C++ API Design

Written by:

- Bernard Teo Zhi Yi

Last updated: 19 July 2022

Feedback form for Lecture 11

1 Introduction

This lecture is going to be somewhat different from the rest.

In all our other lectures, we've been covering various C++ language and library features, and discussing how we should use them. However, this lecture is mostly a design philosophy class, in the search of ways to design classes (i.e. encapsulation) well — how to make the public-facing interface of your class as "sound" as possible.

When we talk about the public-facing interface of a class, we really mean the public (and possibly the protected) member functions. (As we're creating an encapsulation abstraction, all member fields will be private). We want to design these public member functions so that as a whole, they respect some properties that make your class easy to use correctly.

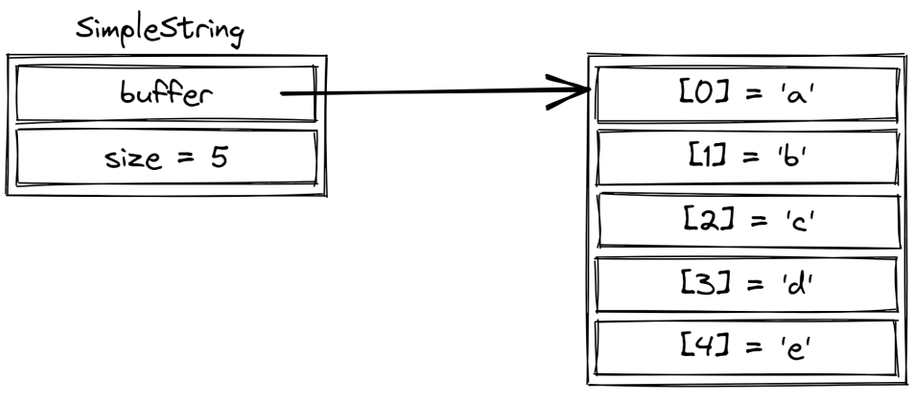
2 Ownership

We've briefly touched on ownership in Lecture 3. We said that an object *owns* a resource (usually heap memory, but possibly file descriptors, mutexes, and many other things) if it is responsible for releasing that resource (e.g. freeing the heap memory), usually in its destructor. We designed the `SimpleString` class, which owns the buffer pointed to by `m_buf`, and frees it in its destructor:

```
class SimpleString {
  size_t m_size;
  char* m_buf;  // the buffer it points to is owned by this SimpleString

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

  SimpleString& operator=(const SimpleString& other) {
    char* new_buf = new char[other.m_size + 1];
    memcpy(new_buf, other.m_buf, other.m_size);
    new_buf[other.m_size] = '\0';

    delete[] m_buf;

    m_size = other.m_size;
    m_buf = new_buf;

    return *this;
  }

  SimpleString(SimpleString&& other)
      : m_size{other.m_size}, m_buf{other.m_buf} {
    other.m_size = 0;
    other.m_buf = new char[1]{'\0'};
  }

  SimpleString& operator=(SimpleString&& other) {
    size_t new_size = other.m_size;
    char* new_buf = other.m_buf;

    other.m_size = 0;
    other.m_buf = new char[1]{'\0'};

    delete[] m_buf;

    m_size = new_size;
    m_buf = new_buf;
```

```
    return *this;
  }

  friend void swap(SimpleString& a, SimpleString& b) {
    std::swap(a.m_size, b.m_size);
    std::swap(a.m_buf, b.m_buf);
  }

  ~SimpleString() {
    delete[] m_buf;  // free the buffer
  }

  /* other member functions */
};
```

Snippet 1: `SimpleString` implementation



`SimpleString` diagram

To ensure that each object owns a unique buffer, we wrote our own copy constructor and copy assignment operator. We also implemented our own move constructor and move assignment operator, so that we can move objects efficiently. In all these functions, we took care to ensure that every buffer remained owned by a unique object, and each object always ended up owning a unique buffer.

So ownership is all about which object is responsible for releasing each resource used in your program.

2.1 `std::unique_ptr`

Pointer ownership is one of the more difficult things to get right, for at least three reasons:

- It's possible to copy or reassign pointers, and these are things you often want to do, given that you're using pointers in the first place. When done without adequate care, this may lead to (buggy) situations where two objects end up holding a pointer to the same heap object, or you end up losing the last reference to a heap object without freeing it (i.e. a memory leak).
- As the type system doesn't distinguish between owned and non-owned pointers, it is left up to the programmer to know which pointers they need to free.
- Functions must ensure that all temporary heap memory owned by a function are freed, even if the function exits by throwing an exception.

