1 RAI

RAII is an acronym for the idiom “Resource Acquisition is Initialization.” What is meant by “resource acquisition is initialization” is that a resource should be acquired when an object is initialized, where a resource can be anything from data stored in the heap or the stack to a text file. In other words, every resource should be associated with an object, and when the object is initialized the resource should be acquired. Although not included in the name, another concept included in RAII is that a resource should be released upon destruction of the associated object. Thus, RAII states that every resource should be associated with an object and that the lifetime of the resource should be tied to the lifetime of the object.

Data stored on the stack automatically, although somewhat trivially due to the lack of an “associated object,” abides by the RAII idiom. Indeed, when any stack variable goes out of scope, the underlying data is destroyed.

```cpp
1 void fun() {
  // an array of integers is allocated on the stack
  int arr[10] = {1, 2, 3, 4, 5};
  // when fun() finishes running, all variables and underlying data
  // in its scope are erased
}

int main() {
  // arr not yet allocated
  fun();
  // arr has been deallocated
}
```

Things are more complicated when heap memory is used, since there is a separation between the pointer, which often lives on the stack, and the underlying data, which often lives on the heap; thus, it is possible for the pointer (the object) to be deleted without the data (the resource) being deleted. Accordingly, one of the main purposes of abiding by the RAII idiom is to prevent memory leaks. Fortunately, C++ provides classes that automatically tie the lifetime of some allocated heap memory to the lifetime of an instance of the class.

1.1 Smart Pointers

1.1.1 Shared Pointers

A shared pointer is an object that manages some heap memory, abiding by the RAII idiom. Multiple shared pointers are allowed to point to the same piece of memory, but once the number of shared pointers drops to 0, the memory is deallocated. To create the resource the shared pointer manages, use the `std::make_shared` function.

```cpp
1 void fun_copy(std::shared_ptr<intVec> v) {
  // print out the number of shared_ptr objects managing the intVec
```
3 std::cout << v.use_count() << std::endl;
4 }
5 void fun_ref(std::shared_ptr<intVec>& v) {
7 std::cout << v.use_count() << std::endl;
8 }
9 int main() {
10 using intVec = std::vector<int>;
11 // create shared_ptr to a vector of 10 integers, all with value 1
12 std::shared_ptr<intVec> shared = std::make_shared<intVec>(10,1);
13 // the following line prints out 2
14 fun_copy(shared);
15 the following line prints out 1
16 fun_ref(shared);
17 }

In the above code, line 14 prints out 2 because the function fun_copy creates a copy of the shared pointers, so that there are 2 shared_ptr objects managing the intVec within the scope of fun_copy. On the other hand, line 16 prints out 1 because the shared_ptr is passed by reference, so that no copy is made and only 1 shared pointer object manages the intVec.

By using the std::make_shared function we avoid bad code like the code that follows

1 int main() {
2 int* ptr = new int(10); // don't do this!
3 std::shared_ptr<int> shared(ptr);
4 std::shared_ptr<int> shared2(ptr);
5 std::cout << shared.use_count();
6 std::cout << shared2.use_count();
7 }

Here, the number 1 will be printed out on both lines 4 and 5 because we did not call the copy constructor of the shared_ptr shared to create shared2. Indeed, in order for the shared_ptrs to know the number of existing shared_ptrs to a resource, new shared_ptrs must be created by calling copy constructors to old ones. In fact, the issue just described creates a runtime error when we try to run the above code: when main() finishes, shared2 is destroyed before shared. But both shared_ptrs manage the same resource, and each thinks it’s the only shared_ptr, so when shared2 is destroyed the resource it manages is deallocated. But then, when shared is destroyed it tries to deallocate its own resource, which was already deallocated during the destruction of shared2; thus, we have a runtime error because it is undefined behavior to deallocate memory that has already been deallocated.

The following code would be the proper way to initialize a shared_ptr

1 int main() {
2 std::shared_ptr<int> shared = std::make_shared<int>(10);
3 std::shared_ptr<int> shared2(shared);
4 std::cout << shared.use_count();
5 std::cout << shared2.use_count();
6 }

It is important to understand how to properly use a shared_ptr that is a member variable of a class.

1 using intVec = std::vector<int>;
2 class Example {
3 public:
4 Example(size_t sz, int num) : v(make_shared<intVec>(sz, num)) {} 
5 Example(const Example& ex) : v(ex.v) {} 
6 std::shared_ptr<intVec> v;
In the code above, the Example class has as a member variable a shared_ptr to an intVec. On line 10, we call the copy constructor of the Example class, which in turn calls the copy constructor of the shared_ptr class to initialize the shared_ptr inside of ex2. Thus, on line 12 there are 2 shared_ptrs referring to the same data, and the shared_ptrs were initialized correctly, so a 2 is printed. When the createObjects function finishes being called, ex2 and then ex1 are destroyed, freeing the intVec to which they refer.

