Effective C++ Programming

NIE-EPC (v. 2021):

TYPE ERASURE, STD::FUNCTION, ANY, VARIANT, CRTP, SWAP, ADL,…

Type erasure

- *Recall* — *type erasure* allowed a shared pointer to own/manage and use a deleter of any type.
- This technique may be generalized:

```
class Any {
  struct Base {
    virtual ~Base() = default;
  };
  template <typename T>  // T - type of type-erased object owned/managed by X
  struct Helper : Base {
    T t_;
    Helper(const T& t) : t_(t) { }
  };
  Base* ptr_;
public:
  template <typename T>
  Any(const T& t) : ptr_( new Helper<T>(t) ) { }
  ~Any() { delete ptr_; }
};
```

- Or, better with RAII:

```
  ...
  std::unique_ptr<Base> ptr_;
  ...
  ~Any() { delete ptr_; }
```

Type erasure *(cont.)*

```
class Any {
  struct Base {
    virtual ~Base() = default;
  };
  template <typename T>
  struct Helper : Base {
    T t_;
    Helper(const T& t) : t_(t) { }
  };
  std::unique_ptr<Base> ptr_;
public:
  template <typename T>
  Any(const T& t) : ptr_( new Helper<T>(t) ) { }
};
```

- Now, we can "store" an object of any type into an Any owner:

```
Any a1( 1 );
Any a2( true );
Any a3( std::string("some string") );
```

- *Note* — thanks to type erasure, type of a1, a2, and a3 is the same.
- *Question* — what can be done with such type-erased objects?
  - With the above-provided definition, nothing at all.

Type erasure *(cont.)*

- *Question* — what can be done with such type-erased objects?
- Two different cases:
  1) These objects have some common interface — they provide some common operation.
  2) They do not have anything in common.
- *Ad 1)* This was the case of shared pointer deleters:
  - All of them are supposed to provide a function call operator with a value type-pointer parameter.
  - This operator is then propagated out of the internal type-erasure classes as a virtual function.

```
template <typename T> class shared_ptr {
  struct Base {
    virtual ~Base() = default;
    virtual void operator()(T* ptr) = 0;
  };
  template <typename D>
  struct Helper : Base {
    D del_;
    Helper(const D& del) : del_(del) {}
    virtual void operator()(T* ptr) override
    {  del_(ptr);  }
  };
  T* ptr_; Base* ptr_del_;
public:
  template <typename D = std::default_delete<T>>
  shared_ptr(T* ptr, const D& del = D{})
    : ptr_(ptr), ptr_del_(new Helper<D>{del})
    { }
  ~shared_ptr() { if (use_count() == 1) {
     ptr_del_->operator()(ptr_);
     delete ptr_;
   } }
```

- *Ad 2)* As with Any class — will be discussed later.

Type erasure *(cont.)*

- Generally, if type-erased objects have some common interface, it may be "propagated" even out of the owner class.
  - That is, it may be made a part of its interface.
- *Example* — *task* owner class which type-erases entities callable without arguments.
- *Motivation* — *thread pool*:
  - Fixed number of running threads.
  - Queue of tasks (with a thread-safe access, such as protected by a mutex).
  - Once a task in enqueued (from outside) and some thread is available (idle), this thread dequeues the task and starts its execution.
  - *Task* = any callable entity that can be called without arguments.
- *Problem* — how to add callable entities of different types into the same queue?

Type erasure *(cont.)*

- Initial attempt:

```
class ThreadPool {
  std::vector<std::thread> threads_;  // running threads
  std::mutex m_;                      // protection of q_ shared by threads
  std::queue< ??? > q_;               // queue of tasks
  ...
};
```

- *Problem* — how to define a queue member variable such that we could enqueue different callable entities into it?
- *Example:*

```
void task1() { std::cout << "task1" << std::endl; }       // (1) ordinary function

struct Task2 { void operator()() { std::cout << "task2" << std::endl; } };

int main() {
  ThreadPool tp;

  Task2 task2;                                              // (2) function object (functor)
  auto task3 = []{ std::cout << "task3" << std::endl; };   // (3) lambda function

  // How to store task1, task2, and task3 into tp.q_?
```

- We have 3 entities callable without arguments.
- Each one is of a different type.
- How to put them into the queue q_ of the thread pool tp?