Thankfully, we have `std::unique_ptr`, which is a thin wrapper around a raw pointer that makes the pointer movable but uncopyable (ensuring that each `std::unique_ptr` object is either a

unique pointer to some heap memory, or null).

```
void f() {
  // Default constructor: sets the pointer to null
  std::unique_ptr<MyClass> anim;

  // `std::make_unique`: convenience function to construct
  // an object and put it in a `std::unique_ptr`
  anim = std::make_unique<MyClass>(arg1, arg2);

  // This won't work, since you can't copy a `std::unique_ptr`
  // (otherwise multiple of them might point to the same heap object)
  // std::unique_ptr<MyClass> anim_bad = anim;

  // Move constructor: moves the contained pointer,
  // and sets the moved-from pointer to null
  std::unique_ptr<MyClass> anim2 = std::move(anim);

}  // The heap object is automatically freed at this point,
   // in the destructor of `std::unique_ptr<MyClass>`
```

Snippet 2: Basic usage of `std::unique_ptr`

Since copying a `std::unique_ptr` is not allowed, and moving sets the moved-from object to null, proper use of the `std::unique_ptr` interface ensures that each `std::unique_ptr` is, well, a *unique* pointer to a `MyClass` object on the heap (or null). Thus, the public interface of `std::unique_ptr` is "correct" in the sense that it ensures that the *uniqueness* property always holds, and the object being owned is freed upon destruction.

SimpleUniquePtr demo

2.1.1 `std::unique_ptr` to a heap allocated array

It's quite common that we want this heap allocation only because we want to store an array with a size that is only known at runtime. This means we can't declare it on the stack (well, unless we use GCC extensions, but still this might overflow the stack). `std::unique_ptr` has a specialisation to handle arrays, and it looks like this:

```
// make a `std::unique_ptr` to an array of 100 ints
std::unique_ptr<int[]> up = std::make_unique<int[]>(100);

up[0] = 42;  // access elements via `operator[]`

int* raw_buf = up.get();  // get the raw pointer
```

Snippet 3: `std::unique_ptr` with an array

2.1.2 Custom deleter for `std::unique_ptr`

`std::unique_ptr` has a `Deleter` template parameter that allows users to customise what destroying the `std::unique_ptr` should do to the contained pointer. By default, this does `delete` (or `delete[]` for arrays), which is usually what we want to do.

For example, if our objects were taken from a custom memory pool, we might want to return them to the pool instead of `delete`-ing them:

```
int f() {
  std::unique_ptr<int, decltype([](int* ptr) {
                    return_to_pool(ptr);  //
                  })>
      up;
  /* do stuff with ptr */
}  // <-- automatically calls return_to_pool(ptr)
```

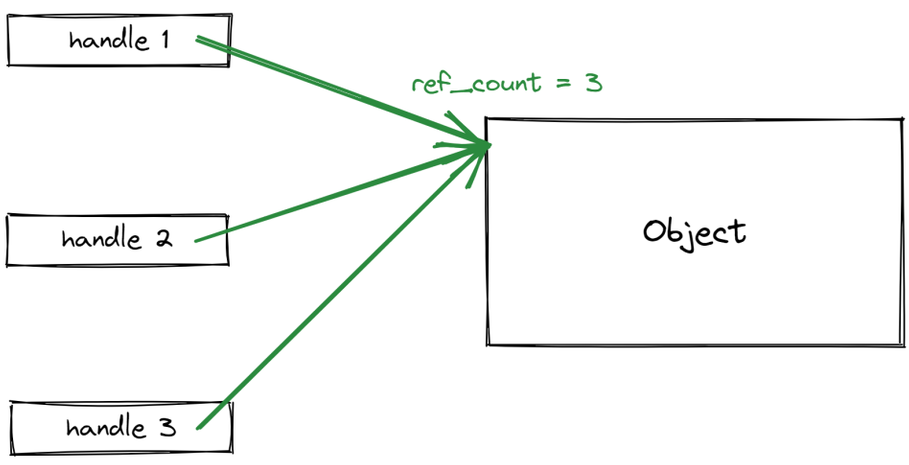
Snippet 4: Example of a custom deleter for `std::unique_ptr`

Other situations include wanting to call `fclose` on a `FILE*`, or interacting with shared libraries that provide a function to free returned objects.

2.2 Shared ownership with reference counting

In most situations we are able to assign a unique owner to every heap allocation, and thereby ensure that the heap allocation will be freed by that unique owner. However, for some situations finding a clear owner isn't always possible, in which case we may have to resort to more complicated ownership semantics.

