Object Oriented Programming

Rupesh Nasre.
rupesh@iitm.ac.in

QEEE
January 2016

Class and Object

- Class: Type
- Object: Variable / instance
  - e.g., Car tn07bw156; Student s;
  - Car, Student are classes; tn07bw156, s are objects of those types.
- Each object has all the properties defined for its class.
  - It has all the member fields.
  - It has all the member functions.

```
class Student {
  String rollno, name;
  String hostel;
  int roomno;
  Teacher facad;

  void changeHostel(
            String& newhostel);
  Teacher& getFacad();
};
```

Encapsulation

- Putting data and associated functions together is called *encapsulation*.
- Encapsulation improves programmer productivity, software design as well as software efficiency.
- **Homework:** How does one achieve encapsulation in C?
- **Homework:** What is the difference between encapsulation and data-hiding?

Data members or fields

Member functions or methods

```
class Student {
  String rollno, name;
  String hostel;
  int roomno;
  Teacher facad;

  void changeHostel(
            String& newhostel);
  Teacher& getFacad();
};
```

Class Instantiation

main.cpp

```
#include <iostream>
#include “Student”

int main() {
      Student s1, s2;
      s1.rollno = “CS16D001”;
      std::cout << s2.getFacad().getName();
      return 0;
}
```

Student

```
#include “Teacher”

class Student {
  String rollno, name;
  String hostel;
  int roomno;
  Teacher facad;

  void changeHostel(
            String& newhostel);
  Teacher& getFacad();
};
```

```
$  g++ main.cpp
Error: rollno is not public
Error: getFacad is not public
```

Access Permissions

- Unlike struct / union in C, C++ classes have access permissions
  - public, private, protected
  - We can say that all struct fields are public.
- Language enforces access checks.
  - This is the reason for error on the last slide.
  - Such checks helps programmers avoid inadvertent or unintentional accesses.
  - They also improve the overall software design.
  - A compiler needs to do more work.

Access Permissions

- A class has two types of members: fields and methods.
- We divide the world into three parts:
  - class, immediate children (inheritance), rest of the world
  - Similar to owner, group and others in Linux file permissions.

	public	protected	private
class	✓	✓	✓
children	✓	✓	×
rest	✓	×	×

```
#include <iostream>
class AccessCheck {
public:
        int pubvar;
        void pubfun() { std::cout << "in pubfun.\n"; }
        void checkFromClass() {
                pubfun();          // okay
                profun();          // okay
                prifun();          // okay
                pubvar = 10;       // okay
                provar = 11;       // okay
                privar = 12;       // okay
        }
protected:
        int provar;
        void profun() { std::cout << "in profun.\n"; }
private:
        int privar;
        void prifun() { std::cout << "in prifun.\n"; }
};
```

access from class

```
int main() {
        AccessCheck ac;
        ac.pubvar = 4;
        ac.provar = 5;    // error
        ac.privar = 6;    // error
        ac.pubfun();
        ac.profun();      // error
        ac.prifun();      // error
        return 0;
}
```

access from world

	public	protected	private
class	✓	✓	✓
children	✓	✓	×
rest	✓	×	×

```
#include <iostream>
class AccessCheck {
public:
        int pubvar;
        void pubfun() { std::cout << "in pubfun.\n"; }
        void checkFromClass() {
                pubfun();          // okay
                profun();          // okay
                prifun();          // okay
                pubvar = 10;       // okay
                provar = 11;       // okay
                privar = 12;       // okay
        }
protected:
        int provar;
        void profun() { std::cout << "in profun.\n"; }
private:
        int privar;
        void prifun() { std::cout << "in prifun.\n"; }
};
```

access from class

```
class DerivedAccessCheck
        : public AccessCheck {
public:
        void checkFromChild() {
                pubfun();
                profun();
                prifun();      // error
                pubvar = 0;
                provar = 1;
                privar = 2;  // error
        }
};
```

