Traits, policies, functors, tags

Traits, policies, tags, etc.

- Traits
  - Class/struct template not designed to be instantiated into objects; contents limited to:
    - type definitions (via typedef/using or nested struct/class)
    - constants (via static constexpr)
    - static functions
  - Used as a compile-time function which assigns types/constants/run-time functions to template arguments
- Policy class
  - Non-template class/struct, usually not instantiated
  - Compile-time equivalent of objects, containing types/constants/run-time functions
  - Passed as template argument to customize the behavior of the template
- Functor
  - Class/struct containing non-static function named operator()
  - Usually passed as run-time argument to function templates
  - Functor acts as a function, created by packing a function body together with some data stored or referenced in the body (closure)
- Tag class
  - Empty class/struct
  - Passed as run-time argument to function templates
  - Used to carry a compile-time information by their types themselves
    - Classes/structs are distinguished by their name, not by contents

Policy class

- Policy class
  - Non-template class/struct, usually not instantiated
  - Compile-time equivalent of objects, containing types/constants/run-time functions
  - Passed as template argument to customize the behavior of the template

```
template< typename P> class container { public:
  container(std::size_t n) { m_ = P::alloc(n); /*...*/ }
private:
  typename P::pointer m_;
};
struct my_policy {
  using pointer = void*;
  static pointer alloc(std::size_t s) { /*...*/ }
};
container< my_policy> k;
```

- Motivation:
  - Policy class allows to pass several types/constants/functions as one argument
  - Policy class is the only way to pass a function as a template argument
  - Policy classes are distinguished by name, not by contents
    - This could be an advantage or a disadvantage, depending on context

Policy classes

- Policy class works as a set of parameters for generic code
  - Types (defined by typedef/using or nested classes/enums)
  - Constants (defined by static constexpr)
  - Functions (defined as static)
- The use of policy class instead of individual arguments...
  - ...makes instantiated template names shorter
  - ...avoids order-related mistakes
  - This is the only way how functions may become parameters of a template
- In simple cases, policy classes are not instantiated into objects
  - Some policy-class tricks would not work well when instantiated

Instantiated policy classes

- Some policy classes may be used as objects
  - Such objects carry run-time options to policies; policy functions are not static

```
template< typename P> class container { public:
  container(std::size_t n, const P & p) : p_(p) { m_ = p_.alloc(n); /*...*/ }
private:
  P p_;                               // the template usually contains a policy-object
  typename P::pointer m_;             // compile-time properties extracted from policy-class
};
class my_policy {
public:
  using pointer = void*;
  pointer alloc(std::size_t s) { /*...*/ }
  my_policy(heap * h) : my_heap_(h) {}
private:
  heap * my_heap_;
};
my_policy mp(/*...*/);
container< my_policy> k( mp);
```

- Functors are special cases of instantiated policy classes
  - no type members, only one member function named operator()

Functors

- Functor
  - Class/struct containing non-static function named operator()
    - Usually non-template, but template cases exists (std::less<T>)
    - Since C++11, mostly created by the compiler as a result of a lambda expression
  - Usually passed as run-time argument to function templates
    - Example: std::sort receives a functor representing a comparison function
  - For class templates, a functor becomes both a template argument to the class and a value argument to its constructor
    - Example: std::map receives a functor representing a comparison function
  - Functor acts as a function, created by packing a function body together with some data referenced in the body (closure)
    - Functionality (i.e. the function implementation) selected at compile-time by template instantiation mechanism
    - Functionality parameterized at run-time by the data members of the functor object
  - If there is no data member in the functor, the functionality is equivalent to a non-instantiated policy class containing a static function
    - However, the functor must be instantiated and passed as an object since operator() cannot be static

Instantiated policy classes vs. object oriented programming

- Effect similar to instantiated policy classes can be implemented using OOP
  - The "policy" must have an explicit interface with virtual functions

```
class abstract_allocator { public: virtual void * alloc(std::size_t) = 0; /*...*/ };
```

  - There is no compile-time argument; a pointer to the abstract class is passed at runtime

```
class container { public:
  container(std::size_t n, abstract_allocator * p) : p_(p) { m_ = p_->alloc(n); /*...*/ }
private:
  abstract_allocator * p_;// instead of a policy-object, there is a pointer to an abstract class
  void * m_;              // types can't be extracted by OOP means
};
```

  - A concrete "policy" must inherit the interface; the functions are now virtual

```
class my_allocator : public abstract_allocator {
public:
  virtual void * alloc(std::size_t s) override { /*...*/ }
  my_policy(heap * h) : my_heap_(h) {}
private:
  heap * my_heap_;
};
my_allocator ma(/*...*/);
container k( & ma);       // because OOP requires pointers, ownership of the "policy-object" must be solved somehow
```

- OOP is a runtime mechanism – significantly slower than policy classes
  - in addition, it cannot supply compile-time configuration (types, constants)