Type erasure *(cont.)*

- *Analysis:*
  - Each callable entity `task1`, `task2`, and `task3` has a different type.
  - All elements of standard library containers (such as `std::vector` or `std::list`) and container adaptors (such as `std::queue` or `std::stack`) must have the same (value) type.
  - ⇒ We cannot store tasks into the thread pool queue directly.
- Possible solution — *type erasure*:
  - Let us start with Any class:

```
class ThreadPool {
  std::vector<std::thread> threads_;  // running threads
  std::mutex m_;                      // protection of q_ shared by threads
  std::queue< Any > q_;               // queue of tasks
```

- Now, we can store all the tasks into the queue.
- *Problem* — we need to eventually dequeue tasks and execute them.
  - ⇒ We need to invoke their function-call operator.
  - Such a functionality is not provided by Any class.

Type erasure *(cont.)*

- *Alternative:*
  - We define tasks as callable entities without arguments.
  - ⇒ All of them share this common interface feature.
  - ⇒ We can make this operation available from the type-erasure owner class:

```
class Task {
  struct Base {
    virtual void operator()() = 0;
    virtual ~Base() = default;
  };
  template <typename T>
  struct Helper : Base {
    T t_;
    Helper(const T& t) : t_(t) { }
    virtual void operator()() override { t_(); }
  };
  std::unique_ptr<Base> ptr_;
public:
  template <typename T>
  Task(const T& t) : ptr_( new Helper<T>(t) ) { }
  void operator()() { ptr_->operator()(); }
};
```

Type erasure *(cont.)*

- Now, we can put any callable entity (without arguments) into `Task` owner and invoke its function call operator:

```
void task1() { std::cout << "task1" << std::endl; }       // (1) ordinary function

struct Task2 { void operator()() { std::cout << "task2" << std::endl; } };

int main() {
  Task2 task2;                                              // (2) function object (functor)
  auto task3 = []{ std::cout << "task3" << std::endl; };  // (3) lambda function

  std::queue<Task> q;
  q.emplace(&task1);  // function is not an object => need to store its pointer instead
  q.emplace(task2);
  q.emplace(task3);

  while (!q.empty()) {
    Task t = std::move(q.front());  // remove task...
    q.pop();                        // ...from the queue...
    t();                            // ...and execute it
  }
}
```

- Live demo: https://godbolt.org/z/W3zPT5x7j.
- The queue in `ThreadPool` can be now defined and used the same way.
- *Note* — if task returns something, the return value is discarded.

std::function

- C++ standard library provides a generic type-erased callable entity owner — class template `std::function`.
- Our `Task`:
  - `Task` can own and call an entity callable without arguments ⇒ basically callable entities without parameters.
  - If the type-erased entity returns, the return value is discarded by `Task`.
- `std::function`:
  - An instance of `std::function` template can own and call callable entities with particular types or parameters and type of return value.
  - The types of parameters and type of return value are determined by `std::function` template argument.
- *Examples:*
  - `std::function<void()>` — basically, equivalent of our `Task` ⇒ stored callable entities have no parameters and do not return values.
  - `std::function<bool(int,int)>` — callable entities have two parameters of type `int` and return `bool`.

std::packaged_task

- `std::packaged_task` — basically `std::function` with some additional functionality.
  - Namely, packaged task allows to obtain a relevant *future* object (of corresponding `std::future` type).
- This future object allows to:
  - Wait for the task to be completed (its execution).
  - Obtain its return value.
  - Catch exception thrown within task execution.
- Moreover, future object may be accessed by other threads than the thread executing the task.
- $\Rightarrow$ Packaged tasks are supposed to be run *asynchronously*.
- *Note* — suitable use case = *thread pools*.

std::packaged_task *(cont.)*

- Simple thread pool (*incomplete*) with packaged tasks:

```
class ThreadPool {
  using Task = std::packaged_task<void()>;
  std::mutex m_;
  std::queue<Task> q_;
  std::condition_variable cv_;
  std::vector<std::thread> threads_;
  void worker() {
    while (true) {
      Task task;
      {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [this]{ return !this->q_.empty(); });
        task = std::move(q_.front());
        q_.pop();
      }
      task();  // task executed here
    }
  }
public:
  ThreadPool(int num_threads = std::thread::hardware_concurrency()) {
    for (int i = 0; i < num_threads; i++)
      threads_.emplace_back( [this]{ this->worker(); } );
  }
  std::future<void> enqueue(Task task) {
    std::future<void> future;
    {
      std::lock_guard<std::mutex> lock(m_);
      future = task.get_future();
      q_.emplace(std::move(task));
    }
    cv_.notify_one();  // wake up some thread
    return future;
  }
};
```

