Advanced Programming Concepts with C++

CSI 2372

Tutorial # 6

Selected exercises from chapters 13

uOttawa

Ahmedou Jreivine

Exercise 13.1:

- **What is a copy constructor? When is it used?**
- **Answer:**
  - A copy constructor is a constructor whose first parameter is a reference to the class type and any additional parameters have default values.
  - The copy constructor may be used when we use copy initialization.

Exercise 13.2:

- Explain why the following declaration is illegal:

  Sales_data::Sales_data(Sales_data rhs);

- **Answer:**
  - The parameter of the constructor should be a reference, otherwise the call would never succeed.
  - To call the copy constructor, we would need to use the copy constructor to copy the argument, but to copy the argument, we would need to call the copy constructor, and so on indefinitely.

Exercise 13.3:

- What happens when we copy a StrBlob? What about StrBlobPtrs?

- **Answer:**
  - When we copy a StrBlob, the underlying smart pointer is copied. But the vector to which the pointer pointed is not copied, which means both StrBlob objects will use the same vector. This may not be what we want.
  - When we copy a StrBlobPtr, the same thing happens as copying a StrBlob, but this may be what we want.

Exercise 13.4:

- Assuming `Point` is a class type with a `public` copy constructor, identify each use of the copy constructor in this program fragment:

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

Answer 13.4:

```
Point global;
Point foo_bar(Point arg)  // copy initialize `arg` may use copy
constructor
{
      Point local = arg, *heap = new Point(global);
      // copy initialize `local` may use copy constructor
      // copy initialize `heap` may use copy constructor
      *heap = local;
      Point pa[ 4 ] = { local, *heap };
      // brace initialize the elements in array `pa` will use copy
      // initialization, which may use copy constructor
      return *heap;
      // copy initialize the return object may use copy constructor
}
```

Exercise 13.5:

- Given the following sketch of a class, write a copy constructor that copies all the members. Your constructor should dynamically allocate a new string (§ 12.1.2, p. 458) and copy the object to which ***ps*** points, rather than copying ***ps*** itself.

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()): ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

Answer 13.5:

```
#include <string>
#include <iostream>
class HasPtr {
public:
	HasPtr(const std::string &s = std::string()) : ps(new std::string(s)), i(0) {}
	HasPtr(const HasPtr &ori) : ps(new std::string(*ori.ps)), i(ori.i) {}
	const std::string &get() const { return *ps; }
	void set(const std::string &s) { *ps = s; }
private:
	std::string *ps;
	int i;
};
int main() {
	HasPtr hp1("World");
	HasPtr hp2 = hp1;
	hp1.set("Hello");
	std::cout << hp1.get() << " " << hp2.get() << std::endl;
	return 0;
}
```

Exercise 13.6:

- What is a copy-assignment operator? When is this operator used? What does the synthesized copy-assignment operator do? When is it synthesized?
- **Answer:**
- What is a copy-assignment operator?
  - It is the **=** operator used to control how objects of its class are assigned.
- When is this operator used?
  - It is used when we assign value to an object.
- What does the synthesized copy-assignment operator do?
  - It disallows assignment for some classes. Otherwise, it assigns each nonstatic member of the right-hand object to the corresponding member of the left-hand object using the copy-assignment operator for the type of that member.
- When is it synthesized?
  - The compiler synthesizes a copy-assignment operator if the class does not define its own.

Exercise 13.8:

- Write the assignment operator for the `HasPtr` class from exercise 13.5 in § 13.1.1 (p. 499). As with the copy constructor, your assignment operator should copy the object to which `ps` points.