Static vs. dynamic polymorphism

- Polymorphism = ability to customize behavior of existing code
- Static/compile-time polymorphism
  - Behavior customized by compile-time (template) arguments
  - There may be a run-time component – policy-objects/functors
  - Customization: The generic code calls **non-virtual** member functions of a class passed as template argument
    - The run-time data are used inside these functions
  - Duck typing: These functions may have any signature compatible with the call
    - They can be templated themselves
  - The compiler compiles the generic code when the template is instantiated with specific policy-object/functor type
    - The compiler knows exactly which function is invoked at the point of customization
    - Function integration (aka. inlining) or inter-procedural optimization possible
- Dynamic/run-time polymorphism
  - Behavior customized by run-time arguments (and run-time type information contained inside objects)
  - Customization: The universal code calls **virtual** member functions via a pointer/reference to an abstract class passed as run-time argument
  - Strong typing: The virtual functions must have exactly the signature defined by the abstract class
    - Virtual functions can not be templates
  - The universal code is not a template – compiled only once
    - There is an indirect virtual-function call at the customization point
    - No optimization possible for the compiler
  - Slower than static/compile-time polymorphism
    - However, the binary code is smaller – relevant in embedded applications etc.
  - Required when behavior must be switched at run time
    - Polymorphic containers, GUI systems, middleware, ...

Static vs. dynamic polymorphism

- Prefer static/compile-time polymorphism whenever possible
- Use dynamic/run-time polymorphism only when needed
    - Polymorphic containers, GUI systems, middleware, ...

- Example: Functors are a case of static polymorphism
    - Functors/lambdas will become arguments of templated functions
    - It is impossible to directly mix different lambdas in one expression/container

```
cond ? [](int & x){ ++x; } : [](int & x){ --x; }          // ERROR
k[0] = [](int & x){ ++x; }; k[1] = [](int & x){ --x; }; // ERROR
```

- These cases may be solved using std::function

```
std::vector< std::function<void(int)>> k;
k[0] = [](int & x){ ++x; }; k[1] = [](int & x){ --x; }; // OK
```

- std::function is implemented using dynamic polymorphism
    - An internal virtual function has a templated implementation
        - Instantiation triggered by the conversion operator of std::function
    - The outer interface of std::function is again a functor
        - std::function acts as a dynamic polymorphism between two static-polymorphism interfaces

Etymology

- Trait [FR]
    - From latin *tractus*
  - Action of firing a projectile
    - Le javelot est une arme de trait. [The javelin is a thrown weapon.]
  - Traction
    - Animaux de trait. [Draft animals.]
  - Line drawn in one movement
    - Un trait noir. [A black line.]
  - Characteristic facial lines
    - Elle a de jolis traits. [She has pretty curves.]
  - Characteristic of a person, a thing
    - Traits saillants d'une rencontre. [Highlights of a meeting.]
- The term "trait" is used in psychology and evolutional biology
    - The set of psychological/evolutional properties of an individual is termed "traits"
  - From there, it was acquired in programming, almost always as "traits":
    - The set of compile-time properties of a programming language item (usually a type)

Traits

- Traits
  - Class/struct template not designed to be instantiated into objects; contents limited to:
    - type definitions (via typedef/using or nested struct/class)
    - constants (via static constexpr)
    - static functions
  - Used as a compile-time function which assigns types/constants/run-time functions to template arguments
  - Most frequently declared with one type argument
    - Used to retrieve information related to the type
    - Example: std::numeric_limits<T> contains constants and functions describing the properties of a numeric type T

- Conventions and syntactic sugar
  - When a traits class contains just one type, the type is named "type"
    - C++11: Usually made accessible directly via template using declaration named "..._t"

```
template< typename T> using some_traits_t = typename some_traits< T>::type;
```

  - When a traits class contains just one constant, the constant is named "value"
    - C++14: Usually made accessible directly via template variable named "..._v"

```
template< typename T> inline constexpr some_type some_traits_v = some_traits< T>::value;
```

Traits - example

- Traits are useful when implementing a template acting on unknown type
  - **std::numeric_limits<T>::lowest()** returns the minimal (finite) value of a numeric type

```
template< typename T> T vector_max(const std::vector<T> & v) {
  T m = std::numeric_limits<T>::lowest();
  for (auto && a : v)
    m = std::max(m, a);
  return m;
}
```

- This example has too narrow interface – a better version uses iterators:
  - Another traits class required to determine the element type:

```
template< typename IT>
std::iterator_traits<IT>::value_type range_max(IT b, IT e) {
  using T = std::iterator_traits<IT>::value_type;
  T m = std::numeric_limits<T>::lowest();
  for (; b != e; ++b)
    m = std::max(m, *b);
  return m;
}
```

std::iterator_traits

- Container-manipulation functions usually use iterators in their interface
- Such functions need to know some properties of the underlying containers

