name, n_moons, rings] = make_saturn();
std::cout << name << ' ' << n_moons << ' ' << rings << '\n';
```

Structured bindings can also be used to extract the individual members of a tuple in a `for` -loop, as follows:

```
auto planets = {
  std::tuple{"Mars"s, 2, false},
  std::tuple{"Neptune"s, 14, true}
};
for (auto&& [name, n_moons, rings] : planets) {
   std::cout << name << ' ' << n_moons << ' ' << rings << '\n';
}
// Output:
// Mars 2 false
// Neptune 14 true
```

Here's a quick tip. If you want to return multiple arguments with named variables instead of tuple indices, it is possible to return a struct defined inside a function and use automatic return type deduction:

```
auto make_earth() {
  struct Planet { std::string name; int n_moons; bool rings; };
  return Planet{"Earth", 1, false};
}
// ...
```

```
auto p = make_earth();
std::cout << p.name << ' ' << p.n_moons << ' ' << p.rings << '\n';
```

Structured bindings also work with structs, so, we might capture the individual data members directly as follows, even if it is a struct:

```
auto [name, num_moons, has_rings] = make_earth();
```

In this case, we can choose arbitrary names for our identifiers since it's the order of the data members of `Planet` that is relevant, just like when returning a tuple.

Now, we will look at another use case for `std::tuple` and `std::tie()` when handling an arbitrary number of function arguments.

The variadic template parameter pack

The **variadic template parameter pack** enables programmers to create template functions that can accept any number of arguments.

An example of a function with a variadic number of arguments

If we were to create a function that makes a string out of any number of arguments without variadic template parameter packs, we would need to use C-style variadic arguments (just like `printf()` does) or create a separate function for every number of arguments:

```
auto make_string(const auto& v0) {
 auto ss = std::ostringstream{};
 ss << v0;
 return ss.str();
}
auto make_string(const auto& v0, const auto& v1) {
  return make_string(v0) + " " + make_string(v1);
}
auto make_string(const auto& v0, const auto& v1, const auto& v2) {
 return make_string(v0, v1) + " " + make_string(v2);
}
// ... and so on for as many parameters we might need
```

This is the intended use of our function:

```
auto str0 = make_string(42);
auto str1 = make_string(42, "hi");
auto str2 = make_string(42, "hi", true);
```

If we require a large number of arguments, this becomes tedious, but with a parameter pack, we can implement this as a function that accepts an arbitrary number of arguments.

How to construct a variadic parameter pack

The parameter pack is identified by putting three dots in front of the typename and three dots after the variadic argument expands the pack, with a comma in-between:

```
template<typename ...Ts>
auto f(Ts... values) {
 g(values...);
}
```

Here's the syntactic explanation:

- `Ts` is a list of types
- `<typename ...Ts>` indicates that the function deals with a list
- `values...` expands the pack such that a comma is added between every value

To put it into code, consider this `expand_pack()` function template:

```
template <typename ...Ts>
auto expand_pack(const Ts& ...values) {
  auto tuple = std::tie(values...);
}
```

Let's call the preceding function like this:

```
expand_pack(42, std::string{"hi"});
```

In this case, the compiler will generate a function similar to this:

```
auto expand_pack(const int& v0, const std::string& v1) {
  auto tuple = std::tie(v0, v1);
}
```

This is what the individual parameter pack parts expand to:

Expression:	Expands to:
`template <typename... Ts>`	`template <typename T0, typename T1>`
`expand_pack(const Ts& ...values)`	`expand_pack(const T0& v0, const T1& v1)`
`std::tie(values...)`	`std::tie(v0, v1)`

Table 9.1: Expanding expressions

Now, let's see how we can create a `make_string()` function with a variadic parameter pack.

Going further with the initial `make_string()` function, in order to create a string out of every parameter, we need to iterate the pack. There is no way to directly iterate a parameter pack, but a simple work-around would be to make a tuple out of it and then iterate it with the `tuple_for_each()` function template, as follows:

