CS100 Lecture 19

Contents

Smart Pointers

C++ standard library `<memory>` provides smart pointers for better management of dynamic memory.

- Raw pointers require a manual `delete` call by users. Need to use with caution to avoid memory leaks or more severe errors.
- Smart pointers automatically dispose of the objects pointing to.

Smart pointers support same operations as raw pointers: dereferencing `*`, member access `->`...
Use smart pointers as a substitute to raw pointers.

Smart Pointers

`<memory>` provides two types of smart pointers:

- `std::unique_ptr<T>`, which uniquely **owns** an object of type `T`.
  - No other smart pointer pointing to the same object is allowed.
  - Disposes of the object (calls its destructor) once this `unique_ptr` gets destroyed or assigned a new value.
- `std::shared_ptr<T>`, which **shares** ownership of an object of type `T`.
  - Multiple `shared_ptr`s pointing to a same object is allowed.
  - Disposes of the object (calls its destructor) when the last `shared_ptr` pointing to that object gets destroyed or assigned a new value.

Using Smart Pointers

Note the `<T>` in smart pointers: they are templates, like `std::vector`. `T` indicates the type of their managed objects:

- `std::unique_ptr<int> pi;` points to an `int`, like a raw pointer `int *`.
- `std::shared_ptr<std::vector<double>> pv;`, like an `std::vector<double> *`.

Dereferencing operators `*` and `->` can be used the same way as for raw pointers:

- `*pi = 3;`
- `pv->push_back(2.0);`

`std::unique_ptr`

Creating an `std::unique_ptr`

Use `std::unique_ptr` to create an object in dynamic memory,

- if no other pointer to this object is needed.

Two ways of creating an `std::unique_ptr`:

- passing a pointer created by `new` in the constructor:

```cpp
std::unique_ptr<Student> p(new Student("Bob", 2020123123));
```

- use `std::make_unique<T>`, pass initializers to it:

```cpp
std::unique_ptr<Student> p1 = std::make_unique<Student>("Bob", 2020123123);
auto p2 = std::make_unique<Student>("Alice", 2020321321);
```

Using `auto` here does not reduce readability, because `std::make_unique<Student>` clearly hints the type.

`std::unique_ptr`: Automatic Memory Management

```cpp
void foo() {
  std::unique_ptr pAlice(new Student("Alice", 2020321321));
  // Do something...
  if (some_condition) {
    std::unique_ptr pBob(new Studnet("Bob", 2020123123));
    // Do something...
  } // Destructor ~Student called for Bob, since pBob goes out of scope.
} // Destructor ~Student called for Alice, since pAlice goes out of scope.
```

An `std::unique_ptr` automatically calls the destructor once it gets destroyed or assigned a new value.

- No manual `delete` needed!

`std::unique_ptr`: Move-only

```cpp
auto p = std::make_unique<std::string>("Hello");
std::cout << *p << std::endl; // Prints "Hello".
std::unique_ptr<std::string> q = p; // Error, copy is not allowed.
std::unique_ptr<std::string> r = std::move(p); // Correct.
// The ownership of this std::string is transferred to r.
std::cout << *r << std::endl; // Prints "Hello".
assert(!p); // p is now invalid
```

An `std::unique_ptr` cannot be copied, but only moved.

- Remember, only one `std::unique_ptr` can own the managed object.
- A move operation transfers its ownership.

Move assignment

`std::unique_ptr` is only move-assignable, not copy-assignable.

```
std::unique_ptr<T> p = some_value(), q = some_other_value();
p = q; // Error
p = std::move(q); // OK.
```

The assignment `p = std::move(q)` does the following:

- `p` releases the object it used to manage. Destructor is called and memory is deallocated.
- Then, the object that `q` manages is transferred to `p`. `q` no longer owns an object.

Returning a `unique_ptr`

```
std::unique_ptr<bf_state> bf_state_create() {
  auto s = std::make_unique<bf_state>(...);
  // ...
  return s; // move
}
std::unique_ptr<bf_state> state = some_value();
state = bf_state_create(); // move-assign
```

A temporary is move-initialized from `s`, and then move-assigned to `state`.

- This move-assignment makes `state` dispose of its original object, calling the destructor.

`unique_ptr` for array type

By default, the destructor of `std::unique_ptr<T>` uses a `delete` expression to destroy the object it holds.

What happens if `std::unique_ptr<T> up(new T[n]);`?

`unique_ptr` for array type

By default, the destructor of `std::unique_ptr<T>` uses a `delete` expression to destroy the object it holds.

What happens if `std::unique_ptr<T> up(new T[n]);`?

- The memory is obtained using `new[]`, but deallocated by `delete`! - Undefined behavior.

