

CS100 Lecture 21

Inheritance and Polymorphism I

Contents

- Inheritance
- Dynamic binding and polymorphism

Inheritance

Example: An item for sale

```
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 netPrice(int cnt) const {  
        return cnt * m_price;  
    }  
};
```

Defining a subclass

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

- `std::size_t m_minQuantity;`
- `double m_discount;`

The net price for such an item is

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

Defining a subclass

Use inheritance to model the "is-a" relationship:

- A discounted item is an item.

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

protected members

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

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

protected members

```
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 netPrice(int cnt) const {  
        return cnt * m_price;  
    }  
};
```


Inheritance

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

- Every data member and member function, except the ctors and dtors, is inherited, **no matter what access level they have.**

What can be inferred from this?

Inheritance

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

- Every data member and member function, except the ctors and dtors, is inherited, no matter what access level they have.

What can be inferred from this?

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

Inheritance

Key points of inheritance:

- Every object of the derived class (subclass) contains a base class subobject.
- Inheritance should not break the encapsulation of the base class.
 - e.g. 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 derived classes

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

- 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 derived classes

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

- 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 derived classes

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

- 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 derived classes

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

```
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 `D` is a subclass of `B`:

- A `B*` can point to a `D`, and
- A `B&` can be bound to a `D`.

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

Reason: The **is-a** relationship! A `D` is a `B`.

But on such references or pointers, 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 real type of the object that the expression is representing. This is known at run-time.

```
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 net price.

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

- Which netPrice should be called?
- How do we define two different netPrice s?

virtual functions

```
class Item {  
public:  
    virtual double netPrice(int cnt) const {  
        return m_price * cnt;  
    }  
    // other members  
};  
class DiscountedItem : public Item {  
public:  
    double netPrice(int cnt) const override {  
        return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;  
    }  
    // other members  
};
```

Note: `auto` cannot be used to deduce the return type of `virtual` functions.

Dynamic binding

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

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

Since `netPrice` is a `virtual` function, which version is called is also determined at run-time:

- If the dynamic type of `item` is `Item`, it calls `Item::netPrice`.
- If the dynamic type of `item` is `DiscountedItem`, it calls `DiscountedItem::netPrice`.

late binding, or dynamic binding

virtual - override

To **override** (覆盖/覆写) a `virtual` function,

- 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 should be the same!

To make sure you are truly overriding the `virtual` function (instead of making a overloaded version), use the `override` keyword.

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

The `override` keyword lets the compiler check and report if the function is not truly overriding.

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

The `virtual` keyword can be omitted.

virtual destructors

```
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

```
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:

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

- If the dtor of the base class is **virtual**, the compiler-generated dtor for the derived class 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 net_price(int n) const {
        return n * price;
    }
    virtual ~Item() = default;
};
```

(Almost) completed `Item` and `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) {}
    double netPrice(int cnt) const override {
        return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
    }
};
```

Usage with smart pointers

Smart pointers are implemented by wrapping the raw pointers, so they can also be used for dynamic binding.

```
std::vector<std::shared_ptr<Item>> myItems;  
for (auto i = 0; i != n; ++i) {  
    if (someCondition) {  
        myItems.push_back(std::make_shared<Item>(someParams));  
    } else {  
        myItems.push_back(std::make_shared<DiscountedItem>(someParams));  
    }  
}
```

A `std::unique_ptr<Derived>` can be implicitly converted to a `std::unique_ptr<Base>`.

A `std::shared_ptr<Derived>` can be implicitly converted to a `std::shared_ptr<Base>`.

Copy-control

Remember to copy/move the base subobject! One possible way:

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

Synthesized copy-control members

Guess!

- What are the behaviors of the compiler-generated copy-control members?
- In what cases will they be `deleted`?

Synthesized copy-control members

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

These rules are quite natural:

- What are the behaviors of the compiler-generated copy-control members?
 - First, it calls the base class's corresponding copy-control member.
 - Then, it performs the corresponding operation on the derived class's own data members.
- In what cases will they be deleted?
 - If the base class's corresponding copy-control member is not accessible (e.g. non-existent, or private),
 - or if any of the data members' corresponding copy-control member is not accessible.

Slicing

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

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

Slicing

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

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

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

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

Downcasting

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

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

- Accessing the members of `Derived` 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 derived from it are polymorphic.

- There is no way to "refuse" to inherit any member functions, 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 Derived{};  
Derived *dp = dynamic_cast<Derived *>(bp);  
Derived &dr = dynamic_cast<Derived &>(*bp);
```

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

`dynamic_cast` can be very slow

`dynamic_cast` performs a runtime **check** to see whether the downcasting should succeed, which uses runtime type information.

This 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 dp = static_cast<Derived *>(bp); // quicker than dynamic_cast,  
// but performs no checks. If the dynamic type is not Derived, UB.
```

Avoiding `dynamic_cast`

Typical abuse of `dynamic_cast`:

```
struct A {  
    virtual ~A() {}  
};  
struct B : A {};  
struct C : 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/3367efGd7>

Avoiding `dynamic_cast`

Use a group of `virtual` functions!

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

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

- This time:

<https://godbolt.org/z/KosbcaGnT>

The generated code is much simpler!

Summary

Inheritance

- Every object of type `Derived` contains a subobject of type `Base` .
 - Every member of `Base` is inherited, no matter whether it is accessible or not.
- Inheritance should not break the base class's encapsulation.
 - Every constructor of `Derived` calls a constructor of `Base` to initialize the base class subobject **before** initializing its own data members.
 - The destructor of `Derived` calls the destructor of `Base` to destroy the base class subobject **after** destroying its own data members.

Summary

Dynamic binding

- Upcasting: A pointer, reference or smart pointer to `Base` can be bound to an object of type `Derived`.
 - static type and dynamic type
- `virtual` functions: A function that can be overridden by derived classes.
 - The base class and the derived class can provide different versions of this function.
- Dynamic (late) binding
 - A call to a virtual function on a pointer or reference to `Base` will actually call the corresponding version of that function according to the dynamic type.
- Avoid downcasting if possible.