ECE 449/590 – OOP and Machine Learning
Lecture 21 Resource Management and Object Composition

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 7, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 11
- Next lecture: Accelerated C++ 11

Outline

Resource Management

Object Composition

Resource Management

- Resources: things with limited availability
  - Memory
  - File handles
  - Network connections
  - Database connections
  - etc.
- Management: release a resource promptly
  - Only when the resource was acquired successfully.
  - After all the usages of the resource.

Resource management seems to be easy.

```
void some_function() {
    FILE *fp = fopen("input_file", "r");
    ... // do something with the file
    fclose(fp);
}
```

- Consider file operations in C.
  - The file handle is acquired by `fopen` and released by `fclose`.
- A typical code sandwich: acquire – use – release.

Resource management could become complicated.

```
// C-style resource management
void some_function() {
    FILE *fp = fopen("input_file", "r");
    if (...) {
       ...
       fclose(fp);
       return;
    }
    for (...) {
        if (...) {
            ...
        }
        else {
             ...
             fclose(fp);
             return;
        }
    }
    fclose(fp);
}
```

- When there are many returns, the programmer is responsible to make sure the resource is always released.

The previous programs are WRONG!

```
void some_function() {
    FILE *fp = fopen("input_file", "r");
    another_function(fp);
    fclose(fp);
}
```

- If `another_function` `throw`s an exception, `fclose(fp);` won't be executed.
- There will be resource leakage.
  - A function in the call stack may allow the program to recover from the exception.
  - However, you won't be able to release the file handle `fp`. It's lost.

Thinking of try-catch?

```
void some_function() {
    FILE *fp = NULL;
    try {
        fp = fopen("input_file", "r");
        another_function(fp);
        fclose(fp);
    }
    catch (...) {
        if (fp != NULL) fclose(fp);
    }
}
```

- I should use "finally" as in many other languages.
- Though in C++ we don't have it since we don't need it.
    - Those languages eventually learned from C++ so that "finally" is not needed in many cases nowadays.

Too complicated!

```
// similar to Java resource management prior to Java 1.7
void some_function() {
    FILE *fp_in = NULL; FILE *fp_out = NULL;
    try {
        fp_in = fopen("input_file", "r");
        do_something(fp_in);
        if (...) { fclose(fp_in); return;}
        fp_out = fopen("output_file", "w");
        do_something_else(fp_in, fp_out);
        for (...) {
            ...
            if (...) { fclose(fp_out); fclose(fp_in); return;}
        }
        another_function(fp_out);
        fclose(fp_out); fclose(fp_in);
    } catch (...) {
        if (fp_in != NULL) fclose(fp_in);
        if (fp_out != NULL) fclose(fp_out);
    }
}
```

- When there are many returns and many resources.
- I can't guarantee the correctness of the above code since it is too complicated.

Resource Acquisition Is Initialization (RAII)

```
void some_function() {
    std::ifstream fin("input_file");
    do_something(fin);
    if (...) return;
    std::ofstream fout("output_file");
    do_something_else(fin, fout);
    for (...) {
        ...
        if (...) return;
    }
    another_function(fout);
}
```

- C++ makes programmer's life much easier via RAII.
  - Leverage object lifetime for resource management (as adopted by many other languages).
  - Use object composition to manage multiple resources.

RAII (Cont.)

```
void some_function() {
    std::ifstream fin("input_file");
    do_something(fin);
    if (...) return;
    std::ofstream fout("output_file");
    do_something_else(fin, fout);
    for (...) {
        ...
        if (...) return;
    }
    another_function(fout);
}
```

- Lifetime of the local objects like `fin` and `fout` is within the function.
  - A resource is acquired in ctor when the object is constructed.
  - The resource is released in dtor when the object is destroyed.
- What about exceptions?

Stack Unwinding

```
void some_function() {
    std::ifstream fin("input_file");
    do_something(fin);
    if (...) return;
    std::ofstream fout("output_file");
    do_something_else(fin, fout);
    for (...) {
        ...
        if (...) return;
    }
    another_function(fout);
}
```

- Local objects are destroyed automatically when the function exits, either normally or due to an unhandled exception, e.g. from `do_something`.
- Since dtors may be called during exception handling, they SHOULD NEVER throw exceptions.

Resource Management and Object Composition

- How to design our own class for resource management via RAII?
  - E.g. `std::vector<T>` and `std::shared_ptr<T>`?
  - For curiousity and to understand C++ better.
- Need to understand how RAII interacts with object composition.
  - Object composition: compose larger object from smaller ones.
  - E.g. `std::vector<T>` contains many objects of type `T`.

Outline

Resource Management

Object Composition

Object Composition

- Object composition: compose larger object from smaller ones
  - The parent (larger) object holds the smaller objects via member variables.
- Many methods to compose the parent object from smaller objects of type `T`.
  - Use class types: a member of type `T`, or other types holding objects of type `T` like `std::vector<T>` and `std::shared_ptr<T>`.
  - Use raw pointers: a member of type `T *` with the objects on the heap.
- Composition means ownership.
  - When the parent object is constructed, the objects it holds should be constructed.
  - When the parent object is destroyed, the objects it holds should be destroyed.