`unique_ptr` for array type

A *template specialization*: `std::unique_ptr<T[]>`

- Specially designed to represent pointers that point to a "dynamic array" of objects.
- Has some array-specific operators, e.g. `operator[]`.
- Does not support `operator*` and `operator->`.
- Uses `delete[]` instead of `delete`.

```cpp
auto up = std::make_unique<int[]>(n);
std::unique_ptr<int[]> up2(new int[n]{}); // equivalent
for (int i = 0; i != n; ++i)
  std::cout << up[i] << ' ';
```

`unique_ptr` for array type

When you want to have "an array of things":

- `std::unique_ptr<T[]>` manages the memory well,
- **but STL containers do a better job!**

`std::shared_ptr`

Motivation

A `unique_ptr` uniquely owns an object, but sometimes this is not convenient:

```cpp
std::vector<Object *> objects;
Object *get_object(int i) {
  return objects[i];
}
```

```cpp
std::vector<unique_ptr<Object>> objects;
unique_ptr<Object> get_object(int i) {
  return objects[i]; // Error
}
```

We want to design a smart pointer (let's call it `SharedPtr`) that allows the object it manages to be *shared*.

- A `unique_ptr` destroys the object it manages when the pointer itself is destroyed.
- If we allow many `SharedPtr`s to point to the same object, how can we know when to destroy that object?

Idea: Reference counting

Set a **counter** that counts how many `SharedPtr`s are pointing to it:

```
struct CountedObject {
  Object the_object;
  int ref_cnt = 1;
};
```

When a new object is created by a `SharedPtr`, set the `ref_cnt` to `1`.

Idea: Reference counting

When a `SharedPtr` is copied, let them point to the same object, and increment the counter.

```
class SharedPtr {
  CountedObject *m_ptr;
 public:
  SharedPtr(const SharedPtr &other)
      : m_ptr(other.m_ptr) { ++m_ptr->ref_cnt; }
};
```

Idea: Reference counting

For copy assignment: the counter of the old object should be decremented.

- If it reaches zero, destroy that object!

```cpp
class SharedPtr {
  CountedObject *m_ptr;
 public:
  SharedPtr &operator=(const SharedPtr &other) {
    if (--m_ptr->ref_cnt == 0)
      delete m_ptr;
    m_ptr = other.m_ptr;
    ++m_ptr->ref_cnt;
    return *this;
  }
};
```

*** Is this correct?**

Idea: Reference counting

Self-assignment safe!!!

```cpp
class SharedPtr {
  CountedObject *m_ptr;
 public:
  SharedPtr &operator=(const SharedPtr &other) {
    ++other.m_ptr->ref_cnt;
    if (--m_ptr->ref_cnt == 0)
      delete m_ptr;
    m_ptr = other.m_ptr;
    return *this;
  }
};
```

Idea: Reference counting

Destructor: decrement the counter, and destroy the object if the counter reaches zero.

```cpp
class SharedPtr {
  CountedObject *m_ptr;
 public:
  ~SharedPtr() {
    if (--m_ptr->ref_cnt == 0)
      delete m_ptr;
  }
};
```

Idea: Reference counting

Move: Just *steal* the object.

```
class SharedPtr {
  CountedObject *m_ptr;
 public:
  SharedPtr(SharedPtr &&other) noexcept
      : m_ptr(other.m_ptr) { other.m_ptr = nullptr; }
  SharedPtr &operator=(SharedPtr &&other) noexcept {
    if (this != &other) {
      if (--m_ptr->ref_cnt == 0)
        delete m_ptr;
      m_ptr = other.m_ptr; other.m_ptr = nullptr;
    }
    return *this;
  }
};
```

`std::shared_ptr`

A smart pointer that uses **reference counting** to manage shared objects.

Create a `shared_ptr`:

```
std::shared_ptr<Type> sp2(new Type(args));
auto sp = std::make_shared<Type>(args); // equivalent, but better
```

For example:

```
auto sp = std::make_shared<std::string>(10, 'c');
// sp points to a string "cccccccccc".
```

Create a `shared_ptr`

Note: For `std::unique_ptr`, both of the following ways are ok (since C++17):

```
auto up = std::make_unique<Type>(args);
std::unique_ptr<Type> up2(new Type(args));
```

But for `std::shared_ptr`, **`std::make_shared` is preferable to directly using `new`**.

```
auto sp = std::make_shared<Type>(args); // preferred
std::shared_ptr<Type> sp2(new Type(args)); // ok, but less preferred
```

Read *Effective Modern C++* Item 21. (Note that this book is based on C++14.)

Operations

`*` and `->` can be used as if it is a raw pointer:

```
auto sp = std::make_shared<std::string>(10, 'c');
std::cout << *sp << std::endl; // cccccccccc
std::cout << sp->size() << std::endl; // 10
```

`sp.use_count()`: The value of the reference counter.

```
auto sp = std::make_shared<std::string>(10, 'c');
{
  auto sp2 = sp;
  std::cout << sp.use_count() << std::endl; // 2
} // sp2 is destroyed
std::cout << sp.use_count() << std::endl; // 1
```

Operations

Full list of members: for shared_ptr, for unique_ptr

Some functions that may be useful: `reset()`, `release()`, ...

Notes:

- Both of them support `.get()`, which returns a raw pointer to the managed object.
  - This is useful when some old interfaces only accept raw pointers, e.g. `glfwSwapBuffers`.
- Be careful! Mixing the usage of raw pointers and smart pointers can lead to disaster!

```cpp
delete up.get(); // disaster
```

Deleters and allocators

Customized *deleters* are supported on `unique_ptr` and `shared_ptr`.

`shared_ptr` also allows customized allocators.

We will talk about how to define such things in later lectures.