Advanced Programming Concepts with C++

Classes and Objects

Fall 2021
CSI2372[A]

Amir EAMAN
PostDoc Researcher
Faculty of Engineering, uOttawa

Classes

- Classes are the blueprints for building objects.
- Classes and Structs are used for defining new data-types (user-defined data-types)
- Classes have members (attributes and methods)

- Objects are instances of classes and they are created from classes.
- Each object has its own copies of class members.

Example)

```
int i (87);
std::string name; // in C++, std::string is a class!
Student st1;

int & ref_toInt = i;
Student & ref_to_Student = st1;
```

Classes

Declaring a Class named Student

```
class Student
{
    int studentID;
    std::string name;
public:
    Student() { studentID = 0; name = ""; } //default constructor - no args constructor
    Student(int id, std::string nameIniti) { studentID = id; name = nameIniti; } // construcotr with two parameters
};

int main()
{
    Student st1;
    Student st2(1, "zina");

}
```

Member Functions VS. Ordinary Functions

```
class Student
{
    int studentID;
    std::string name;
public:
    Student() { studentID = 0; name = ""; course = 2;  } //default constructor - no args constructor
    Student(int id, std::string nameIniti) { this-> studentID = id; (*this).name = nameIniti; course = 2;  } // construcotr with two parameters
    int course;
};

void changeID (Student st, int newID)    { st.course = newID;}
void changeID_byReference (Student& st, int newID)  {st.course = newID; }

int main()
{
    Student st1;
    Student st2(1, "zina");
    changeID (st1, 43);  ≤ 2ms elapsed
    changeID_byReference(st2, 55);
    std::cout << st1.course << " " << st2.course << std::endl; //prints 2 55
}
```

- By default, objects as arguments to method are passed *by value*, so we should change function signature as intended.
- Member functions defined inside the class are automatically inline by default.

this pointer

Member functions access the object on which they were called through a hidden argument this which is accessible within the body of member functions. this is a *const* pointer.

Example) re-write the Student default constructor using this pointer:

```
Student(int id, std::string nameIniti) { this-> studentID = id; (*this).name = nameIniti; course = 2;  }
```

Example) Dereferencing this to obtain the object on which the member function is executing

```
return *this; //   return the object on which the function was called
```

Constructors

- Constructors are automatically called when an object of the class is created.
- No return type
- They are used for initializing and allocating memory and and resources.

```
//overloaded constructors
ClassName(); // default constructor with no args
ClassName(int var);  //constructor which takes one paramater
ClassName(int var, std::string name);
```

Destructors

- Destructors are automatically called when an object is destroyed.
- No return type and no input parameters
- They are used for releasing allocated memory and resources to the objects which are about to expire and therefore they must be destroyed.
- Destructors defines what happens when an object of the type expires.
- Destructors cannot be overloaded.

Example)

```
class ClassName {
    Public :
        ~ClassName();  // destructor
        //…
};
```

Special Member Functions of Classes

- We control what happens when objects are *copied*, *moved*, *assigned*, or *destroyed* through these special member functions when we define C++ classes:
- The Special constructors define what happens when an object is *initialized* from another object of the same type, and special assignment operators define what happens when we *assign* an object of a class type to another object of that same class type.

Special Member Function name	syntax for the class *MyClass*
Default constructor	MyClass();
Copy constructor	MyClass (const MyClass& obj_usedfor_initialization);
Copy-assignment operator	MyClass &operator=(const MyClass& obj_rightHand);
Destructor	~ MyClass();
Move constructor	MyClass (MyClass&& other);
Move-assignment operator	MyClass& operator=(MyClass&& other);

Copy Constructor

- The *default copy constructor* (which compiler provides!) is a **memberwise copying** of members of its argument into the object being created. But we can create our own copy constructor.
- The copy constructor first parameter is a reference to the class type

```
class Foo {
public:
    Foo();              // default constructor
    Foo(const Foo&);    // copy constructor
    // ...
};
```

- When is copy constructor used (an object must be copied):
  - Define one object using another object from the same class and using the assignment = (i.e., object initialization when creating objects)
  - Pass an object as an argument to a parameter of nonreference type (pass objects by value)
  - Returning an object from a function by value that has a nonreference type

Passing an Object by value needs copying

copy

destructor for st is called here

```cpp
void changeID (Student st, int newID)    { st.course = newID;}
void changeID_byReference (Student& st, int newID)  {st.course = newID; }

int main()
{

    Student st1;
    Student st2(1, "zina");
    changeID (st1, 43);
    changeID_byReference(st2, 55);
```

st is a copy of st1

st scope ends here

so it must be destructed

Initializing one object using another object

```
Student st1;
Student st2(1, "zina");

Student st3 { st2 };// copy constructor is called becuase a copy of st2 is makde
```

st3 {studentID=1 name="zina" course=2 }

copy

Because we didn't provide our own copy constructor, compiler generates one copy constructor for us, which has copied all the member of st2 to st3.

