Lecture #18 | Resource Acquisition Is Initialization (RAII) Exception handling

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Today’s topic

- Resource Acquisition Is Initialization (RAII)
- Exception handling
- Run-time type information (RTTI)

Resource Acquisition Is Initialization (RAII)

- Resource acquisition is initialization (RAII) is C++ idiom used to *avoid resource leaks* and *provide exception safety*
- Associate resource with owning object (i.e., RAII object)
- Period of time over which resource held is tied to lifetime (i.e., scope) of RAII object
- Resource acquired during creation of RAII object
- Resource released during destruction of RAII object
- Provided RAII object properly destroyed, resource leak cannot occur

Revisit: unique_ptr

```
void foo()
{
    int* ptr = new int[1024];
    unique_ptr<int[]> sp { new int[1024] };

    // some actions here...

    // no need to delete sp!!!
    delete[] ptr;
}
```

Example: traditional resource management

```
void useBuffer(char* buf) { /* ... */ }

void doWork () {
    char* buf = new char[1024]; // allocation (and initialize)
    useBuffer(buf);             // use
    delete[] buf;               // deallocation (=release)
}
```

Example: RAII object

```
template <class T>
class SmartPtr {
public:
    SmartPtr(int size) : ptr_(new T[size]; ) {}
    SmartPtr () { delete[] ptr_ ; }
    operator T*() { return ptr_; }
    // ...
private:
    T* ptr_;
};
void useBuffer(char* buf) { /* ... */ }
void doWork () {
    SmartPtr <char> buf(1024);
    useBuffer(buf);
}
```

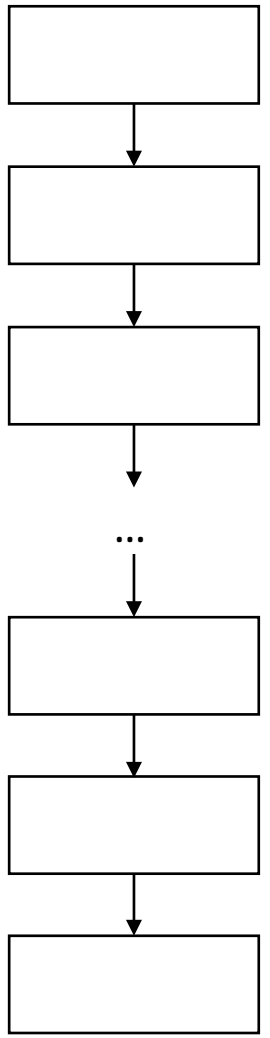
Exception handling

- Exceptions are C++ language mechanism for handling exceptional (i.e., abnormal) situations, often employed for error handling
- Exceptions propagate information from point where error *detected* to point where error *handled*
  - Code that encounters error that it is unable to handle throws exception
  - Code that wants to handle error catches exception and performs processing necessary to handle error
- Exceptions provide convenient way in which to *separate error detection from error handling*

* From “Lecture Slides for the C++ Programming Language

When exception (error) handling is complicated

- Error can be detected in low-level code, but we may want to handle error in high-level code → need to propagate error information up call chain
- Why handling error in high-level code?
  - We may not have control at low-level code
  - We want to handle errors at one location while errors may be detected at several locations
  - We want to keep error handling where we have relevant context



High-level code

Low-level code

Traditional way of exception handling

- Terminate program
- Handle errors locally: handle errors where they are detected by calling a function or implementing (duplicate) codes
  - May not have all the context to properly handle errors
- Pass error code and have caller check error code: via return value, reference parameter or global object
  - Explicit actions are required for error checking and handling
    - Caller function may forget error handling
    - Code can become cluttered for error checking code

Example: traditional error handling

```
bool func3() {
    bool success = false;
    // ...
    if (is_every_thing_okay)
        success = true;
    // ...
    return success;
}
bool func2() {
    if (!func3()) { return false; }
    // ...
    return true;
}
```

```
bool func1() {
    if (!func2()) { return false; }
    // ...
    return true;
}
int main() {
    if (!func1()) {
        cout << "failed\n";
        return 1;
    }
    // ...
}
```

Error handling with exceptions

- Error detection and handling
  - An error condition is signaled by throwing exception (with `throw` statement)
  - An exception is caught by handler (in `catch` clause of `try` statement)
- Pros
  - Exceptions allow for error handling code to be easily separated from code that detects error
  - Exceptions can easily pass error information many levels up call chain
  - Passing of error information up call chain managed by language (no explicit code required)
- Cons
  - Writing code that always behaves correctly in presence of exceptions requires great care
  - Although possible to have no execution-time cost when exceptions not thrown, still have memory cost