```
template <typename ...Ts>
auto make_string(const Ts& ...values) {
```

```
  auto ss = std::ostringstream{};
  // Create a tuple of the variadic parameter pack
  auto tuple = std::tie(values...);
  // Iterate the tuple
  tuple_for_each(tuple, [&ss](const auto& v) { ss << v; });
  return ss.str();
}
// ...
auto str = make_string("C++", 20);  // OK: str is "C++"
```

The parameter pack is converted into a `std::tuple` with `std::tie()` and then iterated using `tuple_for_each()`. To recap, the reason we need to use `std::tuple` to handle the parameters are because we want to support an arbitrary number of parameters of various types. If we only had to support parameters of one specific type, we could instead have used a `std::array` with a range-based `for`-loop, like this:

```
template <typename ...Ts>
auto make_string(const Ts& ...values) {
  auto ss = std::ostringstream{};
  auto a = std::array{values...};    // Only supports one type
  for (auto&& v : a) { ss << v; }
  return ss.str();
}
// ...
auto a = make_string("A", "B", "C");  // OK: Only one type
auto b = make_string(100, 200, 300);  // OK: Only one type
auto c = make_string("C++", 20);      // Error: Mixed types
```

As you have seen, `std::tuple` is a heterogenous collection with a fixed size and fixed element positions—more or less like a regular struct but without named member variables.

How can we expand upon this to create a dynamically sized collection (such as `std::vector` and `std::list`) but with the ability to store elements of mixed types? We'll look at a solution to this in the following section.

Dynamically sized heterogenous collections

We started this chapter by noting that the dynamically sized containers offered by C++ are homogenous, meaning that we can only store elements of one single type. But sometimes, we need to keep track of a collection that's dynamic in size that contains elements of different types. To be able to do that, we will use containers that contain elements of type `std::any` or `std::variant`.

The simplest solution is to use `std::any` as the base type. The `std::any` object can store any type of value in it:

```
auto container = std::vector<std::any>{42, "hi", true};
```

It has some drawbacks, though. First, every time a value in it is accessed, the type must be tested for at runtime. In other words, we completely lose the type information of the stored value at compile time. Rather, we have to rely on runtime type checks for the information. Secondly, it allocates the object on the heap rather than the stack, which can have significant performance implications.

If we want to iterate our container, we need to explicitly say this to every `std::any` object: *if you are an int, do this, and if you are a char pointer, do that*. This is not desirable as it requires repeated source code, and it is also less efficient than using other alternatives, which we will cover later in this chapter.

The following example compiles; the type is explicitly tested for and casted upon:

```
for (const auto& a : container) {
  if (a.type() == typeid(int)) {
    const auto& value = std::any_cast<int>(a);
    std::cout << value;
  }
  else if (a.type() == typeid(const char*)) {
    const auto& value = std::any_cast<const char*>(a);
    std::cout << value;
  }
  else if (a.type() == typeid(bool)) {
    const auto& value = std::any_cast<bool>(a);
    std::cout << value;
  }
}
```

We simply cannot print it with a regular stream operator since the `std::any` object has no idea of how to access its stored value. Therefore, the following code does not compile; the compiler does not know what's stored in `std::any`:

```
for (const auto& a : container) {
  std::cout << a;          // Does not compile
```