Reference counting is the idea of storing a count of the number of references (i.e. pointers) to a shared resource (in this case, a block of heap memory):

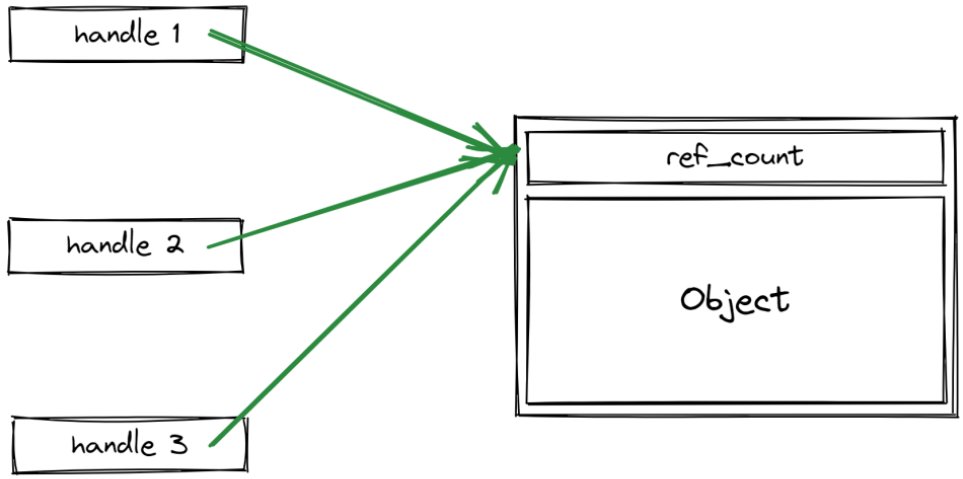


Reference counting

This reference count is stored somewhere accessible by all the handles. We can create new handles from an existing handle, which will point to the same object and increment the reference count.

All these handles collectively *share ownership* of the heap object, so they are responsible for destroying the object and freeing the heap memory. When a handle is destructed, it decrements the reference count. But if it is the last handle (by checking if the reference count is `1`), it must also destroy the object and free the heap memory.

So a shared pointer should behave like a normal pointer, except that we need to store this reference count somewhere. Where? All the handles need to modify the same reference count, so it must be stored somewhere accessible to all of them. The natural place to store it is within the same heap memory used to store the object itself:



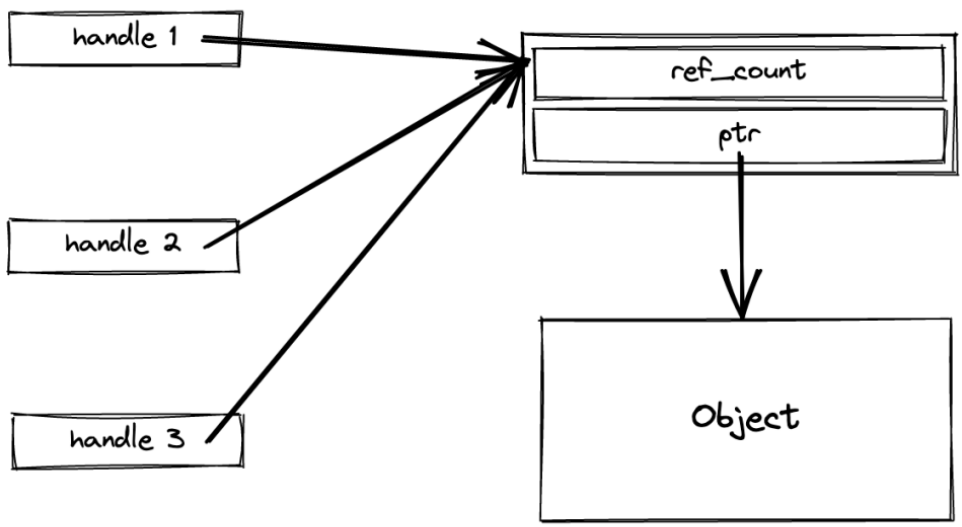
Possible way to store reference count in the same heap allocation as the object