Type erasure — std::any

- *Recall* — Any class can store/own type-erased object of any type.
- Any type ⇒ there is no common interface.
- However, we can provide access to the stored type-erased object.
- *First option* — type-less pointer:
  - In Base and Any classes, there is no knowledge of T.
  - ⇒ Only possibility is to return a pointer to the stored object of type void*.

```
class Any {
  struct Base {
    virtual ~Base() = default;
    virtual void* get() = 0;
  };
  template <typename T>
  struct Helper : Base {
    T t_;
    Helper(const T& t) : t_(t) { }
    virtual void* get() override { return &t_; }
  };
  std::unique_ptr<Base> ptr_;
public:
  template <typename T>
  Any(const T& t) : ptr_( new Helper<T>(t) ) { }
  void* get() { return ptr_->get(); }
};
```

Type erasure — std::any *(cont.)*

- *Consequence* — to use the object, returned pointer needs to be casted into the pointer-to-object-type.
- ⇒ This type must be provided from outside!
  - There is no other way; Any class cannot "remember" a type.

```
std::vector<Any> v;
v.emplace_back(1);                          // type-erased int
v.emplace_back(std::string("some string")); // type-erased std::string

int*         ptr =  static_cast<int*        >( v[0].get() );  // pointer to the stored int
std::string& ref = *static_cast<std::string*>( v[1].get() );  // reference to the stored...
                                                              // ...std::string
```

- *Better option* — wrapping "ugly" casting with a custom function:

```
class Any {
  ...
public:
  ...
  template <typename T> friend T Any_cast(const Any& a);  // friend to have access to ptr_
};

template <typename T> T Any_cast(const Any& a)  // free (non-member) function template
{  return *static_cast< std::remove_reference_t<T>* >( a.ptr_->get() );  }
```

- *Note* — `Any_cast` can either return a reference to the stored object, or its copy.

Type erasure — std::any *(cont.)*

- Such a generic type-erased object owner is provided by the C++ standard library as `std::any` class.
  - It is available since C++17.

```
std::vector<std::any> v;
v.emplace_back(1);                            // type-erased int
v.emplace_back(std::string("some string"));  // type-erased std::strign
int          val = std::any_cast< int          >( v[0] );  // copy of the type-erased object
std::string& ref = std::any_cast< std::string& >( v[1] );  // reference to the type-erased
object
```

- *Notes:*
  - In contrast to *cast operators* (`static_cast`, `reinterpret_cast`,…), `std::any_cast` is not an operator.
  - Instead, it is an “ordinary” function template.
- *Use cases:*
  - `std::any` is suitable for very specific use cases only.
  - *Relevant discussion:* https://devblogs.microsoft.com/cppblog/stdany-how-when-and-why/.

std::variant

- `std::any` — a single-object owner.
  - The owned object is stored in dynamically allocated storage.
  - Its access involves virtual-function dispatch.
  - ⇒ Relatively large runtime and memory overhead.
  - Can store an object of any type, but one needs to “recall” this type during object access.
- Another C++ standard library option — `std::variant`:
  - A single-object owner as well.
  - Is a *variadic template* — template arguments define a list of types.
  - Only objects of these types may be stored in variant.

```
std::any a1 = 1;      // int stored
std::any a2 = true;   // bool stored
std::any a3 = 1.0;    // double stored
a1 = 1.0;             // double stored
```

```
std::variant<int,bool> v1 = 1;      // int stored
std::variant<int,bool> v2 = true;   // bool stored
std::variant<int,bool> v3 = 1.0;    // ERROR
v1 = 1.0;                           // ERROR
```

- *Note* — all *any objects* have the same type, *variants* with different type list have different types.

std::variant *(cont.)*

- List of types defines their *order*.
- ⇒ Each type has some *index* in this order.
- Index of the type of actually stored object is returned by the `index` member function.

```
std::variant<int, bool, double> v = 1.0;  // int has index 0, bool 1, double 2
std::cout << v.index();                   // double is stored => prints out "2"
```

- Stored object is accessed with `std::get` free function template.
  - Template argument — index of the stored object type, or the type itself.
  - Returns a reference to the owned object.

```
std::variant<int, bool, double> v = 1.0;

std::cout << std::get< 2      >();  // prints out "1"
std::cout << std::get< double >();  // effectively the same
```

