UNIVERSITY OF RICHMOND

Smart Pointers

CMSC 240 Software Systems Development

Today – Smart Pointers

- Smart Pointer Introduction
- unique_ptr
- shared_ptr

Dynamic Memory Allocation

- **The problem**: Memory resources are sometimes allocated on the heap, and they must be released at some point

- If we forget, then we have a **memory leak**
  - A long running program with a memory leak will slowly run out of memory, which can kill performance
    - For example: web browsers, text and code editors, web services

Dynamic Memory Allocation

- Dynamically allocating memory is not a problem if you **remember to** deallocate that memory when you are done using that memory

"A solution involving the phrase 'just remember to' is seldom the best solution."

-- Steven Prata (C++ Primer Plus)

Dynamic Memory Allocation

- Consider: memory allocated on the stack is automatically deallocated when it goes out of scope
- Thought: Can we somehow give ownership of a resource allocated dynamically to an object that is deallocated automatically
- If so, the dynamic resource can be released when the owning resource goes out of scope (in destructor call)

Standard Example

```
int leakyFunction()
{
    string* pointerToString = new string("Leak");

    /*  ... some processing ... */


    return 0;
}
```

A more subtle example…

```
int leakyFunction2()
{
    string* pointerToString = new string("Leak");

    /*  ... some processing ... */

    try
    {
        char ch = pointerToString->at(50);
    }
    catch (out_of_range exception)
    {
        cerr << "Caught an out_of_range error: " << exception.what() << endl;
        throw exception;
    }

    delete pointerToString;

    return 0;
}
```

Smart Pointers

- If **`pointerToString`** had a <u>destructor</u>, memory could be released in the destructor automatically when the function returns
- But **`pointerToString`** is just an ordinary pointer, not a class object, so it has no destructor
- If it were an object, then we could code a destructor and the memory would be freed when the object was out of scope after the function returns
- This is the idea behind smart pointers

C++ Smart Pointers

- A smart pointer is an object that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - The smart pointer will delete the pointed-to object at the right time
    - When that is depends on what kind of smart pointer you use

- With correct use of smart pointers, you no longer need to remember when to **`delete`** newly allocated memory!

A Simple Smart Pointer

- We can implement a simple smart pointer with the following:
  - Constructor that accepts a pointer
  - Destructor that deletes the pointer
  - Overload `*` and `->` operators that access the pointer

- A smart pointer is just a C++ Template object

```cpp
template <typename T>
class SimpleSmartPointer
{
public:
    // Constructor will initialize the pointer of type T.
    SimpleSmartPointer(T* ptr) : pointer(ptr) { }

    // Destructor for the simple smart pointer class.
    // Will delete the pointer to free the memory on the heap.
    ~SimpleSmartPointer()
    {
        std::cout << "Deleting pointer..." << std::endl;
        delete pointer;
    }

    // Override the * operator, returns the contents of the pointer.
    T operator*() { return *pointer; }

    // Override the -> operator, returns the pointer.
    T* operator->() { return pointer; }

private:
    // The actual pointer.
    T* pointer;
};
```

A Simple Smart Pointer

- Effectively, a smart pointer is a wrapper for a raw pointer

- Access the encapsulated pointer using the operators **->** and *****, which the smart pointer class overloads so that they return the encapsulated raw pointer

```
void processPointers()
{
    // Create a regular pointer.
    string* leaking = new string("Regular");

    // Create a simple smart pointer.
    SimpleSmartPointer<string> notleaking(new string("Smart"));

    cout << "*leaking == " << *leaking << endl;
    cout << "*notleaking == " << *notleaking << endl;
}

int main()
{
    // Call the processPointers function.
    processPointers();

    // Returned from processPointers function scope.
    cout << "Back in main function." << endl;

    return 0;
}
```

A Simple Smart Pointer

- Can’t handle:
  - Arrays -- (i.e. needs to use `delete[]`)
  - Copying
  - Reassignment
  - Comparison
  - Many other details…

- Luckily, there is a standard library version of smart pointers!
  - `#include <memory>`

