Object-Oriented Programming & Class

Modern cpp Programming lecture 3

In this lecture…

- Object-Oriented Programming
- Class
- Other concepts of cpp which needs to understand class

Object-Oriented Programming?

- In the early days of computing…
  - programming == solving problem
  - programming was just the implementation of *algorithms*
  - focused on *the procedure to solve problem*
  - ***Procedural Programming***
    - Low complexity
    - High maintenance
    - Easy to understand

Object-Oriented Programming?

- However, as the complexity of software evolved…
  - Now software is not just algorithms!!
  - It rather became a kind of *simulation*
  - Procedural programming
    - inefficient to *simulate* complex logics
    - components, interactions, hierarchies…
    - => Object-Oriented Programming appeared!!

Object-Oriented Programming

- Simulates real world
- Software is the collection of components (a.k.a. modules)
- Each components is either a logic or ***Object***
- Programmers should define…
  - Objects,
  - their relationships,
  - and the interactions b/w objects

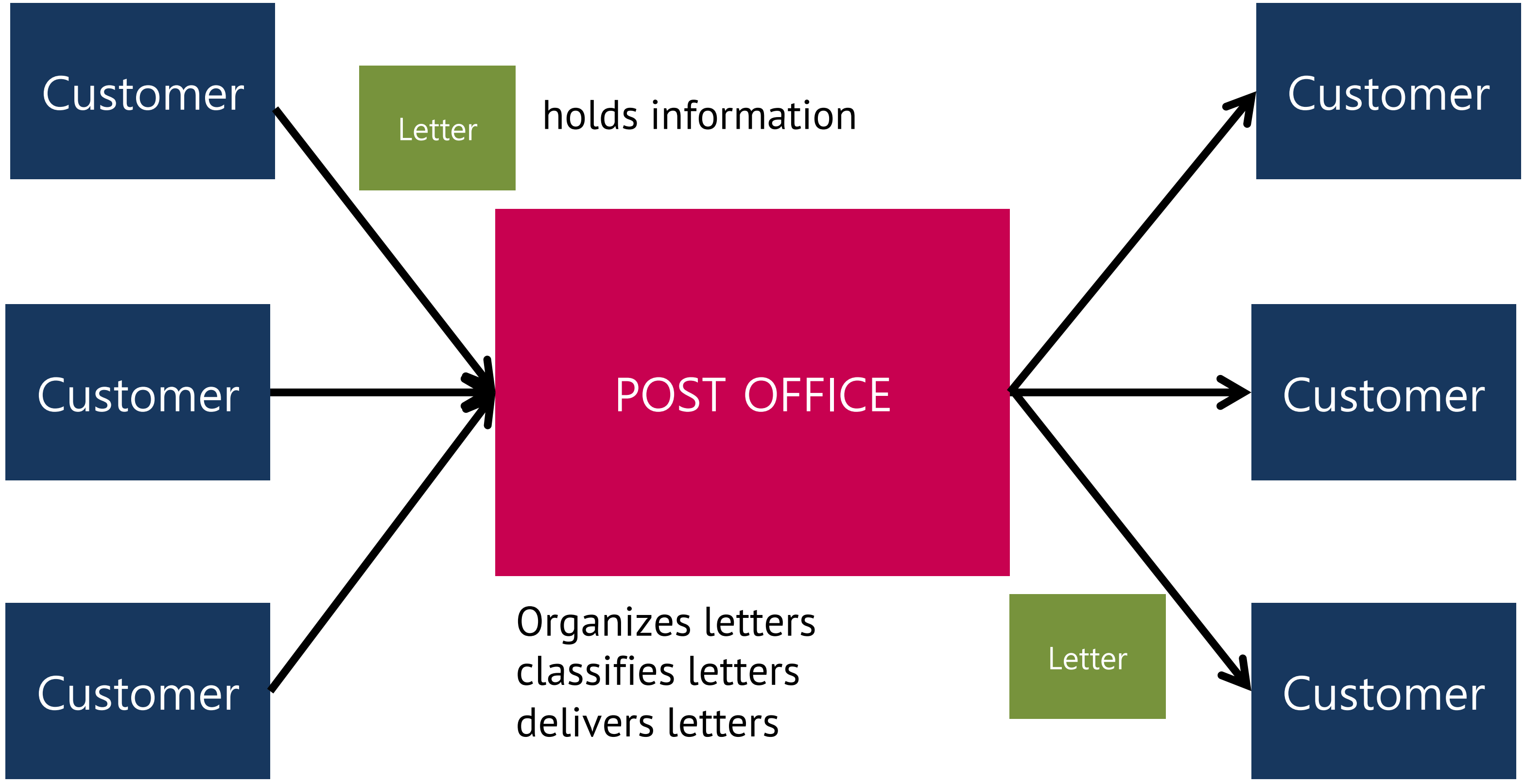
Object-Oriented Programming

- Advantages
  - Most of modern PL => supports OOP
  - Easy Abstraction
  - Supports GUI programming
  - Can easily build *Library* or *module*
    - Easy to reuse code!!
  - Supports Design Pattern (we'll see…)
- Disadvantage
  - slow

Object-Oriented Programming

- Simple example!!
- Suppose you’re developing a program…
- which simulates *post office*
- You should design…
  - Post office
  - Customers
  - Letter (or package)
- as objects