SimpleSharedPtr demo

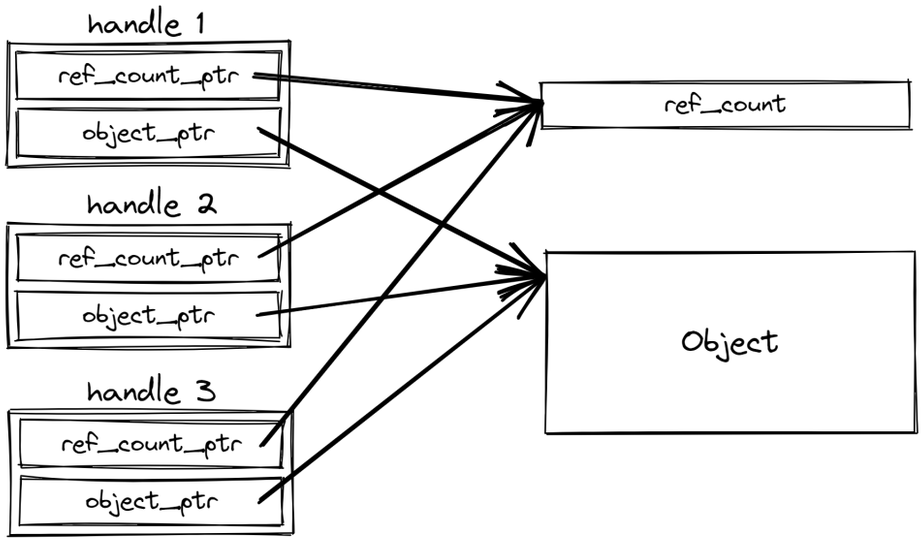
There is, however, one somewhat major limitation of this design. Since the object resides in the same heap allocation as the reference counter, it isn't possible to make a shared pointer out of a previously-allocated object. In other words, the following constructor is unimplementable without copying the object into a new heap allocation:

```
SimpleSharedPtr(T* ptr) {
  // how to implement?
}
```

There are a few ways to allow construction from an existing object, but they all need at least two heap allocations:



Storing the object separately with an additional dereference



Using a fat handle that contains two separate pointers

2.2.1 Weak references

Sometimes, there may be situations where we end up with cyclic references. For example, we might want an object to invoke a callback to a class that owns it:

```
class ButtonPressedDelegate {
  virtual void onButtonPressed() = 0;
};
class Button {
  SimpleSharedPtr<ButtonPressedDelegate> delegate;

  Button(SimpleSharedPtr<ButtonPressedDelegate> delegate)
      : delegate(delegate) {}

  void run() {
    // eventually this calls the delegate,
    // when the button is pressed
    delegate->onButtonPressed();
  }
};
class Window : public ButtonPressedDelegate {
  SimpleSharedPtr<Button> button(this);
};
```

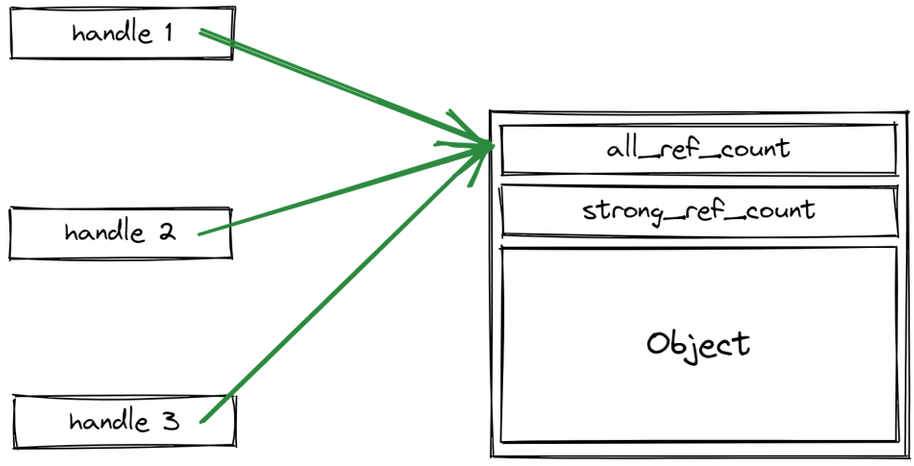
Snippet 5: An example of a cyclic reference

A cyclic reference means that the reference counts of objects in the cycle will never reach zero, even if they are no longer reachable from objects outside the cycle — resulting in a memory leak.

To break this cycle, at least one of the shared pointers must be *weak* — it must not increment the reference count, so that the cycle can be broken.

A weak reference is one that could refer to either the same object referred to by a shared pointer (i.e. a strong reference), or some null value that indicates that the referent has already been destroyed. Obtaining a strong reference from a weak reference is thus a fallible operation — it needs to check if the object is still alive before returning the reference.

For weak references to work, the shared pointer type must implement support for it. The typical way to implement weak references is by having two reference counts:



Two reference counts to implement weak references

`strong_ref_count` is the number of strong references (i.e. regular shared pointers) to the object, while `all_ref_count` counts both the strong and weak references. When the `strong_ref_count` reaches zero, the object is destroyed; when `all_ref_count` is zero, there are no more references to the reference counts, and so the heap memory is freed.

For shared pointer implementations where the object resides in a separate heap allocation from the reference counts, then the object can be freed once `strong_ref_count` reaches zero. The heap allocation containing the ref counts, known as the control block, is freed once `all_ref_count` reaches zero. Note that this can mean the heap memory sticks around for longer than you might expect, and is only freed once *all* pointers — weak and strong — are destructed.

Note: In languages with automatic reference counting (e.g. Swift), references are strong by default, and the language often includes a keyword (e.g. `weak`) to declare a weak reference.