- Note — C++ is a statically typed language.
  - ⇒ All types must be resolved at runtime.
  - ⇒ This holds also for the return type of `std::get`.
  - ⇒ `index` function call cannot be used as a template argument of `std::get`.

std::variant *(cont.)*

- Owned object needs to be stored in the included storage.
  - ⇒ Variant is explicitly disallowed to dynamically allocate memory [link].

```
std::variant<int, bool> v = 1;
std::cout << (uintptr_t)&std::get<0>(v) - (uintptr_t)&v << std::endl;  // print "0" (GCC)
```

- Implementation — included buffer aligned and sized suitably according the type list.
  - *Buffer alignment* = maximum of alignment requirements for of types.
  - *Buffer size* = maximum of size requirements of types.
  - *Note* — it may be defined with `std::aligned_union` library helper type.
- ⇒ No runtime/memory overhead.
- ⇒ `std::variant` is much preferred than `std::any` once there is a fixed list of type alternatives.
- *Benchmark* — `std::variant` vs `std::any`:
  - *insertion* (object storage) — 6.6× faster with `std::variant` [link];
  - stored object *access* — 6.1× faster with `std::variant` [link].

std::variant *(cont.)*

- `std::variant` vs `std::tuple`:
  - `std::tuple` — multiple object owner ⇒ owns/stores a single object for each type from the type list (template arguments).
  - std::variant — single object owner ⇒ owns/stores at most one object at a given moment (of a type from the type list).

```
std::variant<int, bool, double> v = 1;              // owns int
std::cout << &std::get<int>(v) << std::endl;        // printed "0x7fffa6a7b030" in experiment
v = true;                                           // owns bool
std::cout << &std::get<bool>(v) << std::endl;       // printed "0x7fffa6a7b030"
v = 1.0;                                            // owns double
std::cout << &std::get<double>(v) << std::endl;     // printed "0x7fffa6a7b030"
```

- `std::variant` vs `union`:
  - `std::variant` — basically a type-safe union.
  - `union` is low-level "C" type, which in very unsafe and problematic with classes.
  - *Price* — `std::variant` needs to remember the actual "alternative" (index).

```
union { int i; bool b; double d; } u;
std::cout << sizeof(u);                   // printed "8" in experiment

std::variant<int, bool, double> v;
std::cout << sizeof(v);                   // printed "16" in experiment
```

CRTP

- *Example* — vector 2D graphic editor ⇒ hierarchy of polymorphic classes that represent various graphic objects:

```
class Object2D {  // abstract base class
  public:
    virtual ~Object2D() = default;
    virtual void draw() const;
    ...
};

class Line   : public Object2D { ... };
class Square : public Object2D { ... };
class Circle : public Object2D { ... };
```

- *Scene* is represented as a collection of graphic objects:

```
std::vector< std::unique_ptr<Object2D> > scene;  // all objects in a scene
```

- *Exemplary functionality* — loading scene from a file:

```
void load_scene(const std::string& filename) {
  std::ifstream f(filename); std::string line;
  while (std::getline(f, line)) {
    if      (s == "line")   scene.emplace_back( new Line{} );
    else if (s == "circle") scene.emplace_back( new Circle{} );
    else if (s == "square") scene.emplace_back( new Square{} );
    else throw std::invalid_argument("invalid object name");
  }
  for (const auto & obj : scene) obj->draw();  // draw loaded scene
}
```

CRTP *(cont.)*

- Another functionality — selection of objects:

```
std::vector< Object2D* > selected;  // set of selected objects
void select();  // function that put pointers to selected objects into selected vector
```

```
// resolution of user actions:
if (ACTION == SELECT) select();
...
```

- *Note* — selected is a vector of “normal” (raw) pointers.
  - Vectors cannot hold references.
  - *Smart pointers* do not make sense here ⇐ vector `select` does not own selected objects; only refers to them.
- Yet another functionality — duplication of selected objects.

```
// resolution of user actions:
if      (ACTION == SELECT)    select();
else if (ACTION == DUPLICATE) duplicate();
...
```

```
void duplicate() {
  for (auto p_obj : selected) scene.emplace_back(  ???  );
}
```

- *Problem* — how to create a copy of a pointed-to object?