access from child

	public	protected	private
class	✓	✓	✓
children	✓	✓	×
rest	✓	×	×

Interface and Implementation

- C++ allows us to separate interface from the implementation.
  - Similar to declaration and definition.
- This helps in shipping the interface with compiled implementation as a library.
  - User would not have access to C++ source of the implementation.
- Interface is often part of the header files, while implementation can be in .so or .a file, compiled from .cpp files.
  - e.g., <math.h> and libm.so

```
#include <iostream>
class AccessCheck {
public:
        int pubvar;
        void pubfun() { std::cout << "in pubfun.\n"; }
        void checkFromClass() {
                pubfun();          // okay
                profun();          // okay
                prifun();          // okay
                pubvar = 10;       // okay
                provar = 11;       // okay
                privar = 12;       // okay
        }
protected:
        int provar;
        void profun() { std::cout << "in profun.\n"; };
private:
        int privar;
        void prifun() { std::cout << "in prifun.\n"; };
};
```

Interface and implementation together.

Interface and implementation separate.

```
#include <iostream>
class AccessCheck {
public:
        int pubvar;
        void pubfun();
        void checkFromClass();
protected:
        int provar;
        void profun();
private:
        int privar;
        void prifun();
};
```

```
void AccessCheck::pubfun() {
        std::cout << "in pubfun.\n";
}
void AccessCheck::checkFromClass() {
                pubfun();       // okay
                profun();       // okay
                prifun();       // okay
                pubvar = 10;    // okay
                provar = 11;    // okay
                privar = 12;    // okay
}
void AccessCheck::profun() {
        std::cout << "in profun.\n";
}
void AccessCheck::prifun() {
        std::cout << "in prifun.\n";
}
```

- Class method call is also called a message.
- Client-Server model: Class functionality constitutes the server and its use defines the client.
- **Homework**: When can a class be a server as well as a client?

Let's make tea: Constructor

- Often, we need to initialize an object with certain parameters.
  - burner with pot
  - student with rollno
  - person with name
  - ii with 0
- Constructors help us achieve it.
  - Instead of

    Student s; s.init("CS16B001");

    use

    Student s("CS16B001");

```
...
int main() {
        Pot pot;
        Burner burner;
        Water water(1);
        Tealeaves tealeaves(1)
        Sugar sugar(1);
        Milk milk(0.5);
        burner.start(pot);
        pot.add(water);
        pot.add(tealeaves);
        burner.boil(2, false);
        pot.add(sugar);
        pot.add(milk);
        burner.boil(2, true);
        burner.stop();
        std::cout << "Tea is ready.\n";
        return 0;
}
```

Let's make tea: Constructor

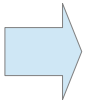
```
…
int main() {
        Pot pot;
        Water water(1);
        Tealeaves tealeaves(1)
        Sugar sugar(1);
        Milk milk(0.5);
        Burner burner(pot);
        pot.add(water);
        pot.add(tealeaves);
        burner.boil(2, false);
        pot.add(sugar);
        pot.add(milk);
        burner.boil(2, true);
        burner.stop();
        std::cout << “Tea is ready.\n”;
        return 0;
}
```

It would be good to avoid it.

```
…
int main() {
        Pot pot;
        Burner burner;
        Water water(1);
        Tealeaves tealeaves(1)
        Sugar sugar(1);
        Milk milk(0.5);
        burner.start(pot);
        pot.add(water);
        pot.add(tealeaves);
        burner.boil(2, false);
        pot.add(sugar);
        pot.add(milk);
        burner.boil(2, true);
        burner.stop();
        std::cout << “Tea is ready.\n”;
        return 0;
}
```