Object Composition $\neq$ Association

- Association: objects may interact with each other
  - Associations are implemented as pointers and references.
- However, association does not imply ownership.
- Object composition or association?
  - There are cases where it is difficult to tell one from the other – so we need GC.
  - Otherwise, it is still preferable in C++ to distinguish the two because ownership and lifetime matter for predictable performance.

Lifetime Management for Member Variables

- Member variables are constructed before any ctor body.
  - They are constructed in the order they appear in the class definition.
  - Programmers may specify how the members are constructed using the initializer list.
  - The compiler will default initialize all the members whose constructions are not specified by programmers.
  - So you can use the members in the ctor body.
- Member variables are destroyed automatically after the dtor body.
  - So you can use the members in the dtor body.
  - In the reversed order they appear in the class definition – compiler generates such code and you shouldn't destroy the members explicitly.
  - Moreover, the compiler will generate an empty public dtor for any non-reference type if the type has no user-defined dtor.
    - For built-in types, it will do nothing.
    - For class types, it will destroy their members.

Exception in Ctors and Dtors

- If a ctor fails, i.e. `throw`s an exception, compiler guarantees that,
  - All-or-None: either the parent object is constructed successfully or (as if) none of the members got constructed.
- What if a dtor fails?
  - Dtors should never fail – you should not throw exceptions out of dtors.

Example I: The Classes

```
struct will_not_throw {
    will_not_throw() {std::cout << "ctor of will_not_throw" << std::endl;}
    ~will_not_throw() {std::cout << "dtor of will_not_throw" << std::endl;}
}; // struct will_not_throw

class ctor_throw {
    will_not_throw wnt_;
public:
    ctor_throw() {
        std::cout << "ctor of ctor_throw" << std::endl;
        throw std::runtime_error("from ctor of ctor_throw");
    }
    ~ctor_throw() {std::cout << "dtor of ctor_throw" << std::endl;}
}; // class ctor_throw
```

Example I: Parent Object on the Stack

```
void some_function() {
    ctor_throw ct;
}

void some_caller() {
    try {
        some_function();
    }
    catch (std::exception &e) {
        std::cout << "exception " << e.what() << std::endl;
    }
}
```

- The output
  ```
  ctor of will_not_throw
  ctor of ctor_throw
  dtor of will_not_throw
  exception from ctor of ctor_throw
  ```
- When the ctor of the parent object fails, the members are destroyed automatically.
  - The dtor of the parent object will not be called since the object has not been constructed – this is exactly what resource management needs!

Example I: Parent Object on the Heap

```
void another_function() {
    ctor_throw *pct = new ctor_throw;
}

void another_caller() {
    try {
        another_function();
    }
    catch (std::exception &e) {
        std::cout << "exception " << e.what() << std::endl;
    }
}
```

- The output is the same as in the previous slide.
- No memory leakage: when the construction fails, the piece of allocated memory is returned to the heap automatically.

Example II: The Classes

```
struct will_throw {
    will_throw() {
        std::cout << "ctor of will_throw" << std::endl;
        throw std::runtime_error("from ctor of will_throw");
    }
    ~will_throw() {std::cout << "dtor of will_throw" << std::endl;}
}; // struct will_throw

class member_throw {
    will_not_throw wnt_;
    will_throw wt_;
public:
    member_throw() {std::cout << "ctor of member_throw" << std::endl;}
    ~member_throw() {std::cout << "dtor of member_throw" << std::endl;}
}; // class member_throw
```

Example II: The Output

```
void some_function() {
    member_throw mt;
}

void some_caller() {
    try {
        some_function();
    }
    catch (std::exception &e) {
        std::cout << "exception " << e.what() << std::endl;
    }
}
```

- The output

  ```
  ctor of will_not_throw
  ctor of will_throw
  dtor of will_not_throw
  exception from ctor of will_throw
  ```

- When the ctor of a member fails, the members that are already constructed will be destroyed automatically.
  - The body of the ctor of the parent object won't be executed.
- Same output if the parent object is created on the heap.

Discussions

- What about smaller objects on the heap managed via raw pointers as members?
- Dtor of the parent object will not `delete` those pointers.
  - Those pointers as member variables will be destroyed after dtor of parent object.
  - If you recall that to destroy objects means to call dtors on them, then as poniters are built-in types, their dtors will do nothing.
- It is the responsibility of programmers to manage those pointers through the ctors/dtor of the parent object.
  - Exception safety: one need to provide the same All-or-None guarantee as the compiler if there are exceptions.

Summary and Advice

- Member variables are constructed before the ctor body and destroyed after the dtor body.
- Exceptions interact with objects in a complicated way.
  - Stack unwinding: local objects are automatically destroyed even when there is an unhandled exceptions.
  - All-or-None: either the parent object is constructed successfully or none of the members got constructed.