2.2.2 `std::shared_ptr` and `std::weak_ptr`

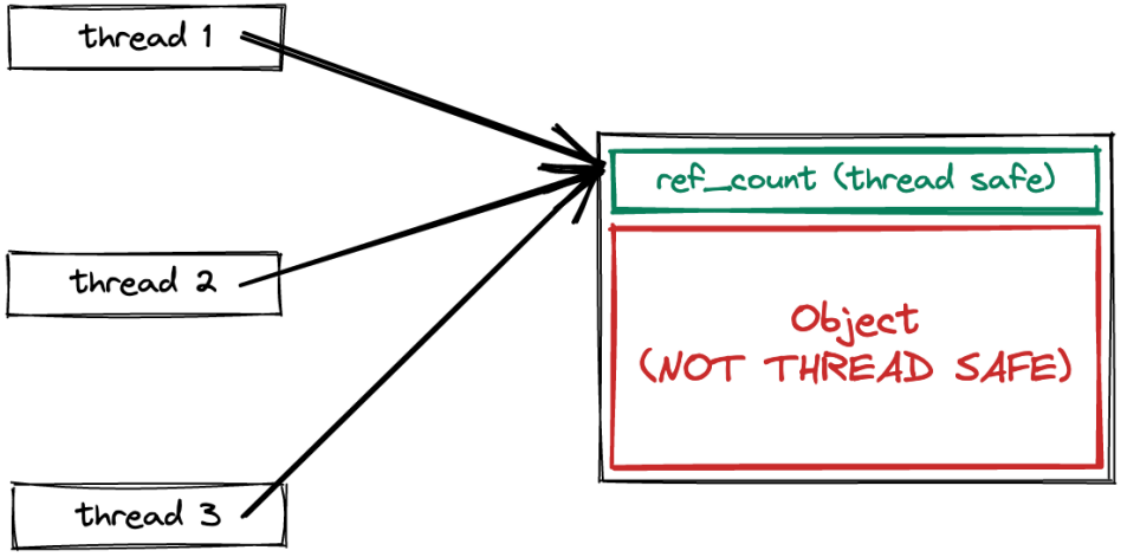
The standard library provides `std::shared_ptr` and `std::weak_ptr`, implementing strong references and weak references respectively.

```
// constructs a new object in a shared_ptr
std::shared_ptr<MyClass> sp1 = std::make_shared(args...);

// copies the shared_ptr; both `sp1` and `sp2`
// are strong references to the same object
std::shared_ptr<MyClass> sp2 = sp1;

// constructs a weak_ptr out of a shared_ptr,
// `wp` is a weak reference to the same object
std::weak_ptr<MyClass> wp = sp2;

// gets a shared_ptr to the same object if the object
// is still alive, otherwise returns a shared_ptr containing nullptr
std::shared_ptr<MyClass> sp3 = wp.lock();
```

Snippet 6: Example use of `std::shared_ptr` and `std::weak_ptr`

These smart pointers also implement an optimisation where it will choose to use either a single heap allocation (containing both the object and the control block) or two heap allocations, depending on how it is initialised.

The control block of these smart pointers is thread-safe — it is possible to have `std::shared_ptr`s from different threads refer to the same object. However, whether concurrent access or mutation of the heap object is actually allowed depends on whether the object is thread safe:



Thread safety of `std::shared_ptr` only applies to the reference count!

Just as `std::shared_ptr` and `std::weak_ptr` are pure library classes, you can implement your own ownership model by designing your own classes if the standard implementations don't fit your needs for whatever reason.

3 Value semantics

So far, we've talked about the role of clear ownership semantics in class design, and how one can use `std::unique_ptr` or `std::shared_ptr` to make it easier to manage the ownership of objects. Having clear ownership of resources is something that is required when designing any class that manages resources, but it isn't the only thing to look out for. For example, even though `std::unique_ptr` is responsible for freeing the heap memory, it meant to feel like a "pointer to an object" rather than "containing the object".

In many situations, it is possible to confer classes with semantics that hide the entire heap allocation, and make the object act as if it were self-contained. For example, we designed our `SimpleString` class to ensure that every instance of that class owned its own buffer, and more than that — we wrote copy and move constructors and assignment operators so that copying and reassignment of strings duplicated the underlying buffer. We did all these so that `SimpleString` "feels intuitive" to the user, the rule of 3 and the rule of 5 hint at this intuitiveness. In this section, we'll formalise this "feels intuitive" notion — this is known as *value semantics*, and a type that satisfies value semantics is known as a *value type*.

3.1 Brief concept of value semantics

A value type behaves like a "value". Each instance of such a type has essentially all of its meaning derived from its value (which it stores), and copying such an object copies its value. Usually, value-semantic objects will also implement an `operator==` that compares equality of values. Such a type is very "mathematically pure", and hence is easy to understand and use.

