ADVANCED PROGRAMMING IN C++

Smart Pointers

Patrick Bader · SS 2023

Smart Pointers

Tie resource lifetime (e.g. heap allocated objects) to local (stack allocated) objects (RAII)

Solve the resource management problem generically for different use cases

Provide exception safety

Mimick interface of raw pointers

Smart Pointers - Idea

Store pointer in class and delete pointee when pointer goes out of scope:

```
struct SmartWidgetPointer
{
  Widget* m_widget;

  explicit SmartWidgetPointer(Widget* widget)
    : m_widget(widget) {

  }

  ~SmartWidgetPointer() {
    delete m_widget;
  }
};
```

Smart Pointers - Scoped Pointer

Prevent copying so resource is not deleted twice:

```
struct SmartWidgetPointer
{
  Widget* m_widget;

  explicit SmartWidgetPointer(Widget* widget)
    : m_widget(widget) {

  }

  SmartWidgetPointer(const SmartWidgetPointer&) = delete;
  SmartWidgetPointer& operator=(const SmartWidgetPointer&) = delete;

  ~SmartWidgetPointer() {
    delete m_widget;
  }
};
```

Smart Pointers - Scoped Pointer

Provide pointer-like interface:

```
struct SmartWidgetPointer
{
  Widget* m_widget;

  explicit SmartWidgetPointer(Widget* widget)
    : m_widget(widget) {
  }

  SmartWidgetPointer(const SmartWidgetPointer&) = delete;
  SmartWidgetPointer& operator=(const SmartWidgetPointer&) = delete;

  Widget* operator->() const { // provide pointer-like member access
    return m_widget;
  }

  Widget& operator*() const { // provide pointer-like dereferencing
    return *m_widget;
  }

  Widget* get() const { // prevent internal pointer from being directly modified
    return m_widget;
  }

  ~SmartWidgetPointer() {
    delete m_widget;
  }
};
```

Smart Pointers

How to generalize to arbitrary objects?

–> Make the SmartWidgetPointer a class template!

Smart Pointers - Scoped Pointer

Provide pointer-like interface:

```
struct SmartWidgetPointer
{
  Widget* m_widget;

  explicit SmartWidgetPointer(Widget* widget)
    : m_widget(widget) {
  }

  SmartWidgetPointer(const SmartWidgetPointer&) = delete;
  SmartWidgetPointer& operator=(const SmartWidgetPointer&) = delete;

  Widget* operator->() const { // provide pointer-like member access
    return m_widget;
  }

  Widget& operator*() const { // provide pointer-like dereferencing
    return *m_widget;
  }

  Widget* get() const { // prevent internal pointer from being directly modified
    return m_widget;
  }

  ~SmartWidgetPointer() {
    delete m_widget;
  }
};
```

Smart Pointers - Scoped Pointer

Generalize to arbitrary objects:

```
template<typename T>
struct SmartPointer
{
  T* m_ptr;

  explicit SmartPointer(T* ptr)
    : m_ptr(ptr) {
  }

  SmartPointer(const SmartPointer&) = delete;
  SmartPointer& operator=(const SmartPointer&) = delete;

  T* operator->() const { // provide pointer-like member access
    return m_ptr;
  }

  T& operator*() const { // provide pointer-like dereferencing
    return *m_ptr;
  }

  T* get() const { // prevent internal pointer from being directly modified
    return m_ptr;
  }

  ~SmartPointer() {
    delete m_ptr;
  }
};
```

Smart Pointers - Unique Pointer

Allow moving the smart pointer around:

```
template<typename T>
struct SmartPointer
{
  T* m_ptr;

  explicit SmartPointer(T* ptr)
    : m_ptr(ptr) {
  }

  SmartPointer(SmartPointer&&); // allow move construction
  SmartPointer& operator=(SmartPointer&&); // allow move assignment

  T* operator->() const {
    return m_ptr;
  }

  T& operator*() const {
    return *m_ptr;
  }

  T* get() const {
    return m_ptr;
  }

  ~SmartPointer() {
    delete m_ptr;
  }
};
```

Smart Pointers - Unique Pointer

Allow moving the smart pointer around:

```
// move constructor:
template<typename T>
SmartPointer<T>::SmartPointer(SmartPointer<T>&& rhs) {
  m_ptr = rhs.m_ptr;
  rhs.m_ptr = nullptr;
}
// move assignment:
template<typename T>
SmartPointer<T>& SmartPointer<T>::operator=(SmartPointer<T>&& rhs)
{
  if (this != &rhs) { // check for self-assignment
    if (m_ptr) {
      delete m_ptr;
    }
    m_ptr = rhs.m_ptr;
    rhs.m_ptr = nullptr;
  }
  return *this;
}
```

