Ve 280

Programming and Introductory Data Structures

Synthesized Maintenance Methods;

Derived Class Maintenance Methods;

Function Objects

Outline

- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor
- Derived Class Constructor, Copy Constructor, Assignment Operator, and Destructor
- Function Objects

Synthesized Default Constructor

- If we do not explicitly define any constructors, what will happen?
  - Answer: The compiler will automatically synthesize a default constructor, i.e., a constructor that takes no argument

```
class foo {
  int i;
  string s;
public:
  int get_int();
  string get_str();
};
```

```
foo f;
```

Call the synthesized default constructor

Will synthesize a default constructor `foo::foo()`

Synthesized Default Constructor

- Members of built-in types (int, double, etc.) are uninitialized
- Members of class types are initialized using their default constructor

```
class foo {
  int i;
  string s;
public:
  int get_int();
  string get_str();
};
```

```
foo f;
```

`f.i` is uninitialized

`f.s` is an empty string

- A good practice is to define a default constructor ourselves

Synthesized Default Constructor

- If a class defines at least one constructor, then the compiler will not generate the default constructor.

```
class foo {
  int i;
public:
  foo(int _i): i(_i) {}
};
```

Note: define a constructor taking one argument

```
foo f;
foo *fp = new foo;
// Both wrong: no
// default constructor
// synthesized
```

- In practice, if other constructors are defined, it is almost always right to provide a default constructor

Synthesized Copy Constructor

- If we don't define a copy constructor explicitly, the compiler synthesizes one for us
- Unlike the synthesized default constructor, a copy constructor is synthesized even if we define other constructors
- What's the behavior of the synthesized copy constructor?
  - Answer: Member-wise copy
  - For a built-in/pointer type, it directly copies the value
  - For a class type, it calls the copy constructor

This causes **shallow copy** when passing class object as function argument as we discussed before

Synthesized Assignment Operator

- If we don't define an overloaded assignment operator explicitly, the compiler also synthesizes one for us

- The behavior is same as synthesized copy constructor
  - For a built-in/pointer type, it directly copies the value
  - For a class type, it calls the copy constructor

Synthesized Destructor

- Surprising result: there is always a synthesized destructor even if you explicitly write one
- Effect: destroys the members in **reverse order** from which they are declared in the class
  - For class member, call the member's destructor
  - For built-in or pointer type, just destroy the member. In particular, does not delete the object pointed to by a pointer member
- What happens when we write an explicit destructor `~foo() { ... }`?
  - First, do everything in `~foo() { ... }`
  - Then, call the synthesized destructor to destroy members

Outline

- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor
- Derived Class Constructor, Copy Constructor, Assignment Operator, and Destructor
- Function Objects

Synthesized Derived-Class Default Constructor

- If you don't explicitly define any constructor for a derived class, compiler synthesizes a default one for you
  - It calls the default constructor of base to initialize base part
  - It initializes derived class members. Initialization rules same as the normal synthesized default constructor

```
class Base
{
  int i;
public:
  Base(): i(1) { }
};
```

```
class Derived: public Base
{
  double d;
public:
  double get_d() const;
};
```

Initialization order:

- First, base class
- Then, the members of derived class

```
Derived o;
```

i (Base)
d

Define Default Constructor of Derived Class

- Because **`Derived`** has members of built-in type, we should define our own default constructor for **`Derived`**
- We may ignore the initialization to the base part
  - Then, the default constructor of **`Derived`** calls the default constructor of **`Base`** to initialize its base-class part.

```
class Base
{
  int i;
public:
  Base(): i(1) { }
};
```

```
class Derived: public Base
{
  double d;
public:
  Derived(): d(2) { }
};
```

Constructor Initializing Base Part

- We can also define constructor that can initialize base part

```
class Base {
  int i;
public:
  Base(int _i = 0):
    i(_i) { }
};
```

```
class Derived: public Base {
  double d;
public:
  Derived(int _i =0, double _d = 0):
    Base(_i), d(_d) { }
};
```

- We indirectly initialize the inherited members of **`Derived`** by using the base class constructor
- We can declare an object of **`Derived`** as

```
Derived x(1, 2.0);
```

Constructor Initializing Base Part

- Rule: Only an immediate base class may be initialized

```
Class B: public A { ... };
Class C: public B { ... };
```

- Even though every C object contains an A part, the constructors for C may not initialize the A part directly.

