CS100 Lecture 20

Inheritance and Polymorphism I

Contents

Inheritance

An item for sale

```cpp
class Item {
  std::string m_name;
  double m_price = 0.0;
public:
  Item() = default;
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
  const auto &getName() const { return m_name; }
  auto amount(int cnt) const {
    return cnt * m_price;
  }
};
```

A discounted item

A discounted item **is an** item, and has more information:

- Minimum quantity
- Discount rate

The amount for such an item is:

$$\text{amount}(n) = \begin{cases} n \cdot \text{price}, & \text{if } n < \text{minQuantity}, \\ n \cdot \text{discount} \cdot \text{price}, & \text{otherwise.} \end{cases}$$

Defining a subclass

Use **inheritance** to model the "is-a" relationship: A discounted item **is an** item.

```cpp
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;
public:
  // constructors
  // amount
};
```

- `DiscountedItem` is a *subclass* (or derived class) of `Item`, and `Item` is the *base class* of `DiscountedItem`.
- `DiscountedItem` inherits every data member and member function of `Item` (except the ctors and dtor), no matter what access level they have.

`protected` members

A `protected` member is private, except that **it is accessible in subclasses**.

- `m_price` needs to be `protected`, of course.
  - It is needed in `DiscountedItem`'s `amount` function.
- Should `m_name` be `protected` or `private`?
  - `private` is ok if `DiscountedItem` does not modify it. It is accessible through the public `getName` interface.
  - `protected` is also reasonable, if `DiscountedItem` needs to modify it.

`protected` members

```cpp
class Item {
  std::string m_name;
protected:
  double m_price = 0.0;
public:
  Item() = default;
  Item(const std::string &name, double price)
      : m_name(name), m_price(price) {}
  const auto &getName() const { return m_name; }
  auto amount(int cnt) const {
    return cnt * m_price;
  }
};
```

Inheritance

By defining `DiscountedItem` to be a subclass of `Item`, **every `DiscountedItem` object contains a subobject of `Item`**.

What can be inferred from this?

Inheritance

By defining `DiscountedItem` to be a subclass of `Item`, **every `DiscountedItem` object contains a subobject of `Item`**.

What can be inferred from this?

- A constructor of `DiscountedItem` must first initialize the `Item` subobject by calling a constructor of `Item`'s.
- The destructor of `DiscountedItem` must call the destructor of `Item` after having destroyed its own data members (`m_minQuantity` and `m_discount`).
- `sizeof(DiscountedItem) >= sizeof(Item)`

Inheritance

Key points of inheritance:

- Every object of the subclass contains a base class subobject.
- Inheritance should not break the encapsulation of the base class.
  - To initialize the base class subobject, **we must call a constructor of the base class**. It is not allowed to initialize data members of the base class subobject directly.

Constructor of `DiscountedItem`

```
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double disc)
      : Item(name, price), m_minQuantity(minQ), m_discount(disc) {}
};
```

It is not allowed to write this:

```
DiscountedItem(const std::string &name, double price,
               int minQ, double disc)
    : m_name(name), m_price(price), m_minQuantity(minQ), m_discount(disc) {}
```

Constructor of subclasses

Before the initialization of the subclass's own data members, the base class subobject **must** be initialized by calling one of its ctors.

- What if we don't call the base class's ctor explicitly?

  ```
  DiscountedItem(...)
    : /* ctor of Item is not called */ m_minQuantity(minQ), m_discount(d) {}
  ```

Constructor of subclasses

Before the initialization of the subclass's own data members, the base class subobject **must** be initialized by calling one of its ctors.

- What if we don't call the base class's ctor explicitly?
  - The default constructor of the base class is called.
  - If the base class is not default-constructible, an error.
- What does this constructor do?

  ```
  DiscountedItem() = default;
  ```

Constructor of subclasses

Before the initialization of the derived class's own data members, the base class subobject **must** be initialized by calling one of its ctors.

- What if we don't call the base class's ctor explicitly?
  - The default constructor of the base class is called.
  - If the base class is not default-constructible, an error.
- What does this constructor do?

  ```
  DiscountedItem() = default;
  ```

  - Calls `Item::Item()` to default-initialize the base class subobject before initializing `m_minQuantity` and `m_discount`.