Let's make tea: Destructor

```
...
int main() {
    Pot pot;
    Water water(1);
    Tealeaves tealeaves(1)
    Sugar sugar(1);
    Milk milk(0.5);
    Burner burner(pot);
    pot.add(water);
    pot.add(tealeaves);
    burner.boil(2, false);
    pot.add(sugar);
    pot.add(milk);
    burner.boil(2, true);
    burner.stop();
    std::cout << “Tea is ready.\n”;
    return 0;
}
```



```
...
int main() {
    Pot pot;
    Water water(1);
    Tealeaves tealeaves(1)
    Sugar sugar(1);
    Milk milk(0.5);
    Burner burner(pot);
    pot.add(water);
    pot.add(tealeaves);
    burner.boil(2, false);
    pot.add(sugar);
    pot.add(milk);
    burner.boil(2, true);
    std::cout << “Tea is ready.\n”;
    return 0;
}
```

Constructor and Destructor

- A constructor is called (automatically) when an object is created / instantiated.
- A destructor is called (automatically) when an object goes out of scope / is destroyed.
- Constructor typically assigns initial values to fields and allocates resources.
- A destructor is helpful when some cleanup is required at the end of life of an object.
  - fopen – fclose
  - lock – unlock
  - malloc – free, new – delete

```
class ConstDest {
public:
        ConstDest(int n) {
                std::cout << "in ConstDest constructor.\n";
                ptr = new char(n);
        }
        ~ConstDest() {
                std::cout << "in ConstDest destructor.\n";
                delete ptr;
        }
private: char *ptr;
};
int main() {
        std::cout << "entering scope.\n";
        {
                std::cout << "entered scope.\n";
                ConstDest cd(10);
                std::cout << "leaving scope.\n";
        }
        std::cout << "left scope.\n";
        return 0;
}
```

```
$ g++ file.cpp
$ a.out
```

```
entering scope.
entered scope.
in ConstDest constructor.
leaving scope.
in ConstDest destructor.
left scope.
```

Why passing arguments to a destructor does not make sense?

Constructors

- If we do not define one, C++ provides a default (with zero arguments).
  - Student s; // okay: default constructor.
  - Student s("CS16D001"); // compilation error.
- If we define one, C++ doesn't provide the default.
  - Student s("CS16D001"); // okay: defined constructor.
  - Student s; // compilation error.
- We can define multiple constructors, with different arguments (polymorphism).
  - Student s("CS16D001"); // okay: defined.
  - Student s; // okay: defined.

How about multiple destructors?

Constructors

- What is the issue with the following code?

```
...
class ConstDefaultArgs {
public:
        ConstDefaultArgs(int n = 10) {
                std::cout << "first constructor.\n";
        }
         ConstDefaultArgs() {
                 std::cout << “second constructor\n”;
         }
};
```

- Ambiguous call to the constructor. Therefore, compilation error.

Class versus Object Variables

- Each object of a class has a different copy of its fields.
  - AAA a, b; a.xx and b.xx are different fields.
  - These are called object variables.
- If a field is defined as ***static***, it has a single copy across all instances (zero or more).
  - BBB a, b; a.xx and b.xx are the same.
  - These are called class variables.

```
class AAA {
public:
        void fun() {
                this->xx = 10;
        }
private:
        int xx;
};
```

```
class AAA {
public:
        void fun() {
                this->xx = 10;
        }
private:
        static int xx;
};
```

Class versus Object Variables

- Static variables exist even when no objects of the class exist.
- A static method can be invoked even when no objects of the class exist.
- A static method can be called as Classname::fun(...). It can as well be called using the object variable.
- A static method cannot use non-static variables (that is, cannot use object variables).
- But a non-static method can use static as well as non-static variables.

Notes

- Call to malloc does not invoke the constructor. Call to free does not invoke the destructor.
- Call to new invokes the constructor with zero arguments.
- struct and class in C++ are almost the same. The only difference is that struct members are by default public, while class members are by default private.
  - Thus, you can define methods in struct in C++.

In these lectures

- ✔ Introduction to OOP
- ✔ Classes and Objects
- • Operator Overloading
- • Inheritance

Concepts are applicable in general.
We will use C++ and Linux as the environments.

Prerequisite:

- Programming experience

References:

- *The C++ Programming Language*, Bjarne Stroustrup, 4e, Pearson
- *C++ Primer Plus*, Stephen Prata, 6e, Pearson