Answer 13.8:

```
#include <string>
#include <iostream>
class HasPtr {
public:
	HasPtr(const std::string &s = std::string()) : ps(new std::string(s)), i(0) {}
	HasPtr(const HasPtr &ori) : ps(new std::string(*ori.ps)), i(ori.i) {}
	HasPtr &operator=(const HasPtr &rhs);
	const std::string &get() const { return *ps; }
	void set(const std::string &s) { *ps = s; }
private:
	std::string *ps;
	int i;
};
```

Answer 13.8:

```
HasPtr &HasPtr::operator=(const HasPtr &rhs) {
	delete ps;
	ps = new std::string(*rhs.ps);
	i = rhs.i;
	return *this;
}
int main() {
	HasPtr hp1("World“);
	HasPtr hp2 = hp1;
	HasPtr hp3;
	hp3 = hp1;
	hp1.set("Hello");
	std::cout << hp1.get() << std::endl;
	std::cout << hp2.get() << std::endl;
	std::cout << hp3.get() << std::endl;
	return 0;
}
```

Exercise 13.9:

- What is a destructor? What does the synthesized destructor do? When is a destructor synthesized?
- **Answer:**
- What is a destructor?
  - A destructor is a member function with the name of the class prefixed by a tilde (~) and has no return value and takes no parameters.
- What does the synthesized destructor do?
  - The synthesized destructor disallows objects of the type from being destroyed for some classes. Otherwise, it has an empty function body and the members are automatically destroyed after the empty destructor body is run.
- When is a destructor synthesized?
  - The compiler defines a synthesized destructor for any class that does not define its own destructor.

Exercise 13.10:

- What happens when a `StrBlob` object is destroyed? What about a `StrBlobPtr`?
- **Answer:**
- When a StrBlob object is destroyed, the shared pointer member is destroyed by calling the pointer's destructor, thus decreasing the reference count of the shared pointer, if the count is zero, then the vector pointed by the smart pointer is destroyed too.
- When a StrBlobPtr object is destroyed, the weak pointer member is destroyed by calling the pointer's destructor, the vector pointed by the smart pointer is not affected.

Exercise 13.11: Add a destructor to your `HasPtr` class from the previous exercises.

- **Answer:**

```
#include <string>
#include <iostream>
class HasPtr {
public:
        HasPtr(const std::string &s = std::string()) : ps(new std::string(s)), i(0) {}
        HasPtr(const HasPtr &ori) : ps(new std::string(*ori.ps)), i(ori.i) {}
        ~HasPtr();
        HasPtr &operator=(const HasPtr &rhs);
        const std::string &get() const { return *ps; }
        void set(const std::string &s) { *ps = s; }
private:
        std::string *ps;
        int i;
};
HasPtr::~HasPtr() { delete ps; }
```

Answer 13.11: Con.

```
HasPtr &HasPtr::operator=(const HasPtr &rhs) {
	delete ps;
	ps = new std::string(*rhs.ps);
	i = rhs.i;
	return *this;
}
int main() {
	HasPtr hp1 = "World";
	HasPtr hp2 = hp1;
	HasPtr hp3;
	hp3 = hp1;
	hp1.set("Hello");
	std::cout << hp1.get() << std::endl;
	std::cout << hp2.get() << std::endl;
	std::cout << hp3.get() << std::endl;
	hp1 = hp1;
	std::cout << "After `hp1 = hp1`: " << hp1.get() << std::endl;
	return 0;
}
```

Exercise 13.12:

- How many destructor calls occur in the following code fragment?

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
    return item1.isbn() != item2.isbn();
}
```

- **Answer:**
  - 3 destructors are called on accum, item1 and item2

Exercise 13.13:

- A good way to understand copy-control members and constructors is to define a simple class with these members in which each member prints its name:

```
struct X {
    X() {std::cout << "X()" << std::endl;}
    X(const X&) {std::cout << "X(const X&)" <<
    std::endl;}
};
```

Add the copy-assignment operator and destructor to X and write a program using X objects in various ways: Pass them as nonreference and reference parameters; dynamically allocate them; put them in containers; and so forth. Study the output until you are certain you understand when and why each copy-control member is used. As you read the output, remember that the compiler can omit calls to the copy constructor.

Answer: 13.13

```
#include <iostream>
#include <vector>
#include <initializer_list>
using namespace std;
struct X {
    X() { std::cout << "X()" << std::endl; }
    X(const X&) { std::cout << "X(const X&)" << std::endl; }
    X& operator=(const X&) { std::cout << "X& operator=(const X&)" << std::endl; return *this; }
    ~X() { std::cout << "~X()" << std::endl; }
};
void f(const X &rx, X x)
{
    std::vector<X> vec;
    vec.reserve(2);
    vec.push_back(rx);
    vec.push_back(x);
}
```