Example: exception

```
void func3() {
    bool success = false;
    // ...
    if (!success) { throw std::runtime_error("OTL"); }
}
void func2() { func3(); // ... }
void func1() { func2(); // ... }
int main() {
    try { func1(); }
    catch (std::runtime_error& e) {
        std::cout << "failed” << e.what() << std::endl;
        return 1;
    }
    // ...
}
```

Exception object

- Exceptions are objects (of any type)
  - Type of object typically indicate error type
  - Value of object provide details of a particular exception
- C++ standard library defines `std::exception` and (directly/indirectly) derived classes

Type	Description
`logic_error`	faulty logic in program
`runtime_error`	error caused by circumstances beyond scope of program
`bad_typeid`	invalid operand for **typeid** operator
`bad_cast`	invalid expression for **dynamic_cast**
`bad_weak_ptr`	bad weak_ptr given
`bad_function_call`	function has no target
`bad_alloc`	storage allocation failure
`bad_exception`	use of invalid exception type in certain contexts

Reference: hierarchy of derived classed of exception

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (C++11)
  - bad_optional_access (C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (C++11)
  - tx_exception (TM TS)
  - system_error (C++11)
    - ios_base::failure (C++11)
    - filesystem::filesystem_error (C++17)
- bad_typeid
- bad_cast
  - bad_any_cast (C++17)
- bad_weak_ptr (C++11)
- bad_function_call (C++11)
- bad_alloc
  - bad_array_new_length (C++11)
- bad_exception
- ios_base::failure (until C++11)
- bad_variant_access(C++17)

Exception catching

- `catch` clause of `try-catch` block catches exceptions
  - `try` block include code (incl. function calls) that might throw exceptions
  - `catch` block(s) catches and handles exceptions of designated types
    - Use reference to avoid copying and to allow exception object to be modified and rethrown
- Example

```
try {
    // code that might throw exception
}
catch (const std::logic_error& e) {
    // handle logic_error exception
}
catch (const std::runtime_error& e) {
    // handle runtime_error exception
}
catch (...) {
    // handle other exception types
}
```

Example: exception handling

```cpp
#include <iostream>
#include <vector>
#include <exception>

int main()
{
    std::vector<int> v(42);

    try {
        v.at(42) = 100;
    }
    catch (const std::out_of_range& e) {
        std::cerr << "Out of Range error: " << e.what() << '\n';
        // Out of Range error: vector
    }
}
```

Example: user-defined exception object

```
#include <iostream>
using namespace std;
class RoomOutOfRange {
public:
    string detail;
    RoomOutOfRange(string s) : detail(s) {}
    string& what() { return detail; }
};
void addRoom(int room) {
    if (room < 0) throw
RoomOutOfRange("addRoom()");
    // ...
}
void printRoom(int room) {
    if (room < 0) throw
RoomOutOfRange("printRoom()");
    // ...
    throw "Something wrong!!!";
}
```

```
void parseLine(string& line) {
    if (line[0] == 'a') addRoom(-1);
    else if (line[0] == 'p') printRoom(-1);
}
int main() {
    string line;
    while (getline(cin, line)) {
        try {
            parseLine(line);
        }
        catch (RoomOutOfRange& e) {
            cout << "Room out of range: ”
                 << e.what() << endl;
        }
        catch (string& s) {
            cout << s << endl;
        }
    }
}
```

(Optional) Run-time type information (RTTI)

- Run-time type information report dynamic type of a specific object, when an object is assign to a pointer/reference variable of base class
- Base class should be polymorphic, i.e., it should have at least one virtual function
- C++ provides two keywords
  - `dynamic_cast<>`
    - Within <>, a pointer or reference type can be specified
    - If conversion is possible, the expression is evaluated to the pointer/reference of the specified type; `nullptr` otherwise
  - `typeid()`
    - Provides type information so that it can be compared if necessary
- Use polymorphism instead of RTTI if possible

(Optional) dynamic_cast

```
struct Base { virtual ~Base() {} }; // must to be polymorphic (i.e., to have virtual function)
struct Derived: Base { virtual void name() {} };
int main() {
    Base* b1 = new Base;
    if(Derived* d = dynamic_cast<Derived*>(b1)) {
        std::cout << "downcast from b1 to d successful\n";
        d->name(); // safe to call
    }
    Base* b2 = new Derived;
    if(Derived* d = dynamic_cast<Derived*>(b2)) {
        std::cout << "downcast from b2 to d successful\n";
        d->name(); // safe to call
    }
    delete b1;
    delete b2;
}
```

Reading list

- Learn C++
  - Exception: Ch. 15
- Reference
  - Exception: https://isocpp.org/wiki/faq/exceptions

What will be covered next time

- Introduction of Design Patterns

ANY QUESTIONS?