CRTP *(cont.)*

```
void duplicate() {
  for (auto p_obj : selected) scene.emplace_back(  ???  );
}
```

- *Analysis:*
  - `p_obj` points to an object of derived class type (`Line`, `Square`, or `Circle`).
  - Type of `p_obj` is a pointer-to-base (`Object2D*`).
  - ⇒ We need to invoke a copy constructor of a derived class through a pointer to a base class. *How?*
- Generally, this mechanism is provided by virtual functions.
- ⇒ We would need a "*virtual copy constructor*".
- However, C++ does not support virtual constructors.
- *Alternative* — delegation of a copying semantics into a separate virtual function.
  - This function if typically called `clone`.
  - The whole solution is sometimes called "*clone pattern/idiom*".

CRTP *(cont.)*

- *Solution* — polymorphic copying/cloning:

```
class Object2D {
  ...
  virtual Object2D* clone() const = 0;
};
class Line : public Object2D {
  ...
  virtual Object2D* clone() const { return new Line(*this); }    // copy constructor of...
};                                                               // ...Line called here
class Square : public Object2D {
  ...
  virtual Object2D* clone() const { return new Square(*this); }  // copy constructor of...
};                                                               // ...Square called here

class Circle : public Object2D {
  ...
  virtual Object2D* clone() const { return new Circle(*this); }  // copy constructor of...
};                                                               // Circle called here
```

- *Application* to scene duplication:

```
void duplicate() {
  for (auto p_obj : selected) scene.emplace_back(  p_obj->clone()  );
}
```

- *Drawback* — clone function needs to be repeatedly / “redundantly” implemented throughout the whole class hierarchy.

CRTP *(cont.)*

- *Alternative option* — couldn't we "*inject*" `clone` function into derived classes from some base class?
- Obviously, it is not possible from `Object2D` base class itself.
  - There is no knowledge of the cloned type there.
- The base class where `clone` is defined must know about the cloned object type.
- *Solution*:
  - Making another base class that is a template…
  - …and "remembering" the cloned type in its template parameter.
- *Note:*
  - We cannot make `Object2D` a template.
  - *Polymorphism* requires a single unique base class.
  - Templated `Object2D` would create a separate base class for each template argument.
- *Solution (cont.)* — templated "middle" class with `clone` that:
  - inherits from `Object2D` (⇒ polymorphism with single base),
  - serves as a base for graphic object classes (⇒ injects `clone` into them).

CRTP *(cont.)*

- *Partial solution:*

```
class Object2D {                           // single (original) base => polymorphism
  ...
  virtual Object2D* clone() const = 0;
};

template <typename T>                      // template argument = cloned type
class ClonableObject2D : public Object2D {      // middle class => defines clone
  ...
  virtual Object2D* clone() const { ??? }
};
```

- Derived 2d graphic object classes:
  - derive from `ClonableObject2d` class template,
  - where cloned type = template argument is derived class itself.
- For example, `Line` derives from `ClonableObject2D` and the cloned type needs to be `Line` ⇒ template argument needs to be `Line`.

```
class Line : public ClonableObject2d<Line> {
  ...
};
```

- This technique — inheritance from derived template where base template argument is derived class itself — is called "*curiously recurring template pattern*" (CRTP).

CRTP *(cont.)*

- *Last step* — definition of `clone`:

```
template <typename T> class ClonableObject2D : public Object2D {
  ...
  virtual Object2D* clone() const { return new T( ??? ); }  // invoke copy constructor of T
};
```

- `clone` needs to create new object with copy constructor of `T`.
- The source object to be copied is pointed to by `this` pointer.
- *Problem* — the type of `this` in `ClonableObject2D<T>::clone()` call is `ClonableObject2D<T>*`.
- But, we know that it actually points to an object of type `T`, which should be copied.
- *Example:*

```
Object2D* ptr = new Line;
ptr->clone();  // inside, type of 'this' is ClonableObject2D<Line>*, but it points to Line
```

- *Consequence* — `this` needs to be casted into `T*`:

```
virtual Object2D* clone() const {
  return new T( *static_cast<const T*>(this) );  // or, 'static_cast<const T&>(*this)'
}
```

CRTP *(cont.)*

- *Summary:*

```
class Object2D {                                // single (original) base => polymorphism
  ...
  virtual Object2D* clone() const = 0;
};

template <typename T>                           // template argument = cloned type
class ClonableObject2D : public Object2D {      // middle class => defines clone
  ...
  virtual Object2D* clone() const {
    return new T( *static_cast<const T*>(this) );
  }
};

// derive 2D graphic object classes:
class Line   : public ClonableObject2d<Line  > { ... };
class Square : public ClonableObject2d<Square> { ... };
class Circle : public ClonableObject2d<Circle> { ... };
```

- *Application* to scene duplication is the same:

```
void duplicate() {
  for (auto p_obj : selected) scene.emplace_back(  p_obj->clone()  );
}
```

- *Advantage* — `clone` is defined only once.
  - ⇒ Less code *redundancy* ⇒ better code *maintenance*.
- *Note* — generally, CRTP has broader use than just the *clone idiom*.