```
int main()
{
    X *px = new X;
    f(*px, *px);
    delete px;
    cin.get();
    return 0;
}
```

Exercise 13.14:

- Assume that `numbered` is a class with a default constructor that generates a unique serial number for each object, which is stored in a data member named `mysn`. Assuming `numbered` uses the synthesized copycontrol members and given the following function:

```
void f (numbered s) { cout << s.mysn << endl; }
```

what output does the following code produce?

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

uOttawa

Ahmedou Jreivine

Answer 13.14:

```
#include <iostream>
using std::cout; using std::endl;
class numbered {
public:
      numbered() : mysn(++sn) {}
      int mysn;
private:
      static int sn;
};
int numbered::sn = 0;
void f(numbered s) { cout <<"the number of objects is: "<< s.mysn << endl; }
int main() {
      numbered a, b = a, c = b;
      f(a);  // 1
      f(b);  // 1
      f(c);  // 1
      numbered d;
      f(d);  // 2
      return 0;
}
```

Exercise 13.15:

- Assume ***numbered*** has a copy constructor that generates a new serial number. Does that change the output of the calls in the previous exercise? If so, why? What output gets generated?

Answer 13.15

```
#include <iostream>
using namespace std;
using std::cout; using std::endl;
class numbered {
public:
      numbered() : mysn(++sn) {}
      numbered(const numbered &) : mysn(++sn) {}
      int mysn;
      private:
      static int sn;
};
int numbered::sn = 0;
void f(numbered s) {
// `s` is copy initialized from argument, thus `s.mysn` is not the same with the argument we passed in.
cout << "the number of objects is: " << s.mysn << endl;
}
int main() {
            numbered a, b = a, c = b;  // a.mysn = 1, b.mysn = 2, c.mysn = 3
            f(a);  // 4
            f(b);  // 5
            f(c);  // 6
            numbered d;
            f(d);  // 8
            return 0;
}
```

Exercise 13.16:

- What if the parameter in `f` were `const numbered&?`
- Does that change the output? If so, why? What output gets generated?

- Answer:
  - Yes, the output will change. Because no copy operation happens within function f. Thus, the three Output are the same.
- In other words:
  - Because the function f haven't any copy operators. Thus, the output are the same when pass each object to f.

Answer 13.16

```
#include <iostream>
using namespace std;
class numbered {
public:
    numbered() : mysn(++sn) {}
    numbered(const numbered &) : mysn(++sn) {}
    int mysn;
private:
    static int sn;
};
int numbered::sn = 0;
void f(const numbered &s) {  // Yes, the output will change,
                             //the reference will not use copy initialization
    cout << s.mysn << endl;
}
int main() {
    numbered a, b = a, c = b;  // a.mysn = 1, b.mysn = 2, c.mysn = 3
    f(a);  // 1
    f(b);  // 2
    f(c);  // 3
    numbered d;
    f(d);  // 4
    return 0;
}
```

Exercise 15

1. What is a virtual member?
   - A virtual member in a base class expects its derived class define its own version.
   - In particular base classes ordinarily should define a virtual destructor, even if it does no work.
2. How does the protected access specifier differ from private?
   - private members : accessible for base class and friend class
   - protected members: accessible for base class, friend and derived classes.

Exercise 15.3:

- **Define your own versions of the `Quote` class and the `print_total` function.**

```
#include <string>
#include <iostream>
using namespace std;
class Quote {
public:
    Quote() = default;
    Quote(const string &book, double pri): bookNo(book), price(pri) { }
    virtual ~Quote() = default;
    string isbn() const { return bookNo; }
    virtual double net_price(size_t n) const { return n * price; }
protected:
    double price = 0.0;
private:
    string bookNo;
};
```