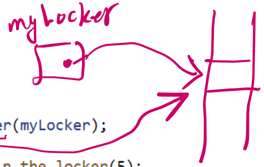
Shallow copying VS. Deep copying

The *default copy constructor,* which compiler generates, performs a ***memberwise copying*** on the objects (this kind of copying is called **Shallow Copying**)**.** This causes problem when the class of the objects has a *pointer* attribute. When we destroy one of the objects, the pointer attribute of the other object points to an invalid memory location. **SOLUTION? Deep Copying**

Shallow copying VS. Deep copying

Shallow copy

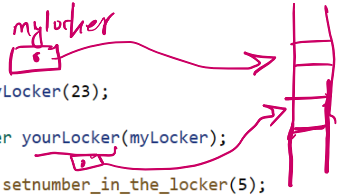
```
class Locker
{
  private:
   int *ptr_to_locker;
  public:
      Locker(int value) : ptr_to_locker (new int(value)) {}

      Locker(const Locker& object_used_for_initialization) //copy cstr with "Shallow Copy"
      {
         ptr_to_locker = object_used_for_initialization.ptr_to_locker;
      }
       void setnumber_in_the_locker(int i) { *ptr_to_locker = i; }
      ~Locker() { delete ptr_to_locker; ptr_to_locker = nullptr; }
};

int main()
{
   Locker myLocker(23);
   {
      Locker yourLocker(myLocker);
   }
   myLocker.setnumber_in_the_locker(5);
   return 0;
}
```



Deep Copy

```
class Locker
{
  private:
   int *ptr_to_locker;
  public:
      Locker(int value) : ptr_to_locker (new int(value)) {}

      Locker(const Locker& object_used_for_initialization) //copy cstr with "Deep Copy"
      {
         int num = *(object_used_for_initialization.ptr_to_locker);
         ptr_to_locker = new int(num);
      }
       void setnumber_in_the_locker(int i) { *ptr_to_locker = i; }
      ~Locker() { delete ptr_to_locker; ptr_to_locker = nullptr; }
};

int main()
{
   Locker myLocker(23);
   {
      Locker yourLocker(myLocker);
   }
   myLocker.setnumber_in_the_locker(5);
   return 0;
}
```

pointer



Copy-Assignment Operator Constructor

- The *default copy-Assignment constructor* (which compiler provides!) is a **memberwise assignment** which is a shallow copy.
- The ***copy-assignment operator constructor*** controls how objects of its class are assigned.
- The copy-assignment operator constructors return a reference to their left-hand operand.
- Actually, we are overloading the Assignment Operator (=).

```
class Foo {
public:
    Foo& operator=(const Foo&); // assignment operator
   // ...
};
```

- When is copy-assignment constructor used:
  - The *left-hand side (lhs)* object in the assignment (lhs=rhs;) is already defined and exists (not for initialization)

```
Student st1;
Student st2(1, "zina");
Student st3 = st2 ;// copy constructor (not copy assignment)
st3 = st1;//copy assignment (assignment)
```

rhs is short for righ-hand side
lhs is short for left-hand side

Copy-Assignment Operator Constructor

- Example) Defining a **copy assignment Operator** Constructor for the Locker class

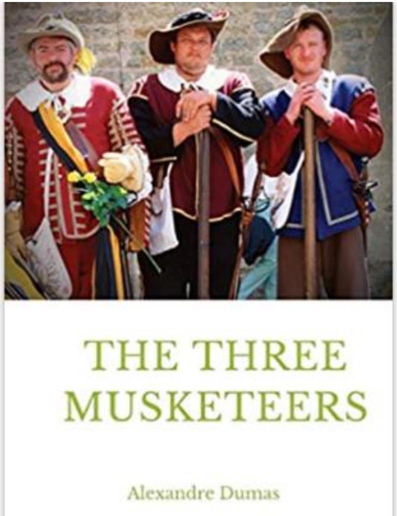
```
Locker& operator= (const Locker& rhs) { //copy assignment operator with "Deep Copy"
    std::cout << "Copy assignment with Deep Copy" << std::endl;
    if (this == &rhs) { return *this; }

    delete ptr_to_locker;
    int num = *(rhs.ptr_to_locker);
    ptr_to_locker = new int(num);
    return *this;

}
```

We need all! The rule of Three

Rule of thumb in C++: (The Big Three)

The Three Musketeers:

1. *Copy Constructor*
2. *Copy-assignment operator*
3. *destructor*



R-value reference &&

L-values have names and they are allocated on memory (they have memory address).
R-Values don't have names and they don't have memory address.

We can use L-values either on the *rhs* or *lhs* of an assignment (=), but R-values can just used on the *rhs* of the assignment because they cannot store any values.

```
int j{ 4 }; //j is an L-value
int var{ 98 }; //var is an L-value
int & lvalue_ref = var;
int & ref_to_Lvalue = 78798; //Error 78798 is a R-value
int && ref_to_Rvalue = 78798;
std::string&& ref_to_another_Rvalue = "C++"; // "C++" is a string literal
int& refLvalue = j + var; //Error j + var is R-value
int&& refRvalue = j + var; //Correct j + var is R-value
```