10 - Encapsulation

February 8, 2023

COMP2404

Darryl Hill

Overview

1. Composition
2. Constants
3. Friendship
4. Static class members

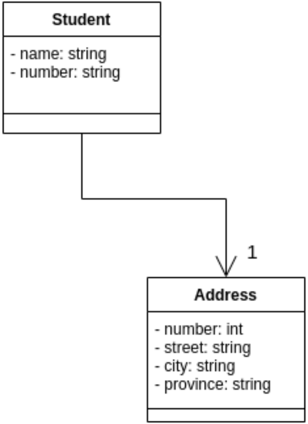
Composition

Composition is when an object contains a *member variable* that is an instance of another class

- as opposed to a primitive
- this is a "has-a" relationship
- containee object can be statically or dynamically allocated

On initialization:

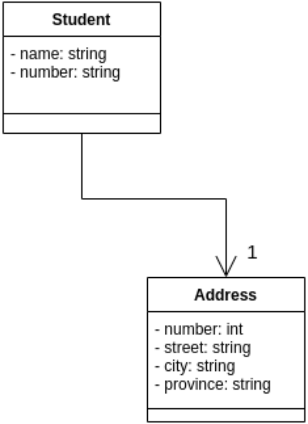
- container initializes containee (calls constructor either implicitly or explicitly)
- ***member initializer syntax*** allows us to call member variable constructor, even statically allocated ones
    - avoids creating temporary objects



Composition - Example

Imagine a `Student` class that requires an `Address`:

- We can store address attributes individually, but better to (re)use an `Address` class.
- The `Student` constructor should include the `Address` parameters.
  - These parameters are supplied to `Address`
- `Student::print()` can make use of `Address::print()`.
- This is proper encapsulation
  - the `Student` class lets `Address` do its thing.
  - how `Address` is initialized is up to the `Address`
  - how `Address` prints is up to the `Address`



Composition - Example

How to initialize `Address`:

- Bad way: try and assign values directly, doesn't work since these are private
- Bad way 2: make a temp variable using constructor, then copy in values using default assignment
  - wasteful

Good way - member initializer syntax

- `Address` constructor is called first, so how can we supply it arguments?

Coding example <p1>

Member Initializer Syntax

Member initializer syntax is the most effective way to initialize member variables

After the constructor, put a colon

- (we can initialize primitives and pointers as well, using their constructors)
- even primitives use (), ie, num(num) where both num's are ints
- pointers also use () and can be passed NULL or another pointer

Often in C++ constructors are empty, since everything is in member initializer syntax

```
Student::Student(string name, float gpa,
      int n, string s, string c, string p)

     : name(name), gpa(gpa), homeAddr(n,s,c,p)  { }
```

Composition

Constructors

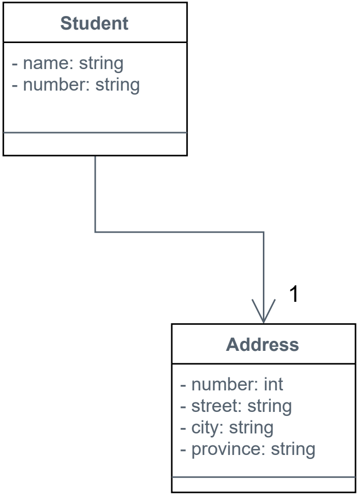
- objects are built from the inside out
- containee objects are first
  - in the order that they are declared
  - not in member initializer order
- every object is initialized (unlike Java)

Destructors

- objects are destroyed from outside in
- container is destroyed first, then containees
- in reverse order of constructors

Composition - Deep Copy

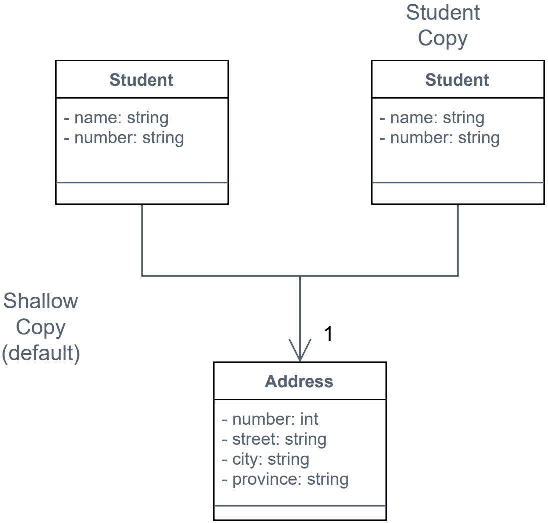
If we have one object with a ***pointer*** to another, we probably want to use ***deep copy*** in the copy constructor



Composition - Deep Copy

If we have one object with a ***pointer*** to another, we probably want to use ***deep copy*** in the copy constructor

Using a shallow copy (default) only copies the `Address` pointer. The two `Student` objects now share an `Address`.



