ECE 449/590 – OOP and Machine Learning
Lecture 12 Inheritance and Polymorphism

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

October 3, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 13
- Next lecture: More on Design Patterns

Outline

Inheritance

Runtime Polymorphism

Implementing Evaluation

```
int evaluation::execute() {
    variables_.clear();
    for (auto &expr: exprs_) {
        if (expr.get_op_type() == "Input") {
            // set variables_[expr_id] to kwargs_[op_name]
        }
        else if (expr.get_op_type() == "Const") {
            // set variables_[expr_id] to op_params["value"]
        }
        else if (expr.get_op_type() == "Add") {
            // locate two inputs from variables_ as a and b
            // set variables_[expr_id] to a+b
        }
        ...
}
```

- Branches are required to handle different operators when evaluating expressions.

Cohesion and OOD

- The design is of low cohesion, which leads to many issues.
  - Each branch focuses on a different type of operators.
- It is difficult to extend the design.
  - It is intuitive for designers to implement a few types of operators at a time.
  - However, to implement a new type of operator, one must add new branch.
- It is almost impossible to reuse the code.
  - A lot of copy-and-paste are needed if you want to extract code for operators.
- OOD seeks designs with high cohesion. But how?

Design with High Cohesion

- It is quite intuitive for us to achieve high cohesion by introducing a class for each type of operators.
- How to fit this design to existing code?
    - Reuse existing code instead of modifying it

Inheritance

```
typedef std::map<int, tensor> vars_type,
typedef std::map<std::string, tensor> kwargs_type;

class eval_op {
    int expr_id_;
    std::string op_name_, op_type_;
    std::vector<int> inputs_;
public:
    eval_op(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_op
```

- `eval_op` refers to an expression to be evaluated.
  - A few pieces of information are copied to save the need to use getters from `expression` everytime.
- The member function `eval` should evaluate the operation using `variables` and `kwargs`.
  - Update `variables` as necessary.
- `typedef` introduces alias for types.

Inheritance (Cont.)

```
class eval_const: public eval_op {
    tensor value_;
public:
    eval_const(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_const
```

- `eval_const` is a special `eval_op`.
  - The `value_` as the output of `Const` operator is provided by the expression `expr`.
  - For evaluation, `kwargs` is not needed. However, we follow `eval_op` to declare the `eval` member function.
- This relation is modeled through inheritance, a language feature that is one of the cornerstones of OOP.

Base Class and Derived Class

```
class eval_const: public eval_op {
    tensor value_;
public:
    eval_const(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_const
```

- The class `eval_const` is derived from, or inherits from, `eval_op`. Equivalently, `eval_op` is a base class of `eval_const`.
  - Members of the base class (data members and member functions), except the special ones, are also members of the derived class.
- `public` inheritance inherits the interface of the base class.
  - An `eval_const` object can be used when an `eval_op` object is required.
  - There are other kinds of inheritance we will discuss later.

Access Base Class Members

```
class eval_const: public eval_op {
    tensor value_;
public:
    eval_const(const expression &expr);
    void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_const
void eval_const::eval(vars_type &variables, const kwargs_type &kwargs) {
    variables[expr_id_] = value_;
}
```

- Move the type-specific code from the branches in `evaluation::execute` here.
  - Any issue?

Protection Revisited

```
void eval_const::eval(vars_type &variables, const kwargs_type &kwargs) {
    variables[expr_id_] = value_;
}
```

- Since `expr_id_` is `private` in `eval_op`, you cannot access it in `eval_const`.
- It is not wise to change the protection of `expr_id_` to `public`.
- We need a protection that's accessible in the derived class but not for general users.

protected Members

```
class eval_op {
protected:
    int expr_id_;
    std::string op_name_, op_type_;
    std::vector<int> inputs_;
    ...
}; // class eval_op
```

- It's the `protected` label.
- Which members should be changed from `private` to `protected`? It's a design decision.
  - Use `private` and accessors to control what derived classes can or cannot access.
  - Use `protected` to grant full access to derived classes for simplicity.

