Part C - Encapsulation

Classes and Resources

Design classes with dynamically allocated resources to model the components of a programming solution
Define the copy constructor and assignment operator for a class with a resource

"Never allocate more than one resource in a single statement" (Sutter, Alexandrescu, 2005)

In object-oriented programming, we design classes to behave independently of their client applications. Wherever client code dictates the amount of memory that an object requires, the memory that needs to be allocated is unknown at compile-time. Only once the client has instantiated the object will the object know how much memory the client requires. To review run-time memory allocation and deallocation see the chapter entitled Dynamic Memory.

Memory that an object allocates at run-time represents a resource to its class. Management of this resource requires additional logic that was unnecessary in simpler designs. This additional logic ensures proper handling of the resource and is often called deep copying and deep assignment.

This chapter describes how to implement deep copying and deep assignment logic. The member functions that manage resources are the constructors, the assignment operator and the destructor.

RESOURCE INSTANCE POINTERS

A C++ object refers to a resource through a *resource instance pointer*. This pointer holds the address of the resource. The address lies outside the object's static memory.

Case Study

Let us upgrade our **`Student`** class to accommodate a variable number of grades. The client code specifies the number at run-time. The array of grades is now a dynamically allocated resource. We allocate

- static memory for the resource instance variable (**`grade`**)
- dynamic memory for the **`grade`** array itself

In this section, we focus on the constructors and the destructor for our **`Student`** class. Let us assume that the client does not copy or assign objects of this class. We shall cover the copying and assignment logic in subsequent sections:

```
// Resources - Constructor and Destructor
// resources.cpp

#include <iostream>
using namespace std;

class Student {
    int no;
    float* grade;
    int ng;
public:
    Student();
    Student(int);
    Student(int, const float*, int);
    ~Student();
```

```
    void display() const;
};

Student::Student() {
    no = 0;
    ng = 0;
    grade = nullptr;
}

Student::Student(int sn) {
    float g[] = {0.0f};
    grade = nullptr;
    *this = Student(sn, g, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_;
        // allocate dynamic memory
        if (ng > 0) {
            grade = new float[ng];
            for (int i = 0; i < ng; i++)
                grade[i] = g[i];
        } else {
            grade = nullptr;
        }
    } else {
        grade = nullptr;
        *this = Student();
    }
}

Student::~Student() {
    delete [] grade;
}

void Student::display() const {
    if (no > 0) {
        cout << no << ":\n";
        cout.setf(ios::fixed);
        cout.precision(2);
        for (int i = 0; i < ng; i++) {
            cout.width(6);
            cout << grade[i] << endl;
        }
        cout.unsetf(ios::fixed);
        cout.precision(6);
    } else {
        cout << "no data available" << endl;
    }
}

int main () {
    float gh[] = {89.4f, 67.8f, 45.5f};
    Student harry(1234, gh, 3);
    harry.display();
}
```

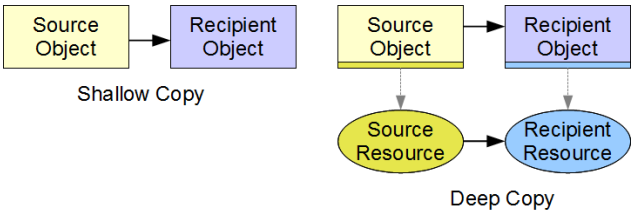
```
1234:
 89.40
 67.80
 45.50
```

The no-argument constructor places the object in a safe empty state. The three-argument constructor allocates dynamic memory for the resource only if the data received is valid. The pre-initialization of **`grade`** is a precaution that ensures no inadvertent destruction of memory (see the assignment operator section below). The destructor deallocates any memory that the constructor allocated. Deallocating memory at the **`nullptr`** address has no effect.

DEEP COPIES AND ASSIGNMENTS

In designing a class with a resource, we expect the resource associated with one object to be independent of the resource associated with another object. That is, if we change the resource data in one object, we expect the resource data in the other object to remain unchanged. In copying and assigning objects we ensure resource independence through deep copying and deep assigning. Deep copying and deep assigning involve copying the resource data. Shallow copying and assigning involve copying the instance variables only and are only appropriate for non-resource instance variables.