Respect the Interface

- Derived-class constructors should not assign to the members of its base class, even if members in base class are public or protected.

```
class A {
protected:
  int i;
public:
  A(int _i = 0): i(_i) { }
};
```

```
class B: public A {
  double d;
public:
  B(int _i =0, double _d = 0){
    i = _i; // Not Good!
    d = _d;
  }
};
```

Synthesized Derived-Class Copy Constructor

- If we don’t define the copy constructor, the compiler just synthesizes one for us.
  - The base part is copied by using the base class' copy constructor
  - The derived class members are copied using the same rules as the normal synthesized copy constructor

Define Derived-Class Copy Constructor

- If we define the copy constructor, it usually should explicitly use the base-class copy constructor to initialize the base part of the object

```
class Base {
  int i;
public:
  Base(const Base &b): i(b.i) {}
};
```

```
class Der: public Base {
  double d;
public:
  Der(const Der &dr): Base(dr), d(dr.d) {}
};
```

- Legal: use a Der& where we expect a Base&
- Initialize base part of derived class object

Derived-Class Assignment Operator

- If not defined, the complier synthesizes one
- If the derived class defines its own assignment operator, then that operator must assign the base part explicitly

```
class Base {
  int i;
public:
  Base &operator=(const Base &rhs);
};
```

```
Der &Der:operator=(const Der &rhs) {
  if(this != &rhs) {
    Base::operator=(rhs);
    d = rhs.d;
  }
  return *this;
};
```

Derived-Class Destructor

- Different from the constructor and assignment operator:
  - The derived destructor is **never** responsible for destroying the members of its base object.
  - The compiler always implicitly invokes the destructor for the **base part** of a derived object.
  - Oder: opposite to the constructor. First derived-class destructor, then base-class destructor

```
class Derived: public Base {
public:
  // Base::~Base invoked automatically
  ~Derived(){ /* clean up derived members */ }
};
```

Virtual Destructors

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

- Why need?
  - When deleting a pointer that points to a **dynamically allocated** object, it may actually point to a derived class object
  - We need to ensure the proper destructor is run
- A base class almost always needs a virtual destructor
  - This is an exception of the **Rule of Big Three**
  - For the base case, can only have a (virtual) destructor, but no copy constructor and assignment operator

Virtual Destructors

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

- Good practice:
  - The root class of an inheritance hierarchy should define a virtual destructor even if the destructor has no work to do

No Virtual Constructor

- Constructors cannot be defined as virtual. Why?
  - Constructors are run before the object is fully constructed. While the constructor is running, the object's dynamic type is not complete
  - Without exact type information, it makes no sense to define virtual function

Outline

- Synthesized Default Constructor, Copy Constructor, Assignment Operator, and Destructor
- Derived Class Constructor, Copy Constructor, Assignment Operator, and Destructor
- Function Objects

Motivation

- We want to write a generic "has_a" function using function pointers by using the `list_t` ADT from project 3.

```
bool has_a(list_t l, bool (*pred)(int));
```

  - `pred` is a pointer to a predicate function taking a single integer argument, returning a Boolean result
  - `has_a` returns true if and only if `l` has an element that makes `pred` true
- We can use `has_a` to see if a list `l` has odd elements, by using predicate **`isOdd`**

```
has_a(l, isOdd);
```

```
bool isOdd(int n) {
  return (n%2);
}
```