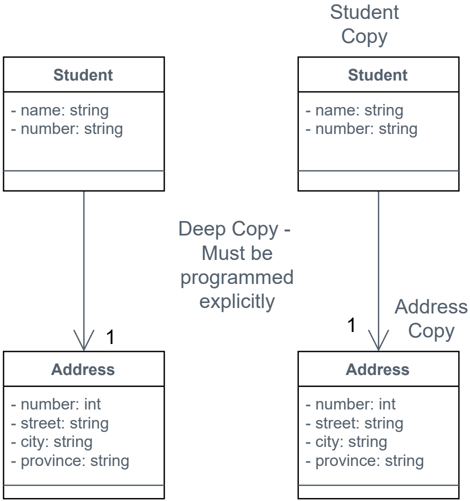
Composition - Deep Copy

If we have one object with a ***pointer*** to another, we probably want to use ***deep copy*** in the copy constructor

Using a shallow copy (default) only copies the `Address` pointer. The two `Student` objects now share an `Address`.

Deep copy will make a new *dynamically allocated* `Address` that is a copy of the old *dynamically allocated* `Address`

Coding example <p2>



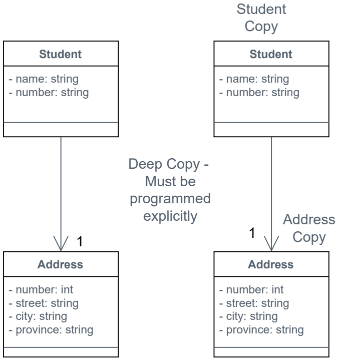
Composition - Deep Copy

This is also the reason behind the rule of three (five)

1) `Address` was dynamically allocated, thus `Student` required a ***destructor***.

2) Also because `Address` was dynamically allocated `Student` required a ***copy constructor***.

3) The ***assignment operator*** runs into this same problem - by default does a shallow copy. `Student` would likely also need an assignment operator.

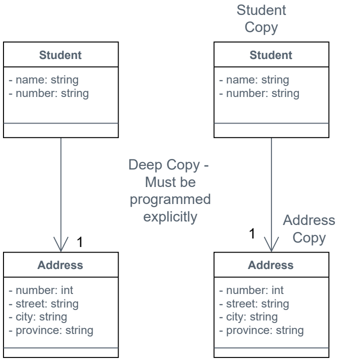


Composition - Deep Copy

This is also the reason behind the rule of three (five)

3) In the section on operator overloading we will see the ***assignment operator***.
   - This also copies one object to another.
   - If we need a destructor and / or a copy constructor, then we also need an ***assignment operator***.

4,5) There are also ***move*** constructors and assignment operators that we will not take in this class.
   - They are the opposite of copying.
   - We want to move an object by stealing its resources.



Constants

A constant is not the same as a ***literal***

We use the `const` qualifier

- purpose is protection
- If there is no reason for an object or variable to be modified we make it `const`
    - compiler will then detect these errors.

`const` has many uses:

- constant objects
- constant member functions
- constant data members

Constants

Compiler error messages will refer to `lvalues` and `rvalues`, so it is useful to know these terms.

- An `lvalue` is a value that can be the LHS of assignment
  - `lvalue = rvalue;`
- An `lvalue` can also be RHS of assignment
- An `rvalue` can **only** be the RHS of assignment
- `(a + b)` is an `rvalue` - cannot write `(a + b) = 5;`
- Any primitive or object returned by value is an `rvalue`.

```
Student getStudent();
```

- The returned `Student` is an `rvalue`.

Constant Objects

Constant variables (either local variables or data members) **cannot** be changed.

A `const` data member is an `rvalue`

- A `const` variable ***must*** be assigned a value upon declaration

Constant object:

- is an object
- once initialized, the ***data members*** of the object cannot change
- no part of the program may change the object, **including** the object itself

Side effect:

- Only ***constant member functions*** can be called on a constant object

coding example <p3>

Constant Objects

Principle of least privilege:

If the objects you create don't need to be modified, make them constant.

- Guarantees the integrity of your objects
  - We can guard against users of your code corrupting values
- Helps the compiler catch errors
- Sometimes difficult to foresee if objects or functions should be made constant.
  - In some cases it is easy

Constant Member Function

Putting a `const` modifier in front of the member function:

- Guarantees function will not modify the object
- We know print() does not modify anything, thus:

  ```
  void print() const;
  ```
- `print()` calls `getMonthStr`
  - Now we have to also change `getMonthStr` to `const`
- **constant member functions may only call other constant member functions**

Again, this is to help the compiler catch mistakes.