Implementing deep copying and assigning requires dynamic allocation and deallocation of memory. The copying process includes not only the non-resource instance variables but also the resource data itself.



In each deep copy, we allocate memory for the underlying resource and copy the contents of the source resource into the destination memory. We shallow copy the instance variables that are NOT resource instance variables. For example, in our **`Student`** class, we shallow copy the student number and number of grades, but not the address stored in the **`grade`** pointer.

Two special member functions manage allocations and deallocations associated with deep copying and deep copy assigning:

- the copy constructor
- the copy assignment operator

If we do not declare a copy constructor, the compiler inserts code that implements a shallow copy. If we do not declare a copy assignment operator, the compiler inserts code that implements a shallow assignment.

COPY CONSTRUCTOR

The copy constructor contains the logic for copying from a source object to a *newly created* object of the same type. The compiler calls this constructor when the client code

1. creates an object by initializing it to an existing object
2. copies an object by value in a function call
3. returns an object by value from a function

Declaration

The declaration of a copy constructor takes the form

```
Type(const Type&);
```

where **`Type`** is the name of the class.

To define a copy constructor, we insert its declaration into the class. For example, we insert the following into the definition of our **`Student`** class:

```
// Student.h

class Student {
    int no;
    float* grade;
    int ng;
public:
    Student();
    Student(int, const char*);
    Student(const Student&);
    ~Student();
    void display() const;
};
```

Definition

The definition of a copy constructor contains logic to

1. perform a shallow copy on all non-resource instance variables

2. allocate memory for each new resource
3. copy data from the source resource to the newly created resource

For example, the following code implements a deep copy on objects of our **`Student`** class:

```
// Student.cpp

#include <iostream>
using namespace std;
#include "Student.h"

// ...

Student::Student(const Student& src) {

    // shallow copies
    no = src.no;
    ng = src.ng;

    // allocate dynamic memory for grades
    if (src.grade != nullptr) {
        grade = new float[ng];
        // copy data from the source resource
        // to the newly allocated resource
        for (int i = 0; i < ng; i++)
            grade[i] = src.grade[i];
    }
    else {
        grade = nullptr;
    }
}
```

Since the source data was validated on its original receipt from the client code and privacy constraints have ensured that this data has not been corrupted in the interim, we do not need to revalidate the data in the copy constructor logic.

COPY ASSIGNMENT OPERATOR

The copy assignment operator contains the logic for copying data from an existing object to an *existing* object. The compiler calls this member operator whenever for client code expressions of the form

```
identifier = identifier
```

`identifier` refers to the name of an object.

Declaration

The declaration of an assignment operator takes the form

```
Type& operator=(const Type&);
```

the left ***`Type`*** is the return type and the right ***`Type`*** is the type of the source operand.

To define the copy assignment operator, we insert its declaration into the class definition. For example, we insert the following declaration into the definition of our **`Student`** class:

```
// Student.h

class Student {
    int no;
    float* grade;
    int ng;
public:
    Student();
    Student(int, const float*, int);
    Student(const Student&);
    Student& operator=(const Student&);
    ~Student();
    void display() const;
};
```

Definition

The definition of the copy assignment operator contains logic to:

1. check for self-assignment
2. shallow copy the non-resource instance variables to destination variables
3. deallocate any previously allocated memory for the resource associated with the current object
4. allocate a new memory for the resource associated with the current object
5. copy resource data from the source object to the newly allocated memory of the current object

For example, the following code performs a deep copy assignment on objects of our `Student` class:

```
// Student.cpp

// ...

Student& Student::operator=(const Student& source) {
    // check for self-assignment
    if (this != &source) {
        // shallow copy non-resource variables
        no = source.no;
        ng = source.ng;
        // deallocate previously allocated dynamic memory
        delete [] grade;
        // allocate new dynamic memory, if needed
        if (source.grade != nullptr) {
            grade = new float[ng];
            // copy the resource data
            for (int i = 0; i < ng; i++)
                grade[i] = source.grade[i];
        }
        else {
            grade = nullptr;
        }
    }
    return *this;
}
```

To trap a self-assignment from the client code (`a = a`), we compare the address of the current object to the address of the source object. If the addresses match, we skip the assignment logic altogether. If we neglect to check for self-assignment, the deallocation statement would release the memory holding the resource data and we would lose access to the source resource resulting in our logic failing at `grade[i] = source.grade[i]`.