- If IT is an iterator type, **std::iterator_traits<IT>** contains the following types:
  - **difference_type** – a signed type large enough to hold distances between iterators
    - usually std::ptrdiff_t
  - **value_type** – the type of an element pointed to by the iterator
  - **reference** – a type acting as a reference to an element
    - this is the type actually returned by operator* of the iterator
    - usually value_type& or const value_type&
    - it may be a class simulating a reference (e.g. for vector<bool>)
  - **pointer** – a type acting as a pointer to an element
    - value_type*, const value_type*, or a class simulating a pointer
  - **iterator_category** – one of predefined tags describing the category of the iterator
    - std::input_iterator_tag, std::output_iterator_tag, std::forward_iterator_tag, std::bidirectional_iterator_tag, or std::random_access_iterator_tag
    - shall be used via template specialization or using std::is_same_v
- These properties can also be determined using C++20 concepts
  - new versions of algorithms in std::ranges do not rely on std::iterator_traits

std::iterator_traits

- Implemented in standard library as

```
template< typename IT> struct iterator_traits {
  using difference_type = typename IT::difference_type;
  using value_type = typename IT::value_type;
  using reference = typename IT::reference;
  using pointer = typename IT::pointer;
  using iterator_category = typename IT::iterator_category;
};
```

- Any class intended to act as an iterator must define the five types referenced above
  - The five types shall be accessed only indirectly through std::iterator_traits
  - Not required if the iterators are passed only to modern concept-aware generic code

- Since raw pointers may act as iterators, there is a partial specialization:

```
template< typename T> struct iterator_traits<T*> {
  using difference_type = std::ptrdiff_t;
  using value_type = std::remove_cv_t<T>;
  using reference = T&;
  using pointer = T*;
  using iterator_category = std::random_access_iterator_tag;
};
```

  - std::remove_cv_t<T> removes any const/volatile modifiers from T

std::is_reference_v

- Traits returning constants, e.g. std::is_reference_v<T>
  - Based on the traits template std::is_reference<T>
    - general template

```
template< typename T> struct is_reference<T> : std::false_type {};
```

    - partial specializations have higher priority

```
template< typename T> struct is_reference<T&> : std::true_type {};
template< typename T> struct is_reference<T&&> : std::true_type {};
```

    - Uses two type aliases (logically acting as policy classes):

```
using false_type = std::integral_constant<bool, false>;
using true_type = std::integral_constant<bool, true>;
```

    - These are aliases of a particular case of a more general auxiliary class:

```
template< typename U, U v> struct integral_constant {
  static constexpr U value = v;
  // ... there are more members here ... explanation later
};
```

    - The result is represented by a static constexpr member named “value” by convention
  - For convenience, the result may be accessed using the global variable alias:

```
template< typename T> inline constexpr is_reference_v = is_reference<T>::value;
```

Tag class

- Tag class
  - Empty class/struct
  - A tag class acts like a compile-time enumeration constant
    - Unlike an enum type, the set of tag classes may be independently extended
    - It is limited to compile-time, therefore there is no need to assign unique numbering
- Two use cases
  - Tag classes are used as type arguments to templates or member types of a class
    - Example: the tags used for iterator_category
    - In this case, a tag class is never instantiated into object, it is also usually empty
  - Tag classes are used as parameters of a function
    - The tag class is instantiated into an empty runtime object
      (usually optimized out by the compiler)
    - This allows to distinguish between different functions of the same name (e.g. constructors)
- In advanced cases, tag classes are templates
  - They are used to carry the "values" of their template arguments

Tag arguments

- Distinguishing constructors
  - Another use-case for value-less function arguments
  - All constructors have the same name
    - the name cannot be used to specify the required behavior
  - Example: std::optional<T> can store T or nothing

```
using string_opt = std::optional< std::string>;
string_opt x;                                     // initialized as nothing
assert(!x.has_value());
string_opt y(std::in_place);                      // initialized as std::string()
assert(y.has_value() && (*y).empty());
string_opt z(std::in_place, “Hello”);             // initialized as std::string(“Hello”)
assert(z.has_value() && *z == “Hello”);
```

  - Implementation:

```
struct in_place_t {};                             // a tag class
inline constexpr in_place_t in_place;             // an empty variable of tag type
template< typename T> class optional { public:
  optional();                           // initialize as nothing
  template< typename... L>
  optional( in_place_t, L &&... l);     // initialize by constructing T from the arguments l
};
```

Employing type non-equivalence with tag classes

```
template< typename P>
class Value {
  double v;
  // ...
};

struct mass {};

struct energy {};

Value< mass> m;
Value< energy> e;

e = m;    // error
```

- Type non-equivalence
  - Two classes/structs/unions/enums are always considered different
    - even if they have the same contents
  - Two instances of the same template are considered different if their parameters are different
  - It also works with empty classes
    - Called **tag** classes
- Usage:
  - To distinguish types which represent different things using the same implementation
    - Physical units
    - Indexes to different arrays
    - Similar effect to *enum class*