```
}
```

We usually don't need the full flexibility of types that `std::any` offers, and in many cases, we are better off using the `std::variant`, which we will cover next.

The std::variant

If we don't need the ability to store *any* type in the container, but instead we want to concentrate on a fixed set of types declared at container initialization, then `std::variant` is a better choice.

The `std::variant` has two main advantages over `std::any`:

- It does not store its contained type on the heap (unlike `std::any`)
- It can be invoked with a generic lambda, meaning you don't explicitly have to know its currently contained type (more about this in the later sections of this chapter)

The `std::variant` works in a somewhat similar manner to a tuple, except that it only stores one object at a time. The contained type and value are the type and value we assigned it last. The following image illustrates the difference between a `std::tuple` and a `std::variant` when they've been instantiated with the same types:

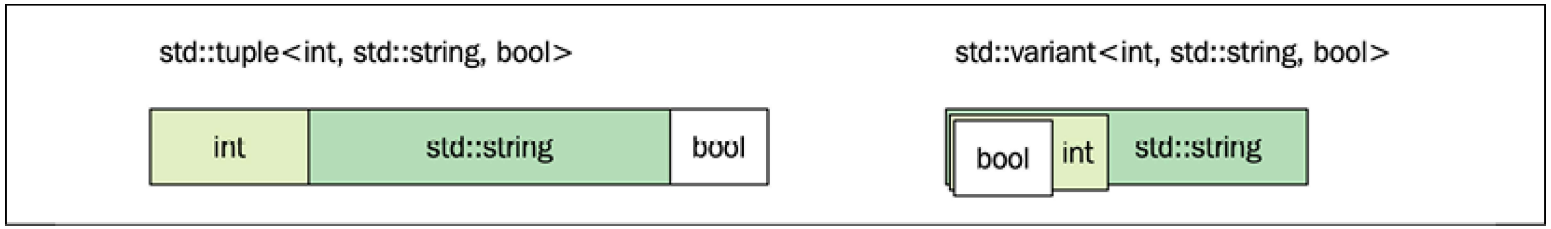

Figure 9.2: Tuple of types versus variant of types

Here's an example of using a `std::variant`:

```
using VariantType = std::variant<int, std::string, bool>;
VariantType v{};
std::holds_alternative<int>(v);  // true, int is first alternative
v = 7;
std::holds_alternative<int>(v);  // true
v = std::string{"Anne"};
std::holds_alternative<int>(v);  // false, int was overwritten
v = false;
std::holds_alternative<bool>(v); // true, v is now bool
```

We are using `std::holds_alternative<T>()` to check whether the variant currently holds a given type. You can see that the type changes when we assign the variant new values.

Apart from storing the actual value, a `std::variant` also keeps track of the currently held alternative by using an index that's usually of size `std::size_t`. This means that the total size of a `std::variant` is typically the size of the biggest alternative, plus the size of the index. We can verify this by using the `sizeof` operator for our types:

```
std::cout << "VariantType: "<< sizeof(VariantType) << '\n';
std::cout << "std::string: "<< sizeof(std::string) << '\n';
std::cout << "std::size_t: "<< sizeof(std::size_t) << '\n';
```

Compiling and running this code using Clang 10.0 with libc++ generates the following output:

```
VariantType: 32
std::string: 24
std::size_t: 8
```

As you can see, the size of the `VariantType` is the sum of `std::string` and `std::size_t`.

Exception safety of std::variant

When a new value is assigned to a `std::variant` object, it is placed in the same location as the currently held value of the variant. If, for some reason, the construction or assignment of the new value fails and throws an exception, the old value may not be restored. Instead, the variant can become **valueless**. You can check whether a variant object is valueless by using the member function `valueless_by_exception()`. This can be demonstrated when trying to construct an object using the `emplace()` member function:

```
struct Widget {
  explicit Widget(int) {    // Throwing constructor
    throw std::exception{};
  }
};
auto var = std::variant<double, Widget>{1.0};
try {
  var.emplace<1>(42); // Try to construct a Widget instance
} catch (...) {
  std::cout << "exception caught\n";
  if (var.valueless_by_exception()) {  // var may or may not
```

```
    std::cout << "valueless\n";      // be valueless
  } else {
    std::cout << std::get<0>(var) << '\n';
  }
}
```

The initial `double` value 1.0 may or may not be gone after the exception has been thrown and caught. The operation is not guaranteed to be rolled back, which we usually can expect from standard library containers. In other words, `std::variant` doesn't provide a strong exception safety guarantee, and the reason for this is performance overhead since it would require `std::variant` to use heap allocations. This behavior of `std::variant` is a useful feature rather than a shortcoming, because it means that you can safely use `std::variant` in code with real-time requirements.

If you instead want a heap allocating version but with a strong exception safety guarantee and a "never-empty" guarantee, `boost::variant` offers this functionality. If you are interested in the challenges of implementing such a type, **https://www.boost.org/doc/libs/1_74_0/doc/html/variant/design.html** offers an interesting read.

Visiting variants

When accessing variables in the `std::variant`, we use the global function `std::visit()`. As you might have guessed, we have to use our main companion when dealing with heterogeneous types: the generic lambda:

```
auto var = std::variant<int, bool, float>{};
std::visit([](auto&& val) { std::cout << val; }, var);
```

When invoking `std::visit()` with the generic lambda and the variant `var` in the example, the compiler will conceptually transform the lambda into a regular class with `operator()` overloads for every type in the variant. This will look something similar to this:

```
struct GeneratedFunctorImpl {
  auto operator()(int&& v)   { std::cout << v; }
  auto operator()(bool&& v)  { std::cout << v; }
  auto operator()(float&& v) { std::cout << v; }
};
```

The `std::visit()` function is then expanded to an `if...else` chain using `std::holds_alternative<T>()`, or a jump table using the index of the `std::variant`, to generate the correct call to `std::get<T>()`.

In the previous example, we passed the value in our generic lambda directly to `std::cout`, regardless of the currently held alternative. But what if we want to do different things, depending on what type we are visiting? A pattern that may be used for this situation is to define a variadic class template that will inherit from a set of lambdas. We then need to define this for each type that we are visiting. Sounds complicated, doesn't it? This may seem a bit magic at first and also puts our metaprogramming skills to the test, but once we have the variadic class template in place, it's easy to use.

We will begin with the variadic class template. Here is how it looks:

```
template<class... Lambdas>
struct Overloaded : Lambdas... {
  using Lambdas::operator()...;
};
```

If you are on a C++17 compiler you also need to add an explicit deduction guide, but it's not needed as of C++20:

```
template<class... Lambdas>
Overloaded(Lambdas...) -> Overloaded<Lambdas...>;
```

That's it. The template class `Overloaded` will inherit from all lambdas that we will instantiate the template with, and the function call operator, `operator()()`, will be overloaded once by each lambda. It's now possible to create a stateless object that only contains multiple overloads of the call operator:

```
auto overloaded_lambdas = Overloaded{
  [](int v)   { std::cout << "Int: " << v; },
  [](bool v)  { std::cout << "Bool: " << v; },
  [](float v) { std::cout << "Float: " << v; }
};
```

We can test it using different arguments and verify that the correct overload is being called:

```
overloaded_lambdas(30031);    // Prints "Int: 30031"
overloaded_lambdas(2.71828f); // Prints "Float: 2.71828"
```

Now, we can use this with `std::visit()` and without the need of having the `Overloaded` object stored in an lvalue. Here is how it finally looks:

```
auto var = std::variant<int, bool, float>{42};
std::visit(Overloaded{
  [](int v)   { std::cout << "Int: " << v; },
  [](bool v)  { std::cout << "Bool: " << v; },
  [](float v) { std::cout << "Float: " << v; }
}, var);
// Outputs: "Int: 42"
```

So, once we have the `Overloaded` template in place, we can use this convenient way of specifying a set of lambdas for different types of arguments. In the next section, we will start using `std::variant` together with standard containers.

Heterogenous collections using variant

Now that we have a variant that can store any type of a provided list, we can expand upon this to a heterogeneous collection. We do this by simply creating a `std::vector` of our variant:

```
using VariantType = std::variant<int, std::string, bool>;
auto container = std::vector<VariantType>{};
```

We can now push elements of different types to our vector:

```
container.push_back(false);
container.push_back("I am a string"s);
```

```
container.push_back("I am also a string"s);
container.push_back(13);
```

The vector will now look like this in memory, where every element in the vector contains the size of the variant, which in this case is `sizeof(std::size_t) + sizeof(std::string)`:

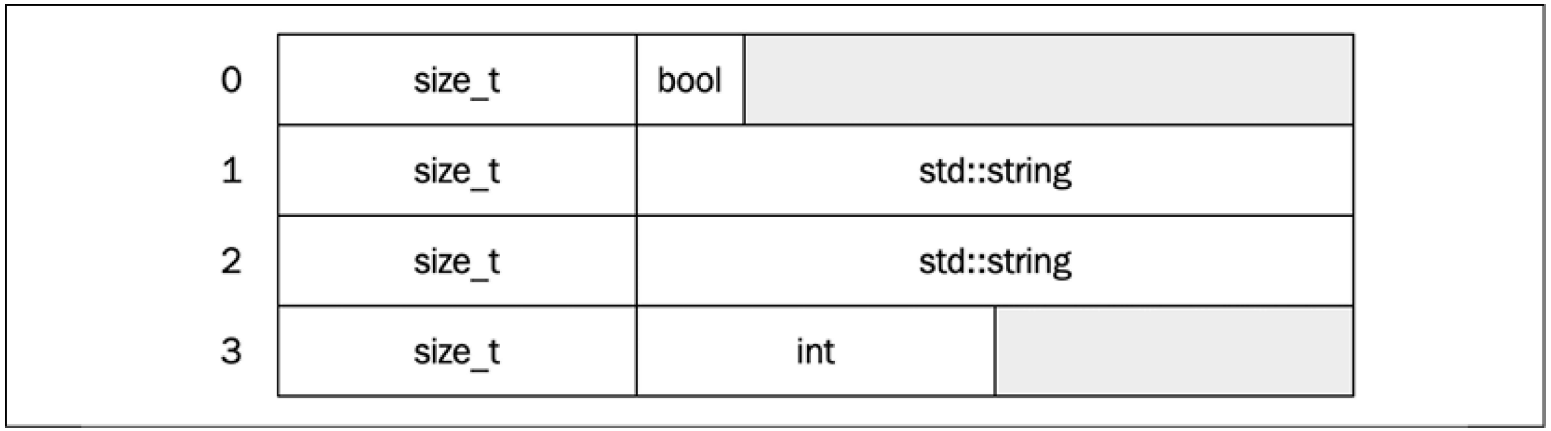

Figure 9.3: Vector of variants

Of course, we can also `pop_back()` or modify the container in any other way the container allows:

```
container.pop_back();
std::reverse(container.begin(), container.end());
// etc...
```

Accessing the values in our variant container

Now that we have the boilerplate for a heterogeneous collection that's dynamic in size, let's see how we can use it like a regular `std::vector`:

1. **Construct a heterogeneous container of variants**: Here, we construct a `std::vector` with different types. Note that the initializer list contains different types:

```
using VariantType = std::variant<int, std::string, bool>;
auto v = std::vector<VariantType>{ 42, "needle"s, true };
```

2. **Print the content by iterating with a regular for-loop**: To iterate the container with a regular `for`-loop, we utilize `std::visit()` and a generic lambda. The global function `std::visit()` takes care of the type conversion. The example prints each value to `std::cout`, independent of the type:

```
for (const auto& item : v) {
  std::visit([](const auto& x) { std::cout << x << '\n';}, item);
}
```

3. **Inspect what types are in the container**: Here, we inspect each element of the container by type. This is achieved by using the global function `std::holds_alternative<type>`, which returns `true` if the variant currently holds the type asked for. The following example counts the number of Booleans currently contained in the container:

```
auto num_bools = std::count_if(v.begin(), v.end(),
                               [](auto&& item) {
```

```
  return std::holds_alternative<bool>(item);
});
```

4. **Find content by both contained type and value**: In this example, we're inspecting the container both for type and value by combining `std::holds_alternative()` and `std::get()`. This example checks whether the container contains a `std::string` with the value `"needle"`:

```
auto contains = std::any_of(v.begin(), v.end(),
                            [](auto&& item) {
  return std::holds_alternative<std::string>(item) &&
    std::get<std::string>(item) == "needle";
});
```

Global function std::get()

The global function template `std::get()` can be used for `std::tuple`, `std::pair`, `std::variant`, and `std::array`. There are two ways to instantiate `std::get()`, with an index or with a type:

- `std::get<Index>()`: When `std::get()` is used with an index, as in `std::get<1>(v)`, it returns the value at the corresponding index in a `std::tuple`, `std::pair`, or `std::array`.
- `std::get<Type>()`: When `std::get()` is used with a type, as in `std::get<int>(v)`, the corresponding value in a `std::tuple`, `std::pair` or `std::variant` is returned. In the case of `std::variant`, a `std::bad_variant_access` exception is thrown if the variant doesn't currently hold that type. Note that if `v` is a `std::tuple` and `Type` is contained more than once, you have to use the index to access the type.

Having discussed the essential templates from the Utility library, let's look at some real-world applications of what we have covered in this chapter.

Some real-world examples

We will end this chapter by examining two examples where `std::tuple`, `std::tie()`, and some template metaprogramming can help us to write clean and efficient code in practice.

Example 1: projections and comparison operators

The need to implement comparison operators for classes dramatically decreased with C++20, but there are still cases where we need to provide a custom comparison function when we want to sort objects in some custom order for a specific scenario. Consider the following class:

```
struct Player {
  std::string name_{};
  int level_{};
  int score_{};
  // etc...
};
auto players = std::vector<Player>{};
// Add players here...
```

Say that we want to sort the players by their attributes: the primary sort order `level_` and the secondary sort order `score_`. It's not uncommon to see code like this when implementing comparison and sorting:

```
auto cmp = [](const Player& lhs, const Player& rhs) {
  if (lhs.level_ == rhs.level_) {
    return lhs.score_ < rhs.score_;
  }
  else {
    return lhs.level_ < rhs.level_;
  }
};
std::sort(players.begin(), players.end(), cmp);
```

Writing comparison operators in this style using nested `if-else` blocks quickly becomes error-prone when the number of attributes increases. What we really want to express is that we are comparing a *projection* of `Player` attributes (in this case, a strict subset). The `std::tuple` can help us rewrite this code in a cleaner way without the need for `if-else` statements.

Let's use `std::tie()`, which creates a `std::tuple` holding references to the lvalues we pass to it. The following code creates two projections, `p1` and `p2`, and compares them using the `<` operator:

```
auto cmp = [](const Player& lhs, const Player& rhs) {
  auto p1 = std::tie(lhs.level_, lhs.score_); // Projection
  auto p2 = std::tie(lhs.level_, lhs.score_); // Projection
  return p1 < p2;
};
std::sort(players.begin(), players.end(), cmp);
```

This is very clean and easy to read compared to the initial version using `if-else` statements. But is this really efficient? It seems like we need to create temporary objects just to compare two players. When running this in a microbenchmark and also inspecting the generated code, there is really no overhead at all to using `std::tie()`; in fact, the version using `std::tie()` was, in this example, slightly faster than the version using `if-else` statements.

Using the ranges algorithms, we can do the sorting by providing the projection as an argument to `std::ranges::sort()`, which makes the code even cleaner:

```
std::ranges::sort(players, std::less{}, [](const Player& p) {
  return std::tie(p.level_, p.score_);
});
```

This is an example of how `std::tuple` can be used in contexts where a full struct with named members is not needed, without sacrificing any clarity in the code.

Example 2: reflection

The term **reflection** refers to the ability to inspect a class without knowing anything about its contents. In contrast to many other programming languages, C++ does not have built-in reflection, which means we have to write the reflection functionality ourselves. Reflection is planned to be included in future versions of the C++ standard; hopefully, we will see this feature in C++23.

In this example, we are going to limit the reflection to give classes the ability to iterate their members, just like we can iterate the members of a tuple. By using reflection, we can create generic functions for

serialization or logging that automatically work with any class. This reduces large amounts of boilerplate code, which is traditionally required for classes in C++.

Making a class reflect its members

Since we need to implement all the reflection functionality ourselves, we will start by exposing the member variables via a function called `reflect()`. We will continue to use the `Player` class that was introduced in the previous section. Here is how it looks when we add the `reflect()` member function and a constructor:

```
class Player {
public:
  Player(std::string name, int level, int score)
     : name_{std::move(name)}, level_{level}, score_{score} {}

  auto reflect() const {
    return std::tie(name_, level_, score_);
  }
private:
  std::string name_;
  int level_{};
  int score_{};
};
```

The `reflect()` member function returns a tuple of references to the member variables by invoking `std::tie()`. We can now start using the `reflect()` function, but first, a note about alternatives to using handcrafted reflection.

C++ libraries that simplify reflection

There have been quite a few attempts in the C++ library world to simplify the creation of reflection. One example is the metaprogramming library *Boost Hana* by Louis Dionne, which gives classes reflection capabilities via a simple macro. Recently, *Boost* has also added *Precise and Flat Reflection* by Anthony Polukhin, which *automatically* reflects public content of classes, as long as all members are simple types.

However, for clarity, in this example, we will only use our own `reflect()` member function.

Using reflection

Now that the `Player` class has the ability to reflect its member variables, we can automate the creation of bulk functionality, which would otherwise require us to retype every member variable. As you may already know, C++ automatically can generate constructors, destructors, and comparison operators, but other operators must be implemented by the programmer. One such function is the `operator<<()`, which outputs its contents to a stream in order to store them in a file, or more commonly, log them in an application log.

By overloading `operator<<()` and using the `tuple_for_each()` function template we implemented earlier in this chapter, we can simplify the creation of `std::ostream` output for a class, like this:

```
auto& operator<<(std::ostream& ostr, const Player& p) {
  tuple_for_each(p.reflect(), [&ostr](const auto& m) {
    ostr << m << " ";
  });
  return ostr;
}
```

Now, the class can be used with any `std::ostream` type, like this:

```
auto v = Player{"Kai", 4, 2568};
std::cout << v;              // Prints: "Kai 4 2568 "
```

By reflecting our class members via a tuple, we only have to update our reflect function when members are added/removed from our class, instead of updating every function and iterating all member variables.

Conditionally overloading global functions

Now that we have a mechanism to write bulk functions using reflection rather than manually typing each variable, we still need to type the simplified bulk functions for every type. What if we wanted these functions to be generated for every type that can be reflected?

We can conditionally enable `operator<<()` for all classes that have a `reflect()` member function by using a constraint.

First, we need to create a new concept that refers to the `reflect()` member function:

```
template <typename T>
concept Reflectable = requires (T& t) {
  t.reflect();
};
```

Of course, this concept only checks whether a class has a member function named `reflect()`; it doesn't always return a tuple. In general, we should be skeptical about weak concepts that only use a single

member function like this, but it serves the purpose of the example. Anyway, we can now overload `operator<<()` in the global namespace, giving all reflectable classes the ability to be compared and printed to a `std::ostream`:

```
auto& operator<<(std::ostream& os, const Reflectable auto& v) {
  tuple_for_each(v.reflect(), [&os](const auto& m) {
    os << m << " ";
  });
  return os;
}
```

The preceding function template will only be instantiated for types that contain the `reflect()` member function, and will therefore not collide with any other overload.

Testing reflection capabilities

Now, we have everything in place:

- The `Player` class we will test has a `reflect()` member function returning a tuple of references to its members
- The global `std::ostream& operator<<()` is overloaded for reflectable types

Here is a simple test that verifies this functionality:

```
int main() {
  auto kai = Player{"Kai", 4, 2568};
  auto ari = Player{"Ari", 2, 1068};
```

```
  std::cout << kai; // Prints "Kai 4 2568"
  std::cout << ari; // Prints "Ari 2 1068"
}
```

These examples have demonstrated the usefulness of small but essential utilities such as `std::tie()` and `std::tuple` when combined with a little bit of metaprogramming.

Summary

In this chapter you have learned how to use `std::optional` to represent optional values in your code. You have also seen how to combine `std::pair`, `std::tuple`, `std::any`, and `std::variant` together with standard containers and metaprogramming to store and iterate over elements of different types. You also learned that `std::tie()` is a conceptually simple yet powerful tool that can be used for projection and reflection.

In the next chapter, you will find out how to further expand your C++ toolbox to create libraries by learning how to construct hidden proxy objects.