LOCALIZATION

The code in our definition of the copy constructor is identical to most of the code in our definition of the assignment operator. To avoid such duplication and thereby improve maintainability we can localize the logic in a:

- private member function - localize the common code in a private member function and call that member function from both the copy constructor and the copy assignment operator
- direct call - call the assignment operator directly from the copy constructor

Private Member Function

The following solution localizes the common code in a private member function named `init()` and calls this function from the copy constructor and the copy assignment operator:

```
void Student::init(const Student& source) {
    no = source.no;
    ng = source.ng;
    if (source.grade != nullptr) {
        grade = new float[ng];
        for (int i = 0; i < ng; i++)
            grade[i] = source.grade[i];
    }
    else {
        grade = nullptr;
```

```
    }
}

Student::Student(const Student& source) {
    init(source);
}

Student& Student::operator=(const Student& source) {
    if (this != &source) {  // check for self-assignment
        // deallocate previously allocated dynamic memory
        delete [] grade;
        init(source);
    }
    return *this;
}
```

Direct Call

The following solution initializes the resource instance variable in the copy constructor to **`nullptr`** and calls the copy assignment operator directly:

```
Student::Student(const Student& source) {
    grade = nullptr;
    *this = source; // calls assignment operator
}

Student& Student::operator=(const Student& source) {
    if (this != &source) {  // check for self-assignment
        no = source.no;
        ng = source.ng;
        // deallocate previously allocated dynamic memory
        delete [] grade;
        // allocate new dynamic memory
        if (source.grade != nullptr) {
            grade = new float[ng];
            // copy resource data
            for (int = 0; i < ng; i++)
                grade[i] = source.grade[i];
        }
        else {
            grade = nullptr;
        }
    }
    return *this;
}
```

Assigning **`grade`** to **`nullptr`** in the copy constructor ensures that the assignment operator does not deallocate any memory if called by the copy constructor.

Assigning Temporary Objects

Assigning a temporary object to the current object requires additional code if the object manages resources. To prevent the assignment operator from releasing not-as-yet-acquired resources we initialize each resource instance variable to an empty value (**`nullptr`**).

For example, in the constructors for our **`Student`** object, we add the highlighted code:

```
class Student {
    int no;
    float* grade;
    int ng;
public:
    // ...
};

Student::Student() {
   no = 0;
   ng = 0;
   grade = nullptr;
}
```

```
Student::Student(int n) {
   float g[] = {0.0f};
   grade = nullptr;
   *this = Student(n, g, 0);
}

Student::Student(int sn, const float* g, int ng_) {
    bool valid = sn > 0 && g != nullptr && ng_ >= 0;
    if (valid)
        for (int i = 0; i < ng_ && valid; i++)
            valid = g[i] >= 0.0f && g[i] <= 100.0f;

    if (valid) {
        // accept the client's data
        no = sn;
        ng = ng_;
        // allocate dynamic memory
        if (ng > 0) {
            grade = new float[ng];
            for (int i = 0; i < ng; i++)
                grade[i] = g[i];
        } else {
            grade = nullptr;
        }
    } else {
        grade = nullptr;
        *this = Student();
    }
}
```

COPIES PROHIBITED

Certain class designs require prohibiting client code from copying or copy assigning any instance of a class. To prohibit copying and/or copy assigning, we declare the copy constructor and/or the copy assignment operator as deleted members of our class:

```
class Student {
    int no;
    float* grade;
    int ng;
public:
    Student();
    Student(int, const float*, int);
    Student(const Student& source) = delete;
    Student& operator=(const Student& source) = delete;
    ~Student();
    void display() const;
};
```

The keyword **`delete`** used in this context has no relation to deallocating dynamic memory.

SUMMARY

- a class with resources requires custom definitions of a copy constructor, copy assignment operator and destructor
- the copy constructor copies data from an existing object to a newly created object
- the copy assignment operator copies data from an existing object to an existing object
- initialization, pass by value, and return by value client code invokes the copy constructor
- the copy constructor and copy assignment operator should shallow copy only the non-resource instance variables
- the copy assignment operator should check for self-assignment

EXERCISES

- Complete the Handout on Class with a Resource
- Complete the Workshop on Class with a Resource


CMS