Introducing: `unique_ptr`

- A `unique_ptr` is the sole owner of its managed pointer
  - It will call `delete` on the managed pointer when it falls out of scope
  - This is accomplished via the `unique_ptr` destructor
- Guarantees uniqueness by **disabling copy and assignment**

```cpp
#include <iostream>
#include <memory>
using namespace std;

void Leaky()
{
    // A pointer to a heap-allocated integer.
    int* rawPointer = new int(42);

    /*  ... some processing ... */

    cout << *rawPointer << endl;

} // After return: Never used delete, therefore leak.


void NotLeaky()
{
    // A smart pointer wrapped heap-allocated integer.
    unique_ptr<int> smartPointer(new int(25));

    /*  ... some processing ... */

    cout << *smartPointer << endl;

} // After return: Never used delete, but no leak.
```

`unique_ptr` Cannot Be Copied

- **`unique_ptr`** has disabled its copy constructor and assignment operator
  - You cannot copy a **`unique_ptr`**, helping maintain “uniqueness” or “ownership” of the managed pointer

```cpp
#include <memory>
using namespace std;

int main()
{
  ✓ unique_ptr<int> x(new int(5)); // Okay: constructor that takes a pointer
  ⊘ unique_ptr<int> y(x);          // Error: copy constructor is disabled
  ✓ unique_ptr<int> z;             // Okay: default constructor, holds nullptr
  ⊘ z = x;                         // Error: operator= is disabled
    return 0;
}
```

```cpp
#include <memory>
using namespace std;

int main()
{
    // Create a new unique pointer to manage a pointer to a double.
    unique_ptr<double> smartPointer(new double(3.141));

    // Return a pointer to pointed-to object.
    double* pointer = smartPointer.get();

    // Return the value of pointed-to object.
    double value = *smartPointer;

    // Access a field or function of a pointed-to object
    unique_ptr<pair<int, string>> pairPointer(new pair<int, string>(1, "Heap Pair"));
    pairPointer->first = 2;
    pairPointer->second = "Update Pair String";

    // Deallocate current pointed-to object and store new pointer.
    smartPointer.reset(new double(2.818));

    // Release responsibility of the managed pointer.
    pointer = smartPointer.release();

    return 0;
}
```

`unique_ptr` Transferring Ownership

- Use **`reset()`** and **`release()`** to transfer ownership
  - **`release`** returns the pointer, sets wrapped pointer to **`nullptr`**
  - **`reset`** will **`delete`** the current pointer and stores a new one

```
unique_ptr<int> x(new int(5));
cout << "x: " << x.get() << endl;

// x releases ownership to y
unique_ptr<int> y(x.release());
cout << "x: " << x.get() << endl;
cout << "y: " << y.get() << endl;

unique_ptr<int> z(new int(10));
// y transfers ownership of its pointer to z.
// z's old pointer was deleted in the process.
z.reset(y.release());
```

Use Caution with `get()`

- Can cause double delete errors

```
#include <memory>
using namespace std;

void processPointers()
{

    // Trying to get two pointers to the same thing
    unique_ptr<int> x(new int(12));
    unique_ptr<int> y(x.get());

} // Error: Double delete upon return!
```

`unique_ptr` and Arrays

- **`unique_ptr`** can store arrays as well
  - Will call **`delete[]`** upon destruction

```
#include <memory>
using namespace std;

int main()
{
    unique_ptr<int[]> smartPointer(new int[100]);

    smartPointer[0] = 1;
    smartPointer[1] = 2;
    smartPointer[2] = 3;

    return 0;
}
```

Introducing: `shared_ptr`

- A `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
  - The copy/assign operators are **not** disabled and increment or decrement reference counts as needed
  - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
- When a `shared_ptr` is destroyed, the reference count is decremented
- When the reference count hits 0, then we `delete` the pointed-to object!

Introducing: `shared_ptr`

- **Reference counting**: a technique for managing resources by counting and storing the number of references (i.e. pointers that hold the address) to an object

```cpp
#include <iostream>
#include <memory>
using namespace std;

void function(shared_ptr<int>& shared)
{
    shared_ptr<int> second = shared;      // reference count: 2
    cout << *second << endl;
}

int main()
{
    shared_ptr<int> first(new int(10));  // reference count: 1

    function(first);

    cout << *first << endl;               // reference count: 1

    return 0;
}                                         // reference count: 0
```