CS100 Recitation 9

GKxx

April 18, 2022

Contents

Contents

Defining a Subclass

An item for sale:

- `std::string name;`
- `double price;`
- `std::string get_name() const;`
- `double net_price(std::size_t n) const;`

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

- `std::size_t min_quantity;`
- `double discount;`

The net price for such item is `n * price` if `n < min_quantity`, or `n * discount * price` otherwise.

Defining a Subclass

Things to consider:

- Does your class need a default constructor?
    - If so, what should be a reasonable behavior?
    - What will happen if not?
- Does your class need special copy-control?
    - Seems not.
    - But what if we have another thing called a `Basket`...?
    - What if every item has a unique id...?
- What value should `discount` have to represent ‘20% off’?

`protected` members

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

- `price` is accessible in `Discounted_item`.
- Should `name` be `protected` or `private`?
  - `private` is ok if the subclass doesn't (shouldn't) modify it. It is accessible through the public `get_name` interface.
  - `protected` is also reasonable.

The core idea is to **separate implementation details and interfaces**.

Inheritance

By defining `Discounted_item` to be a subclass of `Item`, **every object of `Discounted_item` contains an object of `Item`**.

- Every data member and member function, except the constructors, are inherited, no matter what access level they have.
- What can we derive from this?
  - When constructing an object of a subclass, one of the ctors of the base class must be called before initializing the members that the subclass declares.
  - The dtor of the subclass must call the dtor of the base class (automatically) after the members of the subclass are destroyed.
  - `sizeof(Derived) >= sizeof(Base)`.

Inheritance

Core ideas of inheritance:

- Every sub-object contains an object of the base class.
- The father has his own ways of doing things, which children cannot affect!

Inheritance and Constructors

```
class Discounted_item : public Item {
  std::size_t min_quantity = 0;
  double discount = 1.0;
 public:
  Discounted_item(const std::string &s, double p,
                  std::size_t qty, double disc)
      : Item(s, p), min_quantity(qty), discount(disc) {}
  // other members
};
```

- What if we don't call the ctor of the base class explicitly?
- Can we directly initialize the members of the base class?

```
Discounted_item(const std::string &s, double p,
                std::size_t qty, double disc)
    : name(s), price(p), min_quantity(qty),
      discount(disc) {}
```

Inheritance and Constructors

Ctors are not automatically inherited, but we can inherit them explicitly:

```
class Binary_node {
 protected:
  Expr_node *lhs, *rhs;
  Binary_node(Expr_node *left,
      Expr_node *right)
      : lhs(left), rhs(right) {}
  // other members
};
class Plus_node
    : public Binary_node {
  using Binary_node::Binary_node;
  // other members
};
```

then Plus_node has a constructor

```
Plus_node(Expr_node *left,
    Expr_node *right)
  : Binary_node(left, right)
    {}
```

and we can call it by

```
Plus_node pn(a, b);
auto pnp
    = new Plus_node(a, b);
```

Inheritance and Constructors

- Default ctor and copy ctor won't be inherited by a `using` declaration. (Why?)
- All the ctors (except default ctor and copy ctor) are inherited by a `using` declaration. But the subclass can rewrite some.
  - If the subclass has a ctor which has the same parameters as one of the ctors of the base class, then this ctor is hiding the corresponding one of the base class.
- The access-level will be preserved. (Why?)
- The `explicit` attribute, if any, is also preserved.
- How will the inherited ctors initialize the members of the subclass?

Inheritance and `friends`

Friendship cannot be inherited.

- Are you getting along well with your father's friends?

Inheritance and Copy-control

We will talk about this later...

Contents

Upcasting

A reference or pointer to base class can be bound to an object of subclass. (Why?)

```
Discounted_item di = some_value();
Item &ir = di;  // Treat di as an Item object
Item *ip = &di;
```

But on such references or pointers, only the members of base class are accessible. (Why?)