```
class Bulk_quote : public Quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const string &book, double pri,size_t qty, double disc): Quote(book, pri), min_qty(qty), discount(disc)
    { }
    double net_price(size_t n) const override {
        if (n >= min_qty)
        return n * price * (1 - discount);
        else
        return n * price;
    }
private:
    size_t min_qty = 0;
    double discount = 0.0;
};
double print_total(ostream &os, const Quote &item, size_t n) {
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
int main() {
                Quote basic("abc", 5.5);
                Bulk_quote bulk("abc", 5.5, 10, 0.2);
                print_total(cout, basic, 20);
                print_total(cout, bulk, 20);
                return 0;
```

Exercise 15.4:

- **Which of the following declarations, if any, are incorrect? Explain why.**

```
class Base { ... };
```

(a) `class Derived : public Derived { ... };`

(b) `class Derived : private Base { ... };`

(c) `class Derived : public Base;`

- **Answer:**

(a) Incorrect, it is impossible to derive a class from itself.

(b) Correct, it is not only a declaration, but also a definition.

(c) Incorrect. The declaration of a derived class contains the class name but does not include its derivation list.

Exercise 15.(5 & 6):

1. **Define your own version of the `Bulk_quote` class.**
   - **Answer: See ex15.3.**
2. **Test your `print_total` function from the exercises in § 15.2.1 (p. 595) by passing both `Quote` and `Bulk_quote` objects o that function.**
   - **Answer: See ex15.3.**

Exercise 15.7:

- Define a class that implements a limited discount strategy, which applies a discount to books purchased up to a given limit. If the number of copies exceeds that limit, the normal price applies to those purchased beyond the limit.

Answer 15.7

```
#include <string>
#include <iostream>
class Quote {
public:
      Quote() = default;
      Quote(const std::string &book, double pri): bookNo(book), price(pri) { }
      virtual ~Quote() = default;
      std::string isbn() const { return bookNo; }
      virtual double net_price(std::size_t n) const { return n * price; }
protected:
      double price = 0.0;
private:
      std::string bookNo;
};
class Bulk_quote : public Quote {
      public:
      Bulk_quote() = default;
      Bulk_quote(const std::string &book, double pri, std::size_t qty, double disc): Quote(book, pri),
      min_qty(qty), discount(disc){}
      double net_price(std::size_t n) const override {
            if (n >= min_qty)
                  return n * price * (1 - discount);
            else
                  return n * price;
      }
      private:
                  std::size_t min_qty = 0;
                  double discount = 0.0;
};
```

Exercise 15.8:

- Define static type and dynamic type.
- **Answer:**
  - The static type is the type with which a variable is declared or that an expression yields. It is always known at compile time.
  - The dynamic type is the type of the object in memory that the variable or expression represents.It is not known until run time.The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression's static type.

uOttawa

Exercise 15.9:

- When is it possible for an expression’s static type to differ from its dynamic type? Give three examples in which the static and dynamic type differ.
- **Answer:**

The dynamic type of an expression that is either a reference or a pointer could be different from that expression's static type.

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };
Derived d;
Base *bp = &d;
Base &br = d;
Derived *dp = &d;
Base *bp2 = dp;
```

bp2, bp and br have different static type and dynamic type.

Exercise 15.10:

- Recalling the discussion from §8.1 (p. 311), explain how the program on page 317 that passed an ifstream to the Sales_data read function works.
- Answer:
  - ifstream is inherited from istream. Thus, we can use objects of type ifstream as if they were istream objects.
  - The parameter of read function is a reference to base type istream, the automatic derived - to - base conversion applies when we pass a derived type ifstream to that function. Inside read function, we can use the istream subpart in the ifstream object through the reference.

Exercise 15.12:

- Is it ever useful to declare a member function as both override and final? Why or why not?
- Answer:
  - Sure, override means overriding the same name virtual function in base class.final means preventing any overriding this virtual function by any derived classes that are more lower at the hierarchy.

Refereces

Accreditation:

- This presentation is prepared/extracted from the following resources:
  - C++ Primer, Fifth Edition.
    Stanley B. Lippman Josée Lajoie Barbara E. Moo
  - https://github.com/jaege/Cpp-Primer-5th-Exercises
  - https://github.com/Moophy/Cpp-Primer