All the fundamental integer types in C++ are value types, and their values are the integers that they represent. Most types that do not own any resources are also value types (since they often will be value types without any extra work on the part of the programmer). But the interesting part comes when we want to make a value type that needs to allocate heap memory, such as `SimpleString`. (And that's what people are usually concerned about when discussing value semantics).

3.2 Values, objects, and representations

A *value* is an abstract mathematical concept. They are things like `5`, `100`, `"hello world"`, `{1, 3, 6}`, or `{2, 4, 6, 8, 10}`. They are idealised mathematical types, such as numbers, strings (sequences of characters), sets, and lists. We don't talk about modifying values, because they are all immutable. For example, adding `12` to the back of the list `{2, 4, 6, 8, 10}` yields a *new* list `{2, 4, 6, 8, 10, 12}`.

An *object* is an instance of a type. It has an address, and a size — it is a concrete thing that exists in the memory of the program. They're the boxes in our box-and-pointer diagrams. Each object in C++ has a lifetime — its lifetime starts when it is constructed, and ends when it is destructed. They are instances of types like `int`, `std::string`, `std::vector<int>`, or `std::set<int>`.

A value-semantic object represents a value. The *representation* (or *value representation*) is the bits inside the object that represents the value.

Consider the following line of code that declares and initialises a variable:

```
int x = 5;
```

`x` is an object of type `int`. Its value is the mathematical integer `5`. On x86, its bit representation is `00000000000000000000000000000101` in binary, stored in little endian.

We can reassign a value type, for example by doing the following:

```
x = 6;
```

The object `x` still remains. However, we have changed the value stored in the object, and `x`'s value is now 6, and its representation is now `00000000000000000000000000000110` in binary.

3.3 The key idea

The key idea in value semantics is that operations on value types should only depend on the value that the objects represent.

For example, if the object `x` has value `5` and the object `y` has value `8`, then `x + y` will have the value `13`, which is the sum of `5` and `8`. This operation will yield the same result on *any* object with value `5` and *any* object with value `8`, regardless of the current state of the program.

As another example, `x = y` replaces the value of `x` with the value of `y`, so that `x` and `y` now both have the same value.

Similarly, `x += y` multiplies the value of `x` with the value of `y`, and stores the resulting value in `x`. The entire meaning of a value-semantic object is contained in its value.

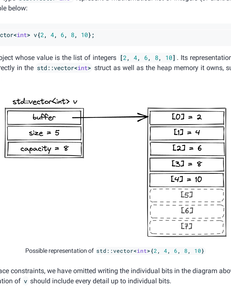
All these operations do what we expect because of the way the compiler implements `int` and their operations. In the `x + y` example above, `x` has representation `000...000101` in binary, `y` has representation `000...001000` in binary, and the `add <x> <y>` instruction is emitted to produce the representation of the resulting object with value `x + y`. This may all seem trivial for `int`, but for more complicated objects their operations will be more complicated.

3.4 A more complicated value type — `std::vector`

Objects of type `std::vector<int>` represent a mathematical list of integers (of arbitrary size). In the example below:

```
std::vector<int> v{2, 4, 6, 8, 10};
```

`v` is an object whose value is the list of integers `[2, 4, 6, 8, 10]`. Its representation is the bit pattern directly in the `std::vector<int>` struct as well as the heap memory it owns, such as the following:

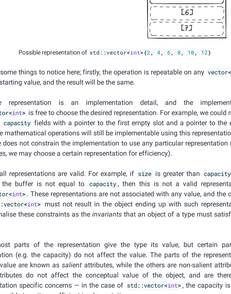


Possible representation of `std::vector<int>{2, 4, 6, 8, 10}`

Due to space constraints, we have omitted writing the individual bits in the diagram above, but the representation of `v` should include every detail up to individual bits.

With this representation, we can implement operations on the values of `v` (i.e. we can make syntactical operations like `v.push_back(arg)` mirror the equivalent pure mathematical operation).

For example, `v.push_back(arg)` performs the operation $v \leftarrow v + [a]$, where v is the value of `v`, a is the value of `arg`, and $+$ performs list concatenation. Because of our chosen representation of `std::vector<int>`, the operation performed in code copies the bits of `arg` to the next available slot in the buffer, and increments the `size` field (and if there is not enough capacity, a reallocation occurs). The resultant value of `v` is then the list of integers `[2, 4, 6, 8, 10, 12]`, and has the following representation:



Possible representation of `std::vector<int>{2, 4, 6, 8, 10, 12}`

There are some things to notice here; firstly, the operation is repeatable on any `vector<int>` with the same starting value, and the result will be the same.