Standard Library - Unique Pointer

`std::unique_ptr` Header: `#include <memory>`

Implements a smart pointer for non-shared objects

Only one pointer may point to the same object at a time

Move operations are allowed

Copies are prohibited

Example:

```
std::unique_ptr<Widget> ptr(new Widget());
// Better:
auto ptr = std::make_unique<Widget>();
```

Documentation: http://en.cppreference.com/w/cpp/memory/unique_ptr

Standard Library - Unique Pointer - Example

```
struct Widget {
  std::string name;
  Widget(const std::string& name)
    : name(name) {
    std::cout << name.c_str() << " constructor\r\n";
  };
  ~Widget() {
    std::cout << name.c_str() << " destructor\r\n";
  };
};

void makeUniqueWidgetAndThrow() {
  auto functionLocal = std::make_unique<Widget>("functionLocal");
  throw std::exception();
}

int main() {
  auto outer = std::make_unique<Widget>("outer");
  {
  auto inner = std::make_unique<Widget>("inner");
  }
  makeUniqueWidgetAndThrow();
  return 0;
}
```

Output:

```
outer constructor
inner constructor
inner destructor
functionLocal constructor
functionLocal destructor
outer destructor
```

Smart Pointers - Shared Pointer

Unique pointers are applicable when there is always a **single owner** of the resource

To enforce this, **copy operations** have been **deleted** in our `UniquePointer` implementation.

Although **in some cases** resources may have **multiple owners** and a resource should only be **deleted when** there **is no owner left**.

How to know when there is no owner left? **–> Counting**

Smart Pointers - Shared Pointer

Count the number of shared pointers to one pointee

```
template<typename T>
class SharedPointer {
  T* m_ptr;
  int* m_count;
public:
  SharedPointer(T* ptr)
    : m_ptr(ptr), m_count(new int(1)) { }

  ~SharedPointer() {
    decrementRC();
  }
  SharedPointer(const SharedPointer<T>& other);
  SharedPointer<T>& operator=(const SharedPointer<T>& other);
private:
  void incrementRC() { (*m_count)++; }
  void decrementRC() {
    if (--(*m_count)==0) {
      delete m_ptr;
      delete m_count;
    }
  }
};
```

Smart Pointers - Shared Pointer

Implement copy operations to increment/decrement reference counts:

```
template<typename T>
SharedPointer<T>::SharedPointer(const SharedPointer<T>& other) {
  m_count = other.m_count;
  m_ptr = other.m_ptr;
  incrementRC();
}

template<typename T>
SharedPointer<T>& SharedPointer<T>::operator=(const SharedPointer<T>& other) {
  if (this != &other) {
    decrementRC();
    m_count = other.m_count;
    m_ptr = other.m_ptr;
    incrementRC();
  }
  return *this;
}
```

Standard Library - Shared Pointer

`std::shared_ptr` Header: `#include <memory>`

Implements a smart pointer for shared objects

Multiple pointers may point to the same object at a time

The object is destroyed when no shared pointer references it anymore (implemented with reference counting)

Example:

```
std::shared_ptr<Widget> ptr(new Widget());
// Better:
auto ptr = std::make_shared<Widget>();
```

Documentation: http://en.cppreference.com/w/cpp/memory/shared_ptr

Standard Library - Shared Pointer - Limitations

Beware: Shared pointers **cannot handle** reference **cycles**

Memory is never freed if a cycle occurs!

Cycles need to be broken up explicitly by the programmer

Standard Library - Weak Pointer

`std::weak_ptr` Header: `#include <memory>`

Holds a non-owning (temporary) reference to an object managed by a shared pointer

Can guarantee that object is only accessed if it still exists, but but does not prevent destruction

A std::weak_ptr must be converted to std::shared_ptr in order to access the referenced object

Can be used to break circular references of shared pointers

Example:

```
auto ptr = std::make_shared<Widget>();
std::weak_ptr<Widget> weak = ptr;
// later when used:
{
  std::shared_ptr<Widget> localPtr = weak.lock();
}
```

Documentation: http://en.cppreference.com/w/cpp/memory/weak_ptr

Smart Pointers - Summary

Don't use pointers when you don't need them!

Don't use raw pointers to represent **ownership**.

Use **`unique_ptr`** when there is a **single owner**.

Use **`shared_ptr`** when there are **multiple owners**.

Use **`weak_ptr`** to **break** ownership **cycles**.

Smart Pointers - When To Use Which Smart Pointer

It’s all about data structures.

For homework: Watch the CppCon 2016 presentation by Herb Sutter: “Leak-Freedom in C++... By Default.” https://www.youtube.com/watch?v=JfmTagWcqoE