Upcasting: Example

```
inline void print_info(const Item &item) {
  std::cout << "Name: " << item.get_name()
            << ", price: " << item.net_price(1)
            << std::endl;
}
// in main
Discounted_item di = some_value();
Item i = some_other_value();
print_info(di);
print_info(i);
```

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 or variable is representing. **Known at runtime**.

```
Discounted_item di = some_value();
Item &ir = di; // ir has static type Item &,
               // but dynamic type Discounted_item.
```

Static Type and Dynamic Type

```
inline void print_info(const Item &item) {
  std::cout << "Name: " << item.get_name()
            << ", price: " << item.net_price(1)
            << std::endl;
}
```

The static type of `item` is `const Item &`, but the dynamic type is unknown.

virtual Functions

```
inline void print_info(const Item &item) {
  std::cout << "Name: " << item.get_name()
            << ", price: " << item.net_price(1)
            << std::endl;
}
```

Which net_price is called?

virtual Functions

```
class Item {
 public:
  virtual double net_price(std::size_t n) const;
  // other members
};
class Discounted_item : public Item {
 public:
  virtual double net_price(std::size_t n) const override;
  // other members
};
```

`virtual` Functions

- The dynamic type of parameter `item` is runtime-determined.
- Since `net_price` is a `virtual` function, which one is called is determined at **runtime**, so that the correct version is called.
- **late-binding**, or **dynamic-binding**.

Overriding a `virtual` Function

To `override` a `virtual` function,

- The function must have parameters the same as the function in the base class has.
- The return-type of the function should be either **identical to** or **covariant with** (What's this?) that of the corresponding function in the base class.
- Don't forget the `const` qualifier!

To make sure that your function overrides the one in the base class, use the `override` keyword.

Overriding a `virtual` Function

- An overriding function is still `virtual`, even if not explicitly declared.
- The best practice is to explicitly write '`virtual`' and '`override`'.
  - The `override` keyword lets the compiler check and report if the function is not actually overriding.
- Distinguish between **overriding**, **overloading** and '**hiding**'.
  - Avoid confusing cases in your program! Don't invite troubles for yourself.

virtual Destructors

```
Base *bp = some_value();
delete bp;
```

which destructor should be called by '`delete bp`'?

virtual Destructors

```
Base *bp = some_value();
delete bp;
```

which destructor should be called by '`delete bp`'?

- To make dynamic binding work correctly, the destructors must be `virtual`!
- The synthesized destructor is **non-virtual**, but we can:

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

- If the dtor of the base class is `virtual`, the synthesized destructor is also `virtual`.

Inheritance and Copy-control

Remember to copy the base part correctly! One possible way:

```
class Derived : public Base {
 public:
  Derived(const Derived &d)
    : Base(d), /* members of Derived */ {}
  Derived &operator=(const Derived &d) {
    Base::operator=(d);
    // copy members of Derived
    return *this;
  }
};
```

Synthesized Copy-control Functions

- When will the compiler synthesize a copy-control function?
- What's the behavior of them?
- When will the compiler mark them as deleted?
- What about default ctors?

Slicing

Suppose `Base` and `Derived` have a `virtual` function `foo`.

```
Derived d = some_value();
Base b = d;
b.foo();     // Base::foo or Derived::foo?
```

When using an object of a subclass to initialize or assign to an object of the base class, the copy-ctor or copy-assignment operator **of the base class** is called.

- Therefore, the sub-part of the object is ignored, or **sliced down**.
- Dynamic binding won't happen.

Downcasting

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

We cannot access the members of the subclass through a pointer to the base class. We need a **downcasting**.

- As long as the following conditions are satisfied, you can make a downcasting:
  - The pointer or reference to the base class is **indeed** bound to an object of the subclass.
  - The base class and the subclass are polymorphic, which means that there is at least one `virtual` function.
- You can make a downcasting by `dynamic_cast`:

```
Derived *dp = dynamic_cast<Derived *>(bp);
Derived &dr = dynamic_cast<Derived &>(*bp);
```

Downcasting

- `dynamic_cast` may have a significant runtime cost.
- Several common ways to avoid `dynamic_cast`, like writing a group of `virtual` functions.
- *Effective C++* Item 27 talks about type-casting.
- *More Effective C++* Item 31 talks about some more complicated cases: Making functions virtual with respect to more than one object.