Next, the representation is an implementation detail, and the implementation of `std::vector<int>` is free to choose the desired representation. For example, we could replace the `size` and `capacity` fields with a pointer to the first empty slot and a pointer to the end of the buffer. The mathematical operations will still be implementable using this representation. Being a value type does not constrain the implementation to use any particular representation (though in some cases, we may choose a certain representation for efficiency).

Also, not all representations are valid. For example, if `size` is greater than `capacity`, or if the length of the buffer is not equal to `capacity`, then this is not a valid representation of a `std::vector<int>`. These representations are not associated with any value, and the correct use of a `std::vector<int>` must not result in the object ending up with such representations. We often formalise these constraints as the *invariants* that an object of a type must satisfy (for it to be valid).

Finally, most parts of the representation give the type its value, but certain parts of the representation (e.g. the capacity) do not affect the value. The parts of the representation that affect its value are known as *salient* attributes, while the others are non-salient attributes. Non-salient attributes do not affect the conceptual value of the object, and are there only for implementation specific concerns — in the case of `std::vector<int>`, the capacity is stored so that it is possible to write an efficient implementation.

3.5 Operations expected on any value type

While a value type can have operations specific to their value domain, such as `push_back`, there are some operations that almost all value types are expected to have, and there are expectations as to how these operations behave.

The equality operator (`operator==`) should return whether the values of the two arguments are equal in the mathematical sense. For example:

```
T x = /* some value */;
T y = /* same value but perhaps obtained in another way */;
assert(x == y);
assert(y == x);
```

The usual properties of an equivalence relation — reflexivity, symmetry, and transitivity — should be satisfied, whether for value types or non-value types as long as `operator==` is defined.

Copy construction and copy assignment should result in the current object having the value of the other object. For example:

```
T x = /* stuff */;
T y = x;
assert(x == y);
```

Move construction and move assignment should result in the current object having the original value of the other object. For example:

```
T x = /* stuff */;
T y = x;
T z = std::move(y);
assert(x == z);
```

Operations on an object should not change the value of other objects. An object should contain its separate copy of its entire value representation (except in very specific cases such as when performing copy-on-write optimisations).

3.6 Value semantics and inheritance

Value semantics is often at odds with public inheritance.

Recall how in Lecture 2 we introduced the `Animal` inheritance hierarchy, and then talked about a `Point` class. The `Point` class is trivially a value type, but you'll find it difficult, or perhaps impossible, to make `Animal` (and its derived classes `Cat` and `Dog`) satisfy value semantics.

The reason for this is that when we have an inheritance hierarchy, the *identity* of an object (i.e. the current address of the object) becomes important. The fact that two different `Animal` objects contain the same attributes do not make them intuitively "equal", because when talking about equality for such objects we intuitively mean *identity* (i.e. whether two `Animal` references/pointers refer to the same `Animal` object). We also can't freely copy or move `Animal`s around in general, because that will result in *slicing*, which we talked about in Lecture 2.

Alternatively, we could say that for objects in an inheritance hierarchy, the *identity* of the object is salient, and that is why value semantics cannot coexist with inheritance.

(Note: We are only talking about public inheritance here, since *private* inheritance in C++ is usually used as an implementation detail of your class, and is not visible to users of your class).

3.7 Summary of value semantics

Value types represent pure mathematical objects, and this makes them very intuitive to reason about. Where possible, we should design types that satisfy value semantics.

4 Const-correctness

If you search for this on the internet, you will find that there is no consistent definition of const-correctness.

The ISO C++ website simply says that you should put `const` on arguments that are references if you don't intend to modify them, and omit `const` if it might be modified. For example, they say that if you have a function `f` that takes in a `SimpleString` and does not modify it, you should pass it by const reference:

```
void f_good(const SimpleString& s) {
  // do stuff that don't modify s
}
```

Snippet 7: Pass by const reference (good)

```
void f_bad(SimpleString& s) {
  // do stuff that don't modify s
}
```

Snippet 8: Pass by non-const reference (bad)

Doing this has two benefits: it serves as documentation about whether the parameter might be modified, and also allows callers with const references pass them into your function easily:

```
void g(const SimpleString& s) {
  f_good(s);  // Okay

  // Compile time error, even though f_bad
  // doesn't actually modify s
  // f_bad(s);
}
```

Snippet 9: Calling functions with a const reference

But this merely passes the buck to the implementer of `SimpleString`. For example, if we did `s[0] = 'a'`, is this considered modifying `s`? Most people will say it does modify `s`. But recall that `SimpleString` is really only a size and a pointer to buffer, and neither of these fields were modified:

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

Snippet 10: Fields of `SimpleString`

The answer to this question is decided by the implementer of `SimpleString`. Remember that we wrote two `operator[]` overloads for `SimpleString`:

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

  const char& operator[](size_t index) const {  // <-- const overload
    return m_buf[index];
  }
  char& operator[](size_t index) {  // <-- non-const overload
    return m_buf[index];
  }
};
```

Snippet 11: `operator[]` for `SimpleString` (good)

The code would also have compiled if we simply wrote this one overload:

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

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

Snippet 12: `operator[]` for `SimpleString` (bad)

It compiles because `m_buf` is just a pointer, and mutating the contents of the buffer doesn't mutate the pointer itself.

We then said that having the two overloads is good, because we're really writing two separate interfaces — one for (non-const) `SimpleString`, and one for `const SimpleString`. But really, what is wrong with the second version?

You should feel that the first version reinforces to users the intuition that the contents of the string is part of the `SimpleString` object itself, instead of `SimpleString` holding a pointer to some external memory. And, as the implementer of `SimpleString`, you should furthermore recognise that that is precisely the intuition we want to give users. What exactly gives rise to such intuition? Well, because the contents of the string is part of its *value*!

4.1 Const-correctness of value types

For value types, we have a way to define const-correctness: a const object should not change value. In other words, a member function should be const if it doesn't change the value of the object, and be non-const otherwise.

Since `SimpleString` is a value type and its value is the list of characters that make up the string, mutating the characters in the buffer should be treated as a non-const operation on the string. This is why writing the const and non-const versions of `operator[]` that respectively return a const and non-const `char&` is const-correct:

```
const char& operator[](size_t index) const;
char& operator[](size_t index);
```

Snippet 13: The two `operator[]` overloads for `SimpleString`

Writing two almost-identical functions that differ in const-ness is rather common in C++, and so there is a term for this — *const propagation* (not to be confused with constant propagation, which is a compiler optimisation). It is unfortunate that there currently is no shorthand to generate both the const and non-const versions of a member function.

We have seen another example of const propagation in Lecture 5 — const and non-const iterators. As iterators permit access and possibly modification of the elements that it iterates over, to ensure const-correctness of a container there needs to be two versions of each iterator that the container provides: the non-const iterator, through which users can both access and modify the elements of the container (i.e. `*it` returns a non-const reference); and the const iterator, through which only access is permitted but not modification (i.e. `*it` returns a const reference). This is why the `begin`/`end` member functions perform const propagation:

```
const_iterator begin() const;  // const version
iterator begin();              // non-const version

// convenience function to get a const iterator
// from a possibly non-const container
const_iterator cbegin() const;
```

Snippet 14: The two `begin` overloads for all standard library containers

With const-propagating iterators, the container author ensures that it isn't possible (barring `const_cast`) for a user to change the value of a const container:

```
const IntContainer& c = /* stuff */;

//    v-- `it` has type Container::const_iterator
for (auto it = c.begin(); it != c.end(); ++it) {
  int out = *it;  // okay, can read the element at the iterator
  // *it = 100;   // compile error, since `it` is a const reference
}

for (auto& x : c) {
  // `x` is a const reference
  int out = x;  // okay
  // x = 100;   // compile error
}

// compile error, since `std::sort` needs
// non-const iterators to swap elements
// std::sort(c.begin(), c.end());
```

Snippet 15: Using const and non-const iterators

For non-value types, we don't have a good definition for const-correctness — just try to do the "intuitive" thing.

4.2 Mutable fields

Even though the *value* of a type should be independent from its representation, we sometimes want to be able to modify the internal representation (as an implementation detail) of the value even if we're only performing a const operation.

One possible scenario is when implementing a cache; in theory, read operations should not require a non-const reference to the cache object, but we would also like it to *cache* the read value (obviously), for performance reasons.

We might then implement something like this:

```
struct Cache {
  int read_value(const std::string& key, int otherwise) const {
    if (auto it = values.find(key); it != values.end()) {
      return it->second;
    } else {
      return values[key] = otherwise;
    }
  }
  mutable std::map<std::string, int> values;
  // other fields
};
```

Snippet 16: An example of using a `mutable` field

Here, because the `values` field was marked `mutable`, we can still modify it through a const reference, or in this case, through a const-qualified member function.

Another prominent example of where `mutable` is often used is for internally synchronised classes that contain some kind of mutex — since the lock/unlock methods of a `std::mutex` are non-const, the mutex field is usually marked `mutable` so that const methods can still synchronise correctly.

5 Additional topics (self study)

- How does `std::unique_ptr` fail to satisfy value semantics? How about `OwnedInt` (Lecture 3)? Can make a generic `Owned<T>`, and have it also support arrays? (Note: This will be similar to Rust's std::boxed::Box).

© 19 July 2022, Bernard Teo Zhi Yi, All Rights Reserved