One can use the member functions .reset and .swap to modify a shared pointer.

```
1 std::shared_ptr<int> sh1 = std::make_shared<int>(10);
2 std::shared_ptr<int> sh2(sh1);
3 std::shared_ptr<int> sh3 = std::make_shared<int>(20);
4 std::cout << *sh1 << " " << *sh2 << " " << *sh3 << std::endl;
5 std::swap(sh1,sh3);
6 std::cout << *sh1 << " " << *sh2 << " " << *sh3 << std::endl;
```

The code above prints out “10 10 20” and then “20 10 1.” Note that swap only changes where a specific shared_ptr points; it does not modify the other shared_ptrs pointing to the same object.

```
1 std::shared_ptr<int> sh1 = std::make_shared<int>(10);
2 std::shared_ptr<int> sh2(sh1);
3 std::cout << *sh1 << " " << *sh2 << " " << sh1.use_count() << std::endl;
4 sh1.reset(new int(50));
5 std::cout << *sh1 << " " << *sh2 << " " << sh1.use_count() << std::endl;
6 sh2.reset(new int(200));
7 std::cout << *sh1 << " " << *sh2 << " " << sh1.use_count() << std::endl;
```

The code above prints out “10 10 2,” “50 10 1,” and then “50 200 1.” This is because the .reset function only changes a specific shared_ptr and not the other shared_ptrs pointing to the same object. Note that if we call .reset on a shared_ptr that is the only one pointing to a certain object, then the object will be destroyed.

### 1.1.2 Unique Pointers

Unique pointers accomplish the same task as shared_ptrs—abiding by RAII—but unique_ptrs cannot be copied. Unique_ptrs should be constructed using the std::make_unique function.

```
1 std::unique_ptr<int> up = std::make_unique<int>(10);
2 std::unique_ptr<int> up2(up); // ERROR! Cannot copy a unique_ptr
```

As in the case of shared_ptrs, for unique_ptrs to be useful, the programmer must use them correctly. For example,
Here, we create 2 unique_ptrs to the same resource. One should always use std::make_unique for construction and attempt to copy smart pointers via their copy constructors, otherwise they will not work as intended.

The swap and reset functions work the same as with shared_ptrs, except the reset function will always destroy the old resource, as we know for certain that there was only one pointer pointing to the resource. Another member function called release will move a unique_ptr’s ownership of a managed object, return a pointer to the old managed object, and change the smart pointer to point to a nullptr.

The above code prints out “10.” (The std::make_unique function is in C++14 but not C++11.)

As mentioned in class, unique_ptrs can be used to seemlessly manage dynamic arrays as long as they’re templated with an array type.

2 Value Categories

Expressions in C++ are placed into different “value categories,” where an expression is any sequence of characters that returns a value. Examples of expressions are given below

2.1 l-values

Expressions that have a place in memory are l-values. Common examples include

1. A named variable.
2. A function call, where the function explicitly returns an l-value reference.
3. A specific entry of an array (int arr[2][1,2]; arr[1];). Think of the “name” as the index.

4. A string literal or an entry thereof (“Hello”). Characters of string literals have names (being an array).

```cpp
const int& fun(const int& arg) {
    return arg;
}

int main() {
    int x = 10;
    int y = x; // the expression "x" is an l-value by 1.
    fun(x); // expression "fun(x)" is an l-value by 2.
    int arr[3]{1, 2, 3};
    arr[1]; // expression is an l-value by 3.
    "Hello world!"[5]; // get the 'o' at the end of "Hello." l-value by 4.
}
```

### 2.2 x-values

An expression is an x-value if it is an object that is reaching the end of its lifetime and member data is being accessed from the object or it is intended that member data will be accessed. X-values usually arise when we wish to harvest data from an object, but we don’t bother giving the object a name, keeping it temporary, because we don’t need the object anymore after we’re done harvesting. Examples of expressions that are x-values include:

1. An expression std::move(something). std::move(something) casts something into a r-value, in particular a x-value, so that data can be harvested from something.

2. L-value member access of an object that is not an l-value.

```cpp
struct Example {
    Example(std::string var = "Hello") : var(var) {}
    std::string var;
}

int main() {
    Example().var; // x-value because we access l-value member data of a temporary object
    // because we access named data (var), the pr-value Example() is made into
    // a temporary object, which is an x-value
    Example ex;
    Example ex2(std::move(ex.var)); // x-value because std::move is used. Here it is intended
    // to take the data stored in the string var within ex
    // and place that data into ex2 (i.e. move the data).
}
```

It is perhaps helpful to think of an x-value as a pr-value+ or an l-value-. When we use std::move on an l-value expression, we are downgrading the data associated with the l-value expression to temporary (1. above). When we access l-value member data from a nameless object returned by a pr-value expression, the nameless object is upgraded to a temporary object, from which we are able to access named member data.
2.3 pr-values

A pr-value is an expression without an identity, in the sense that the expression has no associated location in memory. Examples include

1. Numeric literals.
2. Nameless objects.
3. Function calls on functions that return by value.

Examples follow

```plaintext
int fun(Example ex) {
    return ex.var;
}
struct Example {
    Example(int x = 5) : var(var) {}
    int var;
}
int main() {
    Example(); // pr-value because Example is a nameless object (2.)
    Example ex(1); // pr-value because 1 is a nameless literal (1.)
    fun(ex); // pr-value because function returns by value. (3.)
}
```