CRTP *(cont.)*

- *Instantiation:*

```
class Object2D {
 ...
 virtual Object2D* clone() const = 0;
};

class ClonableObject2D<Line> : public Object2D {
 ...
 virtual Object2D* clone() const { return new Line( *static_cast<const Line*>(this) ); }
};
class ClonableObject2D<Square> : public Object2D {
 ...
 virtual Object2D* clone() const { return new Square( *static_cast<const Square*>(this) ); }
};
class ClonableObject2D<Circle> : public Object2D {
 ...
 virtual Object2D* clone() const { return new Circle( *static_cast<const Circle*>(this) ); }
};
class Line   : public ClonableObject2d<Line  > { ... };
class Square : public ClonableObject2d<Square> { ... };
class Circle : public ClonableObject2d<Circle> { ... };
```

- *Result — one base class*, three *middle classes*, three *derived classes*.
- *Exemplary resolution* — if `ptr` points to `Line`, `ptr->clone()` actually calls `ClonableObject2D<Line>::clone()`, which copies `Line` by its copy constructor.

Function swap

- *Special member functions* related to object content:
  - *default constructor* (sets object to “empty” state),
  - *copy constructor* (copies content into initialized object),
  - *move constructor* (moves content into initialized object),
  - *copy assignment operator* (copies content into existing object),
  - *move assignment operator* (moves content into existing object),
  - *destructor* (destroys content).
- The member functions are *special* in that:
  - they are automatically generated under some circumstances,
  - they may be called without explicit function call syntax.
- *Example:*

```
struct X {
  std::string s;
};  // all 6 special member functions are...
    // ...automatically generated for TwoStrings class

void f(X);

int main() {
  X x;    // default constructor called
  f(x);   // copy constructor called
}         // destructor called
```

Function swap *(cont.)*

- There is one additional "*special*" member function — `swap` for swapping content of two existing objects.
- It is not truly special ⇒ not automatically generated, not called without explicit function call syntax.
- However, it is widely used inside existing (not only C++ standard) library algorithms (sorting, for instance).
- Suppose such an algorithm needs to swap content of two objects of type `T`.
- Then, there are two options:
  1) either there is a custom swap free function specifically designed for arguments of type `T` (typically defined in the same namespace as `T`),
  2) or, there is no such a custom function.
- The algorithms then typically works as follows:
  - Ad 1) They use that custom function for required content swapping.
  - Ad 2) They use `std::swap` instead.

std::swap

- `std::swap` is a library function that is provided in two forms:
  - First, there is a *primary template* that works generally for objects of any type.
  - Then, there are "*specializations*" for various library types [link].

```
struct X { std::string s; };

int main() {
  int i = 1, j = 2;
  std::swap(i, j);              // uses generic primary template
  std::cout << i << j;          // prints out "21" => content-numbers were swapped

  std::string s1("world"), s2("hello ");
  std::swap(s1, s2);            // uses special overload of std::swap for std::string
  std::cout << s1 << s2;        // prints out "hello world" => content-strings were swapped

  X x1("world"), x2("hello ");
  std::swap(x1, x2);            // uses generic primary template
  std::cout << x1.s << x2.s;    // prints out "hello world" => content-strings were swapped
```

- *Questions:*
  1) Why are these "specializations" defined?
  2) Isn't the primary template enough?
  3) Should we define swap function for custom types?

std::swap *(cont.)*

- How is generic primary template of `std::swap` implemented?
  - It is not specified by the C++ standard, but implementations of C++ standard libraries typically define primary template *naturally* as follows:

```
template <typename T>        // primary std::swap template
void swap(T& a, T& b) {

  T temp { std::move(a) }; // moves/copies content from first parameter into a temporary
  a = std::move(b);        // moves/copies content from second parameter into the first
  b = std::move(temp);     // moves/copies content from the temporary into the second param.
}                          // => this effectively swaps content of both parameters
```

- ⇒ For class types, primary template of `std::swap` is implemented in terms of its *special member functions*:

```
template <typename T>         // if T is a class type:
void swap(T& a, T& b) {

  T temp { std::move(a) };  // move/copy constructor called (move preferred)
  a = std::move(b);         // move/copy assignment operator called (ditto)
  b = std::move(temp);      // move/copy assignment operator called (ditto)
}                           // destructor called (for temp)
```

- ⇒ *Move-content operations* are (naturally) preferred if they exist; *otherwise*, *copy-content operations* are used instead.
- *Note* — for *trivially-copyable types*, there is no moving of content.
  - ⇒ All operations involve copying of binary representations.