Constructor of subclasses

In the following code, does the constructor of `DiscountedItem` compile?

```cpp
class Item {
protected:
  std::string m_name;
  double m_price;
public:
  Item(const std::string &name, double p) : m_name(name), m_price(p) {}
};
class DiscountedItem : public Item {
  int m_minQuantity;
  double m_discount;
public:
  DiscountedItem(const std::string &name, double p, int mq, double disc) {
    m_name = name; m_price = p; m_minQuantity = mq; m_discount = disc;
  }
};
```

Constructor of derived classes

In the following code, does the constructor of `DiscountedItem` compile?

```
class Item {
  // ...
public:
  // Since `Item` has a user-declared constructor, it does not have
  // a default constructor.
  Item(const std::string &name, double p) : m_name(name), m_price(p) {}
};
class DiscountedItem : public Item {
  // ...
public:
  DiscountedItem(const std::string &name, double p, int mq, double disc)
  // Before entering the function body, `Item::Item()` is called --> Error!
  { /* ... */ }
};
```

[Best practice] Use constructor initializer lists whenever possible.

Dynamic binding

Upcasting

If `S` is a subclass of `B`:

- A pointer of type `B*` can point to a `S`.
- A reference of type `B&` can be bound to a `S`.

```
DiscountedItem di = someValue();
Item *ip = &di; // Correct.
Item &ir = di; // Correct.
```

Reason: The **is-a** relationship! A `S` can be treated as a `B`.

We move up along the inheritance hierarchy from `S` to `B`.

But on such pointers or references, only the members of `B` can be accessed.

Upcasting: Example

```
void printItemName(const Item &item) {
  std::cout << "Name: " << item.getName() << std::endl;
}

DiscountedItem di("A", 10, 2, 0.8);
Item i("B", 15);
printItemName(i); // "Name: B"
printItemName(di); // "Name: A"
```

`const Item &item` can be bound to either an `Item` or a `DiscountedItem`.

Static type and dynamic type

- **Static type** of an expression: The type known at compile-time.
- **Dynamic type** of an expression: The actual type of the object that the expression is representing. This is known at run-time.

```cpp
void printItemName(const Item &item) {
  std::cout << "Name: " << item.getName() << std::endl;
}
```

The static type of the expression `item` is `const Item`, but its dynamic type is not known until run-time (It may be `const Item` or `const DiscountedItem`).

`virtual` functions

`Item` and `DiscountedItem` have different ways of computing the amount.

```
void printItemInfo(const Item &item) {
  std::cout << "Name: " << item.getName()
            << ", price: " << item.amount(1) << std::endl;
}
```

- Which `amount` should be called? We expect:
  - `Item`'s version is called when a `Item` object is passed in.
  - `DiscountedItem`'s version is called when a `DiscountedItem` object is passed in.
- How do we define two different `amount`s and have them called correctly?

`virtual` functions

Declare `amount` in `Item` as a `virtual` function, and override it in `DiscountedItem`:

```cpp
class Item {
public:
  virtual double amount(int cnt) const { // A virtual function.
    return m_price * cnt;
  }
  // other members
};
class DiscountedItem : public Item {
public:
  double amount(int cnt) const override { // Note `override` here.
    return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
  }
  // other members
};
```

Dynamic binding

```cpp
void printItemInfo(const Item &item) {
  std::cout << "Name: " << item.getName()
            << ", price: " << item.amount(1) << std::endl;
}
```

The dynamic type of `item` is determined at run-time.

Since `amount` in `Item` is a `virtual` function, and is overridden in `DiscountedItem`, which version is called is also determined at run-time:

- If `item`'s dynamic type is `const Item`, it calls `Item::amount`.
- If `item`'s dynamic type is `const DiscountedItem`, it calls `DiscountedItem::amount`.

Dynamic binding, or **Late binding**: determine which version of a virtual function to call at run-time, based on the actual type of the object referred to by a pointer or reference to the base class.

