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:

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

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

```
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

```
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.

`std::shared_ptr`