Custom swap vs std::swap *(cont.)*

- Is it worth defining special swap overload for custom types?
  - Obviously, it is worth for classes without move semantics and "expensive" copy semantics (such as *legacy/pre-C++11* classes).
- *Benchmark* — class with a `std::string` subobject.
- *Two versions* — *first* with move semantics:

```
struct X_move {
  std::string s;
};  // X_move HAS automatically generated move constructor and move assignment operator
```

- *Second*, with copy but without move semantics:

```
struct X_copy {  // for instance, pre-C++11 legacy class
  std::string s;
  X_copy(const char* s) : s(s) { }
  X_copy(const X_copy& other) : s(other.s) { }  // suppresses auto-generation...
                                                  // ...of move-content operations
};  // X_copy HAS NOT automatically generated move constructor and move assignment operator
```

- Comparison of swapping of two objects of these types with `std::`swap primary template:
  - Swapping of content was **4.3× faster** for X_move [link].

Custom swap vs std::swap *(cont.)*

- Let us add a custom swap function for a move-less version:

```
void swap(X_copy& a, X_copy& b)
{
  std::swap(a.s, b.s);  // swapping content = swapping content of subobject s
}
```

- *Results* — swapping of content was **12× faster** with custom swap than with `std::swap` [link].
  - ⇒ It is usually crucial to provide swap function for custom classes if these do not provide move-content operations and are expensive to copy (typically "non-movable" classes that own some resources).
- What about classes that provide move-content operations?

```
void swap(X_move& a, X_move& b)
{
  std::swap(a.s, b.s);
}
```

- *Results* — swapping of content was **2.6× faster** with custom swap than with `std::swap` [link].

Custom swap vs std::swap *(cont.)*

- What is the difference:
  - In all examined cases, swapping of content of `X_...` class object = swapping of content of its subobject (namely, its `s` member variable).
  - In case of `X_move` and custom swap, this *swapping of* `s` is accomplished with "*specialization*" of `std::swap` for `std::string` arguments:

```
void swap(X_move& a, X_move& b) {
  std::swap(a.s, b.s);  // specialization of std::swap for std::string called here
}
```

  - In case of `std::swap`, primary template of `std::swap` is used and instantiated as follows:

```
void swap<X_move>(X_move& a, X_move& b) {

  X_move temp { std::move(a) };  // move constructor of X_move called
  a = std::move(b);              // move assignment operator of X_move called
  b = std::move(temp);           // move assignment operator of X_move called
}                                // destructor of X_move called
```

  - Auto-generated special member function of `X_move` is effectively equivalent with the same special member function of `std::string`.
  - ⇒ *Swapping of* `s` is accomplished with the following calls of `std::string` *special member functions*: 1× *move constructor*, 2× *move assignment operator*, 1x *destructor*.

Custom swap vs std::swap *(cont.)*

- *Conclusion:*
  - With a custom swap function, content of s member variable is swapped with the specialization of `std::swap` for `std::string` arguments.
  - With `std::swap` function, content of s member variable is swapped with:
    - single call of `std::string` *move constructor*,
    - double call of `std::string` *move assignment operator*,
    - and single call of `std::string` *destructor*.
- In both cases, the final outcome (observable behavior) of swapping is the very same!
- However, within our experiment and used C++ implementation, the compiler was not able to optimize both cases into the equally efficient machine code.
- *Why not?*

Custom swap vs std::swap *(cont.)*

- *Exemplary problem:*
  - After each *move-content* operation, s member variable of source object needs to be put into *moved-from* = *"empty string"* state, which requires some work.
  - However, within `std::swap` instance, this is unnecessary since in all 3 cases, the s member variable is then either *assigned new content* or *destructed*:

```
void swap<X_move>(X_move& a, X_move& b) {
  X_move temp { std::move(a) };  // a.s put into empty string state...
  a = std::move(b);              // ...while it is then assigned new content
```

```
  a = std::move(b);    // b.s put into empty string state...
  b = std::move(temp); // ...while it is then assigned new content
```

```
  b = std::move(temp); // temp.s put into empty string state...
}                      // ...while it is then destructed
```

  - Compiler optimizers are simply not *perfect/ideal* from our point of view.
  - Here, a compiler may not recognize that "resetting" strings is unnecessary, and generate machine code for it.
  - *Note* — due to SSO, *move-content* operations for `std::string` are non-trivial but relatively complex.

