Lecture 17 RAII and Smart Pointers

#CS106L

- **RAII**
- **Examples**
- **Smart Pointers**

---

avoiding exceptions entirely:Google C++ Style Guide

---

1. resources that need to be acquired/released

- Heap memory new/delete
- Files openc/lose
- Locks try_lock/unlock
- Sockets socket/close

2. We can’t guarantee the `delete` is called if an exception is thrown

```
string EvaluateSalaryAndReturnName(int idNumber) {

 Employee* e = new Employee(idNumber);

 if ( e.Title() == "CEO" || e.Salary() > 100000 ) {

 cout << e.First() << " "

 << e.Last() << " is overpaid" << endl;

 }

 auto result = e.First() + " " + e.Last();

 delete result; // what if we skip this line?
```

```
  return result;

 }
```

3. Functions can have four levels of exception safety:

• Nothrow exception guarantee:absolutely does not throw exceptions: destructors, swaps, move constructors, etc.

• Strong exception guarantee:rolled back to the state before function call

• Basic exception guarantee:program is in valid state after exception

• No exception guarantee:resource leaks, memory corruption, bad…

4. exceptions:

   - In C++, exceptions are a mechanism for handling errors or exceptional conditions in a program. They allow you to separate the error-handling code from the normal flow of control in your program, making it easier to write and maintain.

   - exceptions are implemented using the `try`, `throw`, and `catch` keywords. When a program encounters an exceptional condition, it throws an exception using the `throw` keyword. The `try` block is used to enclose the code that may throw an exception. When an exception is thrown within a try block, the program looks for a matching catch block. The exception is then caught by a `catch` block that is designed to handle that type of exception. If a matching catch block is found, the program transfers control to that block and the exception is handled. If no matching catch block is found, the program will terminate.

5. RAII(Resource Acquisition Is Initialization):all resources should be acquired in the constructor and released in the destructor.

   - There should never be a "half-valid" state of the object, which is useable after its creation. The destructor is always called (even in the case of an exception or an early exit from a function), so the resource is always freed. The basic idea behind RAII is to tie the lifetime of a resource to the lifetime of an object.

Here's an example of RAII in C++:

```
class File {

public:

    File(const std::string& filename) {

        file_handle_ = fopen(filename.c_str(), "r");

        if (!file_handle_) {

            throw std::runtime_error("Error: Failed to open file");

        }

    }

    ~File() {

        if (file_handle_) {

            fclose(file_handle_);

        }

    }

    // other functions to read and write to the file

private:

    FILE* file_handle_;

};
```

6. Is it RAII complaint?

```
void printFile () {

 ifstream input();

 input.open(“hamlet.txt”);

 string line;
```

```
 while (getline(input, line)) {
 cout << line << endl;
 }
 input.close();
}
```

The code you provided is not completely RAII compliant. Resource not acquired in the constructor or released in the destructor. If an exception occurs during the execution of the function, the file may not be closed properly, which could cause data loss or other problems.

Here's how the function could be modified to be RAII compliant:

```
void printFile () {
    std::ifstream input("hamlet.txt");
    std::string line;
    while (std::getline(input, line)) {
        std::cout << line << std::endl;
    }
}
```

By removing the explicit `open` and `close` calls and creating the `ifstream` object on the stack, it ensures that the file handle is closed automatically when the `ifstream` object goes out of scope, even in case an exception occurs.

another case:

```
void cleanDatabase (mutex& databaseLock,
map<int, int>& database) {
 databaseLock.lock();
 // other threads will not modify database
```

```
 // if exception thrown, mutex never unlocked!
 databaseLock.unlock();
}
```

The fix: an object whose sole job is to release the resource when it goes out of scope.

```
class lock_guard {
public:
lock_guard(mutex& lock) : acquired_lock(lock) {
        acquired_lock.lock()
}
~lock_guard() {
        acquired_lock.unlock();
}
private:
mutex& acquired_lock;
}
void cleanDatabase (mutex& databaseLock,
map<int, int>& database) {
        lock_guard<mutex>(databaseLock);
}
```

mutexes cannot be copied, and cannot be moved,that’s why we initialize `lock_guard` using the initializer_list.

```
class mutex {
```

```
public:

mutex(const mutex& other) = delete;

mutex& operator=(const mutex& rhs) = delete;

mutex(mutex&& other) = delete;

mutex& operator=(mutex&& other) = delete;

}
```

by the way: The following example isn't RAII complaint

```
void rawPtrFn () {

 Node* n = new Node;

 // do some stuff with n…

 delete n;

}
```