Inherited Protections

- What are the protection labels associated with the members in a derived class inherited from a base class?
- For `public` inheritance,
  - Except special member functions like ctors, which are NOT inherited.
  - `private` members in base class $\rightarrow$ inaccessible in derived class
  - `protected` members in base class $\rightarrow$ `protected` members in derived class
  - `public` members in base class $\rightarrow$ `public` members in derived class

Constructors

```
class eval_op {
    ...
    eval_op(const expression &expr);
}; // class eval_op
class eval_const: public eval_op {
    ...
    eval_const(const expression &expr);
}; // class eval_const
eval_const::eval_const(const expression &expr):
    eval_op(expr), value_(expr.get_op_param("value")) {
}
```

- The base class is treated as an anonymous data member in the derived class.
- As any other data members, the base class is constructed before the ctor body of the derived class.
  - To specify how the base class should be constructed, you can refer to it using its class name in the initializer list.
- Anyway, only the `protected`/`public` ctors of the base class can be used for the derived class.
- Error handling: what if the value is not avaliable?

Outline

Working with Derived Classes

- Similar to `eval_const`, we can create a class for each operator type like `eval_input`, `eval_add`, etc., all derived from `eval_op`.
- How should we create/store the objects of these different class types?
- We can hold objects of the same class type in one container.
  - One container per operator type – not an elegant solution.
- Runtime polymorphism: allow to work with many different classes as long as their base classes are the same
  - The base class interface provides means to access both the common functionalities and the extended behavior that are different in different derived classes.
  - Usually via pointers of the base class type to objects of derived types.

Smart Pointers

```
class evaluation {
public:
    evaluation(const std::vector<expression> &exprs);
    ...
private:
    ...
    std::vector<std::shared_ptr<eval_op>> ops_; // instead of exprs_
}; // class evaluation
```

- Let's use the smart pointer `std::shared_ptr<T>` to store pointers of the base class type `T` inside containers.
- Save us from many troubles working with dynamic memory allocation and containers.
  - E.g. when should we `delete` objects if they are created by `new`?
- `ops_` now holds different types of operators.
  - Though they can only be accessed using `eval_op` pointers.

Creating and Managing Derived Objects

```
evaluation::evaluation(const std::vector<expression> &exprs) {
    for (auto &expr: exprs) {
        if (expr.get_op_type() == "Input") {
            ops_.push_back(std::make_shared<eval_input>(expr));
        }
        else if (expr.get_op_type() == "Const" {
            ops_.push_back(std::make_shared<eval_const>(expr));
        }
        else if (expr.get_op_type() == "Add" {
            ops_.push_back(std::make_shared<eval_add>(expr));
        }
        ...
```

- `std::make_shared<T>` creates an object of `T` on the heap.
  - Use a ctor corresponding to the provided arguments.
  - Return `std::shared_ptr<T>`.
- Under `public` inheritance, a pointer of the derived class can be converted implicitly to a pointer of the base class.
  - The smart pointer `std::shared_ptr` supports so as well.
- Branches are still used, though they only contain code for creation.

Accessing Extended Behavior through a Base Pointer

```
class base {
public:
    void do_something() {std::cout << "from base" << std::endl;}
}; // class base

class derived : public base {
public:
    void do_something() {std::cout << "from derived" << std::endl;}
}; // class derived

int main() {
    base *p = new derived;
    p->do_something();
    delete p;
}
```

- What's the output?
  - Since `p` is a pointer to `base`, `base::do_something` will be called to output `from base`.
- So we failed to access the `do_something` of the `derived` object through the `base` pointer.