Notice

Avoid `dynamic_cast`, especially in performance-sensitive code.

Contents

Pure `virtual` Functions

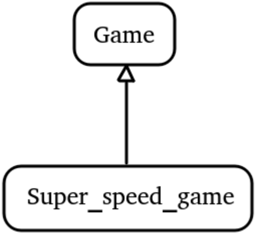
By defining a function to be =0, it is defined as a **pure virtual** function.

- A class with at least one pure virtual function is an **abstract class**.
- A pure virtual function can be overridden in a subclass. But if it is not overridden, the subclass is still abstract.
- Creating objects of a type that is an abstract class is **not allowed**.

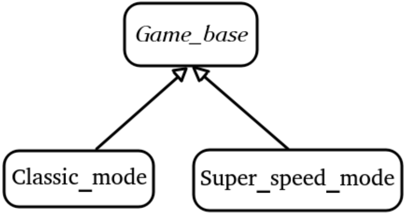
Generally, virtual functions in the base class that do not have a reasonable behavior should be pure virtual, and such class should be abstract.

Example: Greedy Snake

"A super-speed game is a game."

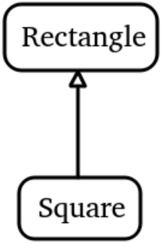


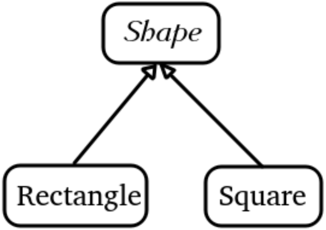
"A classic-mode game is a game. A super-speed game is also a game."



It turns out that the super-speed mode has too many differences from the classic-mode, so I **refactored** the program according to the diagram on the right.

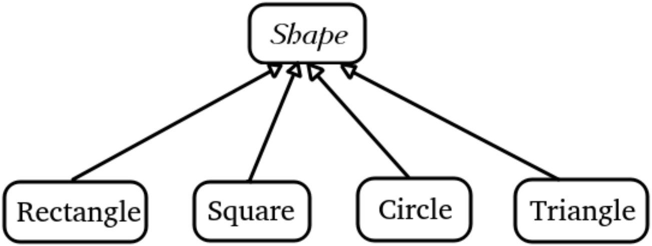
Which One is Better?





Which One is Better?

- "A square **is a** rectangle" is correct, but sometimes this is deceptive. (*Effective C++* Item 32, very important)
- The structure on the right can be extended easily: (**reusability**)



A Pure `virtual` Destructor

Sometimes a class should be abstract, but there seems to be no reasonable choice over which function should be pure virtual.

A Pure `virtual` Destructor

Sometimes a class should be abstract, but there seems to be no reasonable choice over which function should be pure virtual.

- Define the destructor to be pure virtual, and provide another definition.

```
class Base {
 public:
  virtual ~Base() = 0;
};
Base::~Base() {}
```

In fact, we can provide definitions for pure virtual functions.

More on Inheritance...

- There is still one thing that is magic to us: the '`public`' keyword:

```
class Discounted_item : public Item {};
```

- `public` inheritance models 'is-a', while `private` inheritance models 'is-implemented-in-terms-of'. What's that?

Contents

Default Arguments

```
void create_window(std::size_t height = 24,
                   std::size_t width = 80) {
  // create a window with given height and width
}
```

If the caller omit the ‘width’ argument

```
create_window(30);
```

then `width` will be set to default value 80. If both arguments are omitted

```
create_window();
```

then `height` is set to 24 and `width` 80.