`virtual` - `override`

To **override** (覆盖/覆写) a `virtual` function of the base class in the subclass,

- The function parameter list must be the same as that of the base class's version.
- The return type should be **identical to** (or ***covariant with***) that of the corresponding function in the base class.
  - We will talk about "covariant with" in later lectures or recitations.
- **The `const`ness on the functions should be the same!**

Not to be confused with "overloading"（重载）.

`virtual`-`override`

An overriding function is also `virtual`, even if not explicitly declared.

```
class DiscountedItem : public Item {
  virtual double amount(int cnt) const override; // correct, explicitly virtual
};
class DiscountedItem : public Item {
  double amount(int cnt) const override; // correct, `virtual` can be omitted
};
class DiscountedItem : public Item {
  double amount(int cnt) const; // also correct, but not recommended
};
```

The `override` keyword lets the compiler check if the function is truly overriding.

[Best practice] To override a virtual function, write the `override` keyword explicitly.

`virtual` destructors

```cpp
Item *ip = nullptr;
if (some_condition)
  ip = new Item(/* ... */);
else
  ip = new DiscountedItem(/* ... */);
// ...
delete ip;
```

Whose destructor should be called?

- Only looking at the static type of `*ip` is not enough.

`virtual` destructors

```
Item *ip = nullptr;
if (some_condition)
  ip = new Item(/* ... */);
else
  ip = new DiscountedItem(/* ... */);
// ...
delete ip;
```

Whose destructor should be called? - It needs to be determined at run-time!

- **To use dynamic binding correctly, you almost always need a `virtual` destructor.**

`virtual` destructors

```cpp
Item *ip = nullptr;
if (some_condition)
  ip = new Item(/* ... */);
else
  ip = new DiscountedItem(/* ... */);
// ...
delete ip;
```

- The implicitly-defined (compiler-generated) destructor is **non-`virtual`**, but we can explicitly require a `virtual` one:

  ```cpp
  virtual ~Item() = default;
  ```

- If the dtor of the base class is `virtual`, the dtor (either user-defined or compiler-generated) for the subclass is also `virtual`.

(Almost) completed `Item` and `DiscountedItem`

```
class Item {
  std::string m_name;
protected:
  double m_price = 0.0;
public:
  Item() = default;
  Item(const std::string &name, double price) : m_name(name), m_price(price) {}
  const auto &getName() const { return name; }
  virtual double amount(int n) const {
    return n * price;
  }
  virtual ~Item() = default;
};
```

(Almost) completed `Item` and `DiscountedItem`

```cpp
class DiscountedItem : public Item {
  int m_minQuantity = 0;
  double m_discount = 1.0;
public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double disc)
      : Item(name, price), m_minQuantity(minQ), m_discount(disc) {}
  double amount(int cnt) const override {
    return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
  }
};
```

Copy and move members

In a subclass's copy and move members, copy and move the base class subobject.

```cpp
class Sub : public Base {
public:
  Sub(const Sub &other)
      : Base(other), /* Sub's own members */ { /* ... */ }
  Sub &operator=(const Sub &other) {
    Base::operator=(other); // call Base's operator= explicitly
    // copy Sub's own members
    return *this;
  }
  // ...
};
```

Why `Base(other)` and `Base::operator=(other)` work?

- The parameter type is `const Base &`, which can be bound to a `Sub` object.

Synthesized copy and move members

Guess!

- What are the behaviors of compiler-generated copy and move members for a subclass?
- In what cases will they be `delete`d?

Synthesized copy and move members

Remeber that the base class's subobject is always handled first.

- What are the behaviors of compiler-generated copy and move members for a subclass?
  - First, it calls the base class's corresponding copy or move member.
  - Then, it handles the subclass's own data members.
- In what cases will they be `delete`d?
  - If the base class's corresponding copy or move member is not accessible (e.g., non-existent, or `private`),
  - If the corresponding copy or move member of any data member of the subclass is not accessible.