- Technically we can make nothing `const` and "be really careful"
- Easier to make the compiler do the work for us.

Constant Member Function

Constant member functions can be called by non-constant objects

- We can have two versions of a member function, a `const` and a non-`const`
- Notice compiler complains when we try to change a member variable in a `const` function

coding example <p4>

Constant Member Function

Principle of least privilege:

- If you don't need to change the object in a member function, make it constant
  - This ensures it can be called no matter what
  - by a `const` object, or from another `const` function
- All getters and `print` functions can be `const`
- All simple `bool` functions can be `const`
  - functions that simply test some condition

```
bool lessThan(Object&) const
```

Constant Data Members

A **constant data member** is a data member that can never be modified

- not even in the constructor!

Must be initialized before the body of the constructor using member initializer syntax.

coding example <p5>

Constant Data Members

Applying the principle of least privilege:

- consider the data members you create
- if they will never be modified, make them constant
- this guarantees their integrity
  - even from member functions

CAUTION! A constant member variable will cause the default assignment operator for that class to be deleted!

```
Student amy("1", "Amy"), bob("2", "Bob");
bob = amy; // this will cause an error
```

Constants cannot be re-assigned a value, so the compiler says " ¯_(ツ)_/¯ you figure it out"

Member Initializer Syntax

Member Initializer Syntax, Review:

- Used between the constructor parameter list and the body
- May be used to initialize non-constant data members
- ***Must*** be used to initialize constant data members
- Executes before the body of the constructor
- Allows us to pass parameters to member variable constructors
  - The ***only way*** to pass values to statically allocated member variable objects
- Can result in an empty constructor body (which is fine)

Constant Parameters

One of the rules of Software Engineering is:

> "Pass by reference or pointer, not by value"

This can cause trouble with `string` arguments

- We often pass in string literals as arguments

  ```
  Student stu("Bob","1111");
  ```

But string literals are stored in the data segment and cannot be modified.

If our constructor looks like this:

```
Student(string&, string&);
```

This will cause an error.

We can solve this problem by making the arguments
`const string&`

- **coding example <p6>**

Friendship

Recall we said objects should be introverts and keep private things private.

- Sometimes objects will share private things with a **Friend**.

Friendship:

- Violates principles of OO programming and data encapsulation.
- Welcome to C++ where we break every rule.
  - There are sometimes good reasons for this.

A class may grant friendship to

- a global function
  - (not a member function)
- another class

Friendship

Friendship gives away complete access to the class members

- even `private` and `protected` members
- Friendship can only be **given**, not taken
- **not** symmetric, **not** transitive
  - (A friend to B) and (B friend to C) $\not\Rightarrow$ (A friend to C)
  - (A friend to B) $\not\Rightarrow$ (B friend to A)
  - Each friendship must be explicitly granted.

Should only be used in very specific situations.

- Nested classes that work closely together (`Student` and `Address` for example)
- Some overloaded operators.

Friendship

Friend function (of a Class)

- global function given complete access to Class
- can access all members (public, private, protected)

Friend Class (of a Class)

- another Class that is given complete access to the Class
- Again, can access all members

```
class Address{
     friend Class Student;
};
```

coding example <p7>

Friendship

We want `Students` to be able to change their `Address` (but not other classes)

- A `setter` would be public, can be used by any class.
- Could make a new `Address` and destroy the old, or
- change the `Address` using friendship
  - More efficient, however...
  - If I change `Address` implementation it might break `Student` code
  - Entangling objects makes updating more difficult.

One other problem (not exclusive to Friendship):

- In this Friendship example both classes require some knowledge of the other.
- At least one class must be forward referenced.
- What would happen otherwise? Think like a compiler.

Static Class Members

Exactly one copy of a static member exists

- Irrespective of the number of instances of the class there are.
- Exist even if no objects of the class are created

Can be accessed:

- using class name and binary scope resolution operator
- from any object of that class

Static Class Members

Static data member

- property of the **class** as a whole
  - value shared by all instances
- must be initialized at file scope
  - in source file by convention
- Do **NOT** include the `static` keyword when initializing a static variable in the source file!
  - Only include it with the class definition
  - `static` has a different meaning there
  - Limits global variable to the file it is declared in

Static Class Members

Static member function

- Service of the class as a whole
- Can only access static members
- Though can still take objects of this type as arguments
- Specified as static in the class definition
  - not in source file, since it wouldn't be visible

coding example <p8> - updating the id variable