Virtual Functions

```
class base {
public:
    virtual void do_something() {std::cout << "from base" << std::endl;}
    virtual ~base() {}
}; // class base

class derived : public base {
public:
    void do_something() override {std::cout << "from derived" << std::endl;}
}; // class derived

int main() {
    ...
}
```

- To access extended behavior, e.g. `do_something`, through a base pointer, the corresponding member function should be declared `virtual` in the base class.
    - A `virtual` function remains `virtual` no matter whether you declare it to be `virtual` or not in the derived class.
    - `override` asks the compiler to make sure the function indeed overrides a base `virtual` function.

Virtual Destructor

```
class base {
public:
    virtual void do_something() {std::cout << "from base" << std::endl;}
    virtual ~base() {}
}; // class base
```

- The destructor (dtor) is a special member function.
  - Has the name of `~` plus the class name.
  - Won't return a result and takes no argument.
  - Will be called automatically when the object is destroyed.
  - Unlike ctors, a class has exactly one dtor.
- A base class with a `virtual` function should also have a `virtual` dtor.
  - We'll talk about the details later.

Updated Class Design

```
class eval_op {
    ...
    virtual ~eval_op();
    virtual void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_op

class eval_const: public eval_op {
    ...
    void eval(vars_type &variables, const kwargs_type &kwargs) override;
}; // class eval_const

eval_op::~eval_op() {
}

void eval_op::eval(vars_type &variables, const kwargs_type &kwargs) {
    assert(false); // should be provided by derived classes
}
```

- No change to implementation of `eval_const::eval`.

Implementing Evaluation Using Virtual Functions

```
int evaluation::execute() {
    variables_.clear();
    for (auto &op: ops_) {
        op->eval(variables_, kwargs_);
    }
    return 0;
}
```

- You may treat `std::shared_ptr<T>` as a pointer of `T` and use `->` to access its members.
- You may need to modify the code above to include error handling.

Virtual Functions as the Interface

```
class eval_op {
    ...
    virtual ~eval_op();
    virtual void eval(vars_type &variables, const kwargs_type &kwargs);
}; // class eval_op

void eval_op::eval(vars_type &variables, const kwargs_type &kwargs) {
    assert(false); // should be provided by derived classes
}
```

- We put an assertion into `eval_op::eval` since it doesn't make much sense to implement it.
  - This `virtual` function serves as an interface where an implementation should be provided in the derived classes.
- However, if a derived class forgets to implement its own `eval`, the error can only be detected at runtime.
- Can we enforce such requirement at compile-time?

Pure Virtual Functions and Abstract Class

```
class eval_op {
    ...
    virtual void eval(vars_type &variables, const kwargs_type &kwargs) = 0;
}; // class eval_op
```

- A `virtual` function can be declared to be pure by put `= 0` inside its declaration.
  - You don't need to define/implement it.
- A class with one or more pure `virtual` functions is called an abstract class, or an interface.
  - The `virtual` function remains pure in a derived class if not implemented – so the derived class will remain abstract.
- It is not possible to create an object of an abstract class.
  - Only objects of a non-abstract derived class can be created.
  - In other words, the `virtual` function must have been provided for objects of a class derived from the abstract class.

Accessing Behavior in Derived Classes not Using Virtual Functions

- At certain point working with the derived classes, one may find that it is not enough to work with existing `virtual` functions.
- Common anti-pattern (things that you should not do):
  - First, decide the actual type of the object that a base pointer points to.
  - Then, convert the base pointer to the pointer of the actual type.
  - Finally, use the pointer of the actual type to access its members.
- You should consider adding additional `virtual` functions to the base class.
  - Either provide default implementations or make them pure.

Summary and Advice

- Runtime polymorphism via inheritance
  - Use inheritance to reuse base classes in derived classes
  - Objects of different classes derived from a same base class can be accessible through a base class pointer.
  - `virtual` function in the base class allows to access extended behaviors defined in derived classes from the base class.
- It is always a bad idea to branch on types of objects in C++ at runtime, except when creating those objects.