Default Arguments

- Only the last few parameters can have default arguments.

```
void fun(int a = 42, int b); // Error
```

- Functions that have default arguments will be treated as **overloading functions**. For the `create_window` function, it is the same as

```
void create_window();
void create_window(std::size_t height);
void create_window(std::size_t height,
                   std::size_t width);
```

Default Arguments

Member functions can also have default arguments:

```
class Vector {
 public:
  Vector(std::size_t n, int val = 0)
      : m_size(n), m_capacity(n),
        m_data(new int[n]{}) {
    for (std::size_t i = 0; i < n; ++i)
      m_data[i] = val;
  }
  // other members
};
```

It will be treated as if there are two constructors

```
Vector::Vector(std::size_t);
Vector::Vector(std::size_t, int);
```

Default Argument Declaration

A function may be declared multiple times, but default arguments should **not** be redeclared.

```
class Vector {
 public:
  Vector(std::size_t n, int val = 0);
};
Vector::Vector(std::size_t n, int val = 0) // Error.
    : m_size(n), m_capacity(n), m_data(new int[n]{}) {
  std::fill_n(m_data, n, val);
}
```

Default Argument Declaration

A function may be declared multiple times, but default arguments should **not** be redeclared. (Why?)

```
class Vector {
 public:
  Vector(std::size_t n, int val = 0);
};
Vector::Vector(std::size_t n, int val) // Correct.
    : m_size(n), m_capacity(n), m_data(new int[n]{}) {
  std::fill_n(m_data, n, val);
}
```

Default Argument Declaration

Although it seems weird, subsequent declarations can have additional default arguments.

```
void create_window(std::size_t height,
                   std::size_t width = 80);
void craete_window(std::size_t height = 24, // OK.
                   std::size_t width) {
  // ...
}
```

Defaults can be specified only when all parameters to the right already have defaults.

Contents

Command-Line Arguments

Suppose you are the author of `g++`. When the user type

`g++ -o hello hello.cpp`

in the terminal, there should be a way to let your program get this command.

Command-Line Arguments

Suppose you are the author of `g++`. When the user type
`g++ -o hello hello.cpp`
in the terminal, there should be a way to let your program get this command.

```
int main(int argc, char **argv) {
  // ...
}
```

- `argv` is an array of strings. In this example, `argv = {"g++", "-o", "hello", "hello.cpp"}`.
- `argc` is the number of strings in the array `argv`.
- `char *argv[]` is the same as `char **argv`.

Command-Line Arguments

The only two correct versions of the `main` function:

```
int main();
int main(int argc, char **argv);
```

Assertion

```
#include <cassert>
int main() {
  int a, b;
  std::cin >> a >> b;
  assert(b != 0);
  int c = a / b;
  // ...
}
```

C++11 also provides compile-time assertion `static_assert`, but it's too early for you now... (We used this in Problem 2 to detect whether your `Shape` class is abstract.)

Some Helpful Macros

To disable assertions, we can use the NDEBUG macro.

```
int main() {
  int a, b;
  std::cin >> a >> b;
#define NDEBUG
  assert(b != 0); // This assertion will not be performed.
#undef NDEBUG
  assert(b != 0); // This assertion will be performed.
  // ...
}
```

Some Helpful Macros

- `__LINE__`: `int`, the line number.
- `__func__`: `const char []`, the name of the current function.
- `__FILE__`: `const char []`, the name of the current file.
- `__TIME__`: `const char []`, the current time.

New-style Alias Declaration

```
using LL = long long;
```

The new-style type alias declaration is more clear:

```
typedef int arr_t[10];
using arr_t = int[10];
```

The `using` type alias declaration can also be a template, but `typedef` cannot.

Type Alias Member

```
class Vector {
 public:
  using size_type = std::size_t;
  using value_type = int;
  using pointer = int *;
  using reference = int &;
  // other members
};
int main() {
  Vector v = some_value();
  for (Vector::size_type i = 0;
       i < v.size(); ++i)
    // do something
}
```

- Access modifiers also apply to type alias members.
- To access a type alias member, use *class-name* :: *type-member*.