Custom swap vs std::swap *(cont.)*

- On the contrary, "specialization" of `std::swap` for `std::string` does not require any resetting to empty state.
- If strings are *long*, swapping of content effectively just turns into swapping content of few member variables (pointer + integers).
- ⇒ "Specialization" can be implemented in such a way that it does not contain any unnecessary operations.
- With generic `std::swap` based on *move-content* operations, additional operations are involved and it may be hard or impossible for a compiler to optimize them away and generate equally efficient machine code.
- *Experiment* — swapping content of two `std::string` objects with primary template and specialization of `std::swap`: [link].
- *Consequence* — generally, even for classes that provide *move-content* operations, it may be worth providing custom swap function.

Using swap function

- How to use swap function, for instance, in a quicksort algorithm?

```
template <typename T> long partition(T* a, long lo, long hi) {
  long i = lo;
  for (long j = lo; j < hi; j++)
    if (a[j] < a[hi]) ??? ;  // need to swap a[i++] and a[j], HOW?
  ??? ;                      // need to swap a[i] and a[hi], HOW?
  return i;
}

template <typename T> void qs(T* a, long lo, long hi) {
  if (lo < hi) {
    long p = partition(a, lo, hi);
    qs(a, lo, p - 1);
    qs(a, p + 1, hi);
  }
}
```

- Within partitioning, we need to swap sorted elements; *how*?
- We want:
  1) to use custom swap function if it exists,
  2) to use `std::swap` otherwise.
- If we called
  - `swap`, then compilation error would be generated in the *second case*.
  - `std::swap`, then custom swap would not be used in the *first case*.

Using swap function

- Solution is simple:

```
template <typename T> long partition(T* a, long lo, long hi) {
  using std::swap;

  long i = lo;
  for (long j = lo; j < hi; j++)
    if (a[j] < a[hi]) swap(a[i++], a[j]);
  swap(a[i], a[hi]);
  return i;
}
```

- Directive `using` “injects” the name swap from `std` namespace into *global namespace*.
- *Consequently*, swap call will:
  - use custom swap (its “specialization”) if it exists (better overloading candidate),
  - fall back to primary `std::swap` template otherwise.
- *Apparent problem:*
  - What if custom swap is in some custom (for instance, library) *namespace*?
  - Will this solution work?

Argument dependent lookup

- *Example:*

```
namespace our_library
{
  struct X {
    std::string s;
    bool operator<(const X& rhs) const { return s < rhs.s; }  // lexicographic comparison
  };

  void swap(X& a, X& b) { std::swap(a.s, b.s); }
}
```

```
int main() {
  our_library::X a[2] = { "B", "A" };
  qs(a, 0, 1);
}
```

- In partition, there is no "knowledge" of `our_library` namespace:

```
template <typename T> long partition(T* a, long lo, long hi) {
  using std::swap;
  using our_library::swap;  // we don't know about our_library namespace here

  long i = lo;
  for (long j = lo; j < hi; j++)
    if (a[j] < a[hi]) swap(a[i++], a[j]);
  swap(a[i], a[hi]);
  return i;
}
```

Argument dependent lookup *(cont.)*

- Which overload of swap will be called when T is `our_library::X`?

```
long partition<our_library::X>(our_library::X* a, long lo, long hi) {
  using std::swap;
  long i = lo;
  for (long j = lo; j < hi; j++)
    if (a[j] < a[hi]) swap(a[i++], a[j]);
  swap(a[i], a[hi]);
  return i;
}
```

- *Good news* — custom `our_library::swap` will be used in this case.
- Reason = "***argument dependent lookup***" (**ADL**):
  - If types of arguments are from some namespace, candidates for overloading are looked-up in this namespace as well.
- Without ADL, it would be almost impossible to write generic code.
  - Its developer can't know about namespaces from which involved types of objects will come from.