Slicing

Dynamic binding only happens on references or pointers to the base class.

```
DiscountedItem di("A", 10, 2, 0.8);
Item i = di; // What happens?
auto x = i.amount(3); // Which amount?
```

Slicing

Dynamic binding only happens on references or pointers to the base class.

```
DiscountedItem di("A", 10, 2, 0.8);
Item i = di; // What happens?
auto x = i.amount(3); // Which amount?
```

`Item i = di;` calls the **copy constructor of** **`Item`**

- but `Item`'s copy constructor handles only the base class part.
- So `DiscountedItem`'s own members are **ignored**, or **"sliced down"**.
- `i.amount(3)` calls `Item::amount`.

Downcasting

```
Base *bp = new Sub{};
```

If we only have a `Base` pointer, but we are quite sure that it points to a `Sub` object

- Accessing the members of `Sub` through `bp` is not allowed.
- How can we perform a "**downcasting**"?

Polymorphic class

A class is said to be **polymorphic** if it has (declares or inherits) at least one virtual function.

- Either a `virtual` normal member function or a `virtual` dtor is ok.

If a class is polymorphic, all classes inheriting from it are polymorphic.

- There is no way to "refuse" to inherit any member function, so `virtual` member functions must be inherited.
- The dtor must be `virtual` if the dtor of the base class is `virtual`.

Downcasting: For polymorphic class only

`dynamic_cast<Target>(expr)`.

```
Base *bp = new Sub{};
Sub *sp = dynamic_cast<Sub *>(bp);
Sub &dr = dynamic_cast<Sub &>(*bp);
```

- `Target` must be a **pointer** or **reference** type.
- `dynamic_cast` will perform **runtime type identification (RTTI)** to check the dynamic type of `*expr` (if `expr` is a pointer) or `expr` (if `expr` is a reference).
  - If the dynamic type is `Sub`, the downcasting succeeds.
  - Otherwise, the downcasting fails. If `Target` is a pointer type, returns a null pointer. If `Target` is a reference type, throws an exception `std::bad_cast`.

`dynamic_cast` can be very slow

`dynamic_cast` performs a runtime **check** to see if the downcasting can succeed.

It is **much slower** than other types of casting, e.g., `const_cast`, or arithmetic conversions.

[Best practice] Avoid `dynamic_cast` whenever possible.

Guaranteed successful downcasting: Use `static_cast`.

If the downcasting is guaranteed to be successful, you may use `static_cast`

```
auto sp = static_cast<Sub *>(bp); // Quicker than dynamic_cast,
// but performs no checks. If the dynamic type is not Sub, UB.
```

Avoiding `dynamic_cast`

Typical abuse of `dynamic_cast`:

```
class A {
public:
  virtual ~A() {}
};
class B : public A {};
class C : public A {};
```

```
std::string getType(const A *ap) {
  if (dynamic_cast<const B *>(ap))
    return "B";
  else if (dynamic_cast<const C *>(ap))
    return "C";
  else
    return "A";
}
```

Click here to see how large and slow the generated code is:

https://godbolt.org/z/46d613P43

Avoiding `dynamic_cast`

Use dynamic binding!

```cpp
class A {
public:
  virtual ~A() {}
  virtual std::string name() const {
    return "A";
  }
};
class B : public A {
public:
  std::string name() const override{
    return "B";
  }
};
class C : public A {
public:
  std::string name() const override{
    return "C";
  }
};
```

```cpp
auto getType(const A *ap) {
  return ap->name();
}
```

- This time: https://godbolt.org/z/MMYMT77zK

  The generated code is much simpler!

Summary

Inheritance

- Every object of the subclass contains a base class subobject.
  - Every member of the base class (except the ctors and dtor) is inherited.
- Inheritance should not break the base class's encapsulation.
  - Every constructor of the subclass calls a constructor of the base class to initialize the base class subobject **before** initializing its own data members.
  - The destructor of the subclass calls the destructor of the base class to destroy the base class subobject **after** destroying its own data members.

Summary

Dynamic binding

- Upcasting: A pointer or reference to the base class can point to or be bound to a subclass object.
- `virtual` function: A function that can be overridden by the subclass.
  - The base class and the subclass can provide different versions of this function.
- Dynamic (late) binding
  - A call to a virtual function on a pointer or reference to the base class will call a particular version of the function, based on the type of the object being referred to.