7. avoid calling `new` and `delete` explicitly

Q1:Without any extra information, how many potential code paths are in this function?

```
string EvaluateSalaryAndReturnName( Employee e ) {

 if ( e.Title() == "CEO" || e.Salary() > 100000 ) {

 cout << e.First() << " "

 << e.Last() << " is overpaid" << endl;

 }

 return e.First() + " " + e.Last();

}
```

There are (at least) 23 code paths in the code before!

- 1 – Copy constructor of Employee parameter, may throw.
- 5 – Constructor of temp string/ints, may throw.
- 6 – Call to Title, Salary, First (2), Last (2), may throw.
- 10 – Operators may be user-overloaded, may throw.
- 1 – Copy constructor of string for return value, may throw.

Q2: `try` and `catch`?

In C++, the `try` and `catch` keywords are used to handle exceptions. The `try` block encloses the code that may throw an exception, and the `catch` block contains the code that will handle the exception if one is thrown.

Here's an example of how to use `try` and `catch` in C++:

```
try {
    // code that may throw an exception
    int x = some_function();
    if (x == 0) {
        throw std::runtime_error("Error: x cannot be zero");
    }
    // more code that may throw an exception
} catch (std::runtime_error& e) {
    // code to handle the exception
    std::cout << e.what() << std::endl;
}
```

In this example, the code in the try block calls a function `some_function()` and checks the returned value. If the value is zero, it throws a

`std::runtime_error` exception with an error message. The catch block catches the exception and prints the error message to the console.

It's also possible to have multiple catch blocks to handle different types of exceptions, like this:

```
try {
    // code that may throw an exception
} catch (std::runtime_error& e) {
    // code to handle std::runtime_error exceptions
} catch (std::exception& e) {
    // code to handle other types of std::exception
} catch (...) {
    // code to handle any other type of exception
}
```

It's worth noting that the catch block that can handle the specific exception should be put before the catch block that can handle more general exception, as the catch block that can handle the specific exception will be chosen.

You can also use the `catch(...)` block to handle all types of exceptions.

It's worth noting that the `catch` block should be used to handle the exception and take appropriate action, but not to continue the normal execution flow, as the exception may indicate a severe problem.

Q3:memory leak?

A memory leak occurs when a program dynamically allocates memory for an object or data structure, but then fails to deallocate that memory when it is no longer needed. As a result, the memory remains allocated and can't be used for other purposes.

Over time, as the program continues to allocate and fail to deallocate memory, the amount of memory used by the program can grow until it exhausts all available memory and causes the program to crash or behave erratically.

Memory leaks can be caused by various reasons, such as

- forgetting to call the delete operator on dynamically allocated memory
- using new/malloc in a loop without freeing the memory
- creating circular references between objects that leads to memory not being freed
- using memory management functions incorrectly
- using a memory management library with a bug

Detecting and fixing memory leaks can be difficult, as they often don't manifest themselves until the program has been running for a long time. Some tools, such as valgrind, can help detect memory leaks by analyzing the memory usage of a program at runtime.

Q4: `std::unique_ptr`?

A `std::unique_ptr` is a C++ object that wraps a raw pointer and automatically releases the resource that the raw pointer points to when the `std::unique_ptr` object is destroyed. This ensures that the resource is always properly released, even in the case of an exception or an early exit from a function. It is defined in the `<memory>` header.

Here's an example of how to use a `std::unique_ptr`:

```
#include <memory>

int main() {

    std::unique_ptr<int> p(new int(5)); // create a unique_ptr that owns a new int

    *p = 6; // use the unique_ptr as a regular pointer

    std::cout << *p << std::endl; // prints 6

    // the int is automatically deleted when p goes out of scope

    return 0;

}
```

You can also use the `std::make_unique` function to create a `std::unique_ptr`:

```
#include <memory>

int main() {

    auto p = std::make_unique<int>(5); // create a unique_ptr that owns a new int

    *p = 6; // use the unique_ptr as a regular pointer

    std::cout << *p << std::endl; // prints 6

    // the int is automatically deleted when p goes out of scope

    return 0;

}
```

You can also use move semantics with `std::unique_ptr`:

```
        std::unique_ptr<int> p1 = std::make_unique<int>(5);

        std::unique_ptr<int> p2 = std::move(p1);
```

This will transfer ownership of the pointer to `p2` and `p1` will be reset.