Motivation

- The recursive definition of `has_a` is:
  - The empty list should return false
  - A non-empty list should return true if and only if:
    - The first element makes `pred` true
    - or applying `has_a` on the rest of the list returns true

```
bool has_a(list_t l, bool (*pred)(int)) {
  if(list_isEmpty(l)) return false;
  else if(pred(list_first(l))) return true;
  else return has_a(list_rest(l), pred);
}
```

Motivation

- **<u>Question</u>**: How would you use `has_a` to see if a list `l` has any elements larger than 2? How would you use it to see if a list has any elements larger than 42?

```
bool has_a(list_t l, bool (*pred)(int)) {
  if(list_isEmpty(l)) return false;
  else if(pred(list_first(l))) return true;
  else return has_a(list_rest(l), pred);
}
```

Motivation

- **Answer**: Write some new predicates!

```
bool larger2(int n) {
   return (n>2);
}

bool larger42(int n) {
   return (n>42);
}
```

This is not good. We really should be able to generalize more by writing a single predicate that is "larger_than", and use that single predicate no matter what the larger than target is.

Motivation

- **<u>Answer</u>**: Write some new predicates!

```
bool larger2(int n) {
   return (n>2);
}

bool larger42(int n) {
   return (n>42);
}
```

Worse yet, we have to know how much "larger" the elements needed to be at compile time – and we might not know that!

Question: why don't we define the following?

```
bool larger(int n, int l) {
   return (n > l);
}
```

Motivation

- One way to solve the problem of creating a larger_than function with function pointers requires a global variable:

```
int larger_target;  // global variable
bool larger(int n) {
  return (n>larger_target);
}
```

- To use this, we would do something like this:

```
list_t l;
...
cin >> larger_target; // get upper bound
cout << has_a(l, larger);
```

Motivation

- To avoid using a global variable, we want a "function-creating function" – one that, given an integer `i`, returns a predicate that takes one integer argument, N, and returns true of N > `i`.

- There is no way to do this with C++ functions

- But, we **can** do it with the class mechanism plus one neat trick – **operator overloading**

Overloading Function Call

- Recall that we can overload many operators of a class. E.g.,
  `Foo &operator=(const Foo &f);`
- It turns out that the "function-call" is just another operator and we can overload it.
- Suppose we have a class called `Multiply2`, with no private members and only one public one:

```
class Multiply2 {
 public:
  int operator() (int n);
};
```

Overloading Function Call

```
class Multiply2 {
 public:
  int operator() (int n);
};
```

- This public method overloads the "function-call" operator on **instances** of `Multiply2` – the method takes a single integer argument, and returns an integer result
- Here's the body of that method:

```
int Multiply2::operator() (int n) {
  return n*2;
}
```

The function-call method takes an integer argument and returns twice that argument.

Function Objects

- So, we can do this:

```
Multiply2 doubleIt;
cout << doubleIt(4) << endl;
```

- The second line looks like a function call; however, `doubleIt` is **not** a function. Rather, it is an instance of the class `Multiply2`.
- The class `Multiply2` has defined the "function-call operator“. So, we invoke that operator, passing the argument 4.
- The method body returns 2*4, printing 8 to the terminal.
- Objects that overload the function-call operator are called **function objects**, or sometimes **functors**.

Function Objects

Implementation

- So far, this is not very interesting
- However, unlike functions, objects can have **per-object state**, which allows us to specialize on a per-object basis
- For example, suppose we defined the class `MultiplyN` to be:

```
class MultiplyN {
  int factor;
  MultiplyN() {}  // Private ctor
 public:
  MultiplyN(int n);
  int operator()(int n);
};
```

The idea here is that when we create instances of `MultiplyN`, we bind the "multiplicative factor" to some constant, and later can multiply numbers by that factor.