`std::unique_ptr` is designed to be used in single ownership situations, where one object owns the resource and is responsible for releasing it. If you want to share ownership of a resource among multiple objects, you can use `std::shared_ptr`.

It's worth noting that `std::unique_ptr` is not copyable, only movable. This means that you cannot make a copy of a `std::unique_ptr`, which eliminates the possibility of sharing a pointer and causing a double delete, but you can transfer ownership of the pointer to another `std::unique_ptr` using the move constructor or move assignment. This is to ensure that there is only one owner of the resource at any given time, and to prevent multiple `std::unique_ptr` objects from releasing the same resource.

Also, `std::unique_ptr` is a better alternative than using raw pointers, as it automatically releases the resource when the `unique_ptr` object is destroyed, which prevents memory leaks.

Q5: `std::shared_ptr`?

A `std::shared_ptr` is a C++ object that wraps a raw pointer and automatically releases the resource that the raw pointer points to when all `std::shared_ptr` objects that point to the resource have been destroyed. This ensures that the resource is always properly released, even in the case of an exception or an early exit from a function.

Here's an example of how to use a `std::shared_ptr`:

```
#include <memory>

int main() {

    std::shared_ptr<int> p1(new int(5)); // create a shared_ptr that owns a new int

    std::shared_ptr<int> p2 = p1; // p2 shares ownership of the int with p1

    *p1 = 6; // use the shared_ptr as a regular pointer

    std::cout << *p2 << std::endl; // prints 6

    // the int is only deleted when both p1 and p2 go out of scope

    return 0;

}
```

You can also use the `std::make_shared` function to create a `std::shared_ptr`:

```
#include <memory>

int main() {

    auto p = std::make_shared<int>(5); // create a shared_ptr that owns a new int

    *p = 6; // use the shared_ptr as a regular pointer

    std::cout << *p << std::endl; // prints 6

    // the int is only deleted when p goes out of scope

    return 0;

}
```

It's worth noting that `std::shared_ptr` can be used to share ownership of a resource among multiple objects, which allows for easy management of dynamically allocated resources that may be shared among multiple parts of a program.

Q6: `std::weak_ptr`?

A `std::weak_ptr` is a C++ object that holds a non-owning "weak" reference to an object managed by `std::shared_ptr`.

Here's an example of how to use a `std::weak_ptr`:

```
#include <memory>

class A;class B;

int main() {

    std::shared_ptr<A> pA(new A);

    std::shared_ptr<B> pB(new B);

    pA->b = pB;

    pB->a = std::weak_ptr<A>(pA); // pB holds a weak reference to A, so it doesn't increase the reference count of pA

    // Now, pA and pB have a circular reference to each other, but the weak reference from pB to pA will not prevent pA from being deleted when it goes out of scope

    return 0;

}
```

You can check whether the object managed by a `std::weak_ptr` is still valid by calling the `expired` member function, which returns `true` if the object has been deleted and `false` otherwise. To obtain a `std::shared_ptr` to the object, you can use the `lock` member function, which returns a `std::shared_ptr` if the object is still valid, and an empty `std::shared_ptr` otherwise.

```
std::weak_ptr<int> wp = std::make_shared<int>(5);

if (!wp.expired()) {
```

```
    std::shared_ptr<int> sp = wp.lock();

    if (sp) {

        std::cout << *sp << std::endl; // prints 5

    }

}
```

It's worth noting that a `std::weak_ptr` does not increase the reference count of the object it points to, so it does not affect the lifetime of the object. It's useful in situations where a circular reference is present, where one object holds a shared_ptr to another, and vice versa.

It's also useful when multiple objects share ownership of a resource, but one or more of them do not need to keep the resource alive. `std::weak_ptr` can be used to hold a non-owning reference to the resource, which allows the resource to be deleted when the last `std::shared_ptr` that owns it goes out of scope, without affecting the lifetime of the objects that hold a `std::weak_ptr` to the resource.