Function Objects

Implementation

- So far, this is not very interesting
- However, unlike functions, objects can have **per-object state**, which allows us to specialize on a per-object basis
- For example, suppose we defined the class `MultiplyN` to be:

```
class MultiplyN {
  int factor;
  MultiplyN() {}  // Private ctor
 public:
  MultiplyN(int n);
  int operator()(int n);
};
```

Private ctor ensures that you must construct the object using the version **`MultiplyN(int n)`**, because we need to assign initial value for **`factor`**

Function Objects

Implementation

- So, the constructor would be:

```
MultiplyN::MultiplyN(int n) {
   factor = n;
}
```

- And the function-call operator:

```
int MultiplyN::operator() (int n) {
   return n*factor;
}
```

Function Objects

Implementation

- Now, we can use this new class to "generate" specialized functors:

Note the different use of "()"

```
MultiplyN doubleIt(2);
MultiplyN tripleIt(3);
cout << doubleIt(4) << endl;
cout << tripleIt(4) << endl;
```

This () calls the constructor

This () calls function-call operator

- Which prints to the screen:

```
8
12
```

Function Objects

Implementation

- Finally, we can use functors to write our generic `has_a` routine.
- We first define an abstract `Predicate` class, specifying that a `Predicate` must provide an appropriate function-call method:

```
class Predicate {
 public:
  virtual bool operator()(int n) = 0;
};
```

Function Objects

Implementation

- Now, define `has_a` in terms of this `Predicate` class:

```
bool has_a(list_t l, Predicate &pred) {
  if(list_isEmpty(l)) return false;
  else if(pred(list_first(l)))
    return true;
  else return has_a(list_rest(l), pred);
}
```

Note: The body of `has_a` is **exactly the same** as it was before. But, rather than take a function pointer, it takes a functor.

Function Objects

Implementation

- If we want to use `has_a` to check for entries greater than some specific value, we can write a subtype of `Predicate`:

```
class GreaterN : public Predicate {
  int target;
  GreaterN() {}
 public:
  GreaterN(int n);
  bool operator() (int n);
};
GreaterN::GreaterN(int n) {
  target = n;
}
bool GreaterN::operator() (int n) {
  return (n > target);
}
```

Function Objects

Implementation

- We can also check for entries less than some specific value by writing another subtype of `Predicate`:

```
class LessN : public Predicate {
   int target;
   LessN() {}
 public:
   LessN(int n);
   bool operator() (int n);
};
LessN::LessN(int n) {
   target = n;
}
bool LessN::operator() (int n) {
   return (n < target);
}
```

Function Objects

Implementation

- Now, given a list, we can find out if it has elements greater than 2:

```
list_t l;
... // fill in the list
GreaterN gt2(2);
cout << has_a(l, gt2);
```

- or 42:

```
GreaterN gt42(42);
cout << has_a(l, gt42);
```

Function Objects

Implementation

- We can also test if a list has values less than 2:

```
list_t l2;
... // fill in the list
LessN lt2(2);
cout << has_a(l2, lt2);
```

- or 42:

```
LessN lt42(42);
cout << has_a(l2, lt42);
```

Function Objects

Implementation

- We can even get limits from the user:

```
list_t l;
... // fill in the list

int GT_Limit, LT_Limit;
cin >> GT_Limit >> LT_Limit; // user input

GreaterN gt(GT_Limit);
LessN lt(LT_Limit);

cout << has_a(l, gt);
cout << has_a(l, lt);
```

So, the ability of objects to carry per-object state **plus** override the "function call" operator gives us the equivalent of a "function factory".

Function Objects

Implementation

- We can even get limits from the user:

```
list_t l;
... // fill in the list

int GT_Limit, LT_Limit;
cin >> GT_Limit >> LT_Limit; // user input

GreaterN gt(GT_Limit);
LessN lt(LT_Limit);

cout << has_a(l, gt);
cout << has_a(l, lt);
```

This allows us to generalize predicates without resorting to the global variable trick.

Reference

- **C++ Primer (4th Edision)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
  - Chapter 12.4.3 The Default Constructor
  - Chapter 14.8 Call Operator and Function Objects
  - Chapter 15.4 Constructors and Copy Control