Object-Oriented Programming



Sends Letter
Receives Letter

Object-Oriented Programming

- cpp supports OOP
- Users can define ***classes***
- Class
  - *Class is not a object!!*
  - *Class is just a* ***Data type***
    - like int, float, string, bool…
  - ***User-defined data type***
  - Each class can generate multiple ***objects***

Object-Oriented Programming

- Terminology & Definition
  - Class
    - An extensible program-code-template for creating objects
  - Object
    - In computer science
      - can be a variable, data structure, a function or a method, and as such, is a blue in a value in memory referenced by an identifier.
    - In OOP
      - Refers to a particular instance of a class, where the object can be a com bination of variables, functions, and data structures

Object-Oriented Programming

- Terminology & Definition
  - Instance
    - concrete occurrence of any object, existing during the runtime of a program
    - is synonymous with *object* as they are each a particular value (realization), and these may be called an *instance object*.
    - Often refers the relationship b/w abstract concept and realization
      - *An Object is the instance of a Class*
      - *A Link b/w Objects is the instance of a relationship b/w Classes*
      - *A Process is the instance of certain Program, or executable*

Class (in cpp)

- Remember!!
  - Defining *new data type!!*
- Data type consists of…
  - value
    - int => integer number
    - string => sequence of characters
  - methods(functions)
    - int => addition, subtraction, multiplication…
    - bool => AND, OR, NOT…

Class

- *User-defined data type* is the same!!
- value
  - *variable*
- functions
  - *methods*
- So many features in class…
- example will help you!!

Class

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

variables

methods

Access Identifier

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

private and public??

Access Identifier

- Access identifier
  - public
    - accessible from itself / outside the class / child classes
  - private
    - accessible from itself
  - protected
    - accessible from itself / child classes
- child class? we'll see… (in inheritance)
- why access identifier??

Access Identifier

- Why access identifier??
  - if you're working alone…it's okay
  - but if other programmers should use code…
    - collaboration, building library
  - It is better to hide specific information & implementation
  - and just give the concrete way to access class!!
  - ***Information hiding!!***
  - *privatize* information and *publicize* access methods!!

Access Identifier

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

direct access to age variable is impossible!!

age can be only changed by aging() method

identifies ownership (class "Human")

```
void Human :: aging() {
   this->age++;
}
```

quite natural...

"this" is the pointer to current object(class instance)

Accessing objects

- Object variable can be either pointer or not

```
Human human;
Human* humanPointer = new Human();

human.aging();
humanPointer->aging();
```

- use ”.” if normal variable
- use –> if pointer
- to access it’s element (variable or method)

Getter & Setter

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

better to maintain variables as private
Why? publicizing variables
decreases coherency

Therefore, we need getter / setter to access and modify variables!!

Getter & Setter

- get / set : convention

```
void Human :: setName(string newName) {
   this->name = newName;
}

string Human :: getName() {
   return this->name;
}
```

- Always remember!!

- better to privatize variable and publicize it’s getter / setter

Constructor & Destructor

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

Constructor

- Constructor
  - literarily “constructs” the class!!
  - constructor name == name of the class
- Default constructor
  - no parameter
  - set default initialization

```
Human :: Human() {
   this->age = 0;
   this->name = "Alice";
   this->nationality = "Korea, Republic of";
}

Human* defaultHuman = new Human();    // dynamic allocation
Human defaultHuman;                   // static declaration
// in both cases, program calls default constructor
```

Constructor

- User-defined Constructor

```
Human :: Human(int age, string name, string nationality) {
   this->age = age;
   this->name = name;
   this->nationality = nationality
}
```

parameter

variable

- **“Overloading”!!**
- Same function name, different *signature*
- Users can define various constructor using *function overloading*

Function Overloading

- Different signature -> Different function!!
- signature:
  - function name, argument type, argument number

```
int add(int a, int b) {
   return a + b;
}

int add(int a, int b, int c) {
   return a + b + c;
}

add(1, 2)      // 3
add(1, 2, 3)   // 6
```

  - return type is not the member of signature!!

Copy Constructor

- If you want to *copy* such objects…

- should define & call *copy constructor!!*

```
class Book{
private:
   string title;
   string* authors;

public:
   Book() {
      this->title = "untitled";
      this->authors = new string[3];
      this->authors[0] = "Alice";
      this->authors[1] = "Bob";
      this->authors[2] = "Carol";
   }

   Book(const Book& book) {
      this->title = book.title;
      this->authors = book.authors;
   }
};
```

```
Book book1;
Book book2(book1) // book2 copies book1
```

should follow the form
”const” or “&”??

copy constructor (copies “book”)

Constant variable

- While writing program…
  - some variables should not be modified!!
    - critical features in OS
    - number of commands…
  - However, while collaborating, these variables can be changed!!
  - To prevent such situation, we use const identifier
  - if the program tries to change const variable, it generates error

```
void printConstant(const int a) {
   const string format = "Number: ";
   a = 10;
   format = "Integer";
   cout << format << a << endl;
}
```

compile error!!!!

Reference

- Assigning *new name* to the variable!!

- Uses & identifier

- You can access to the variable with any name!!

```
int a = 3;
int& b = a;
cout << a << endl;      // 3
cout << b << endl;      // 3
```

- can also get arguments as references (*call-by-reference)*

```
void modifyInt(int& a) {
    a = 5;
}

int a = 8;
cout << a << endl;      // 8
modifyInt(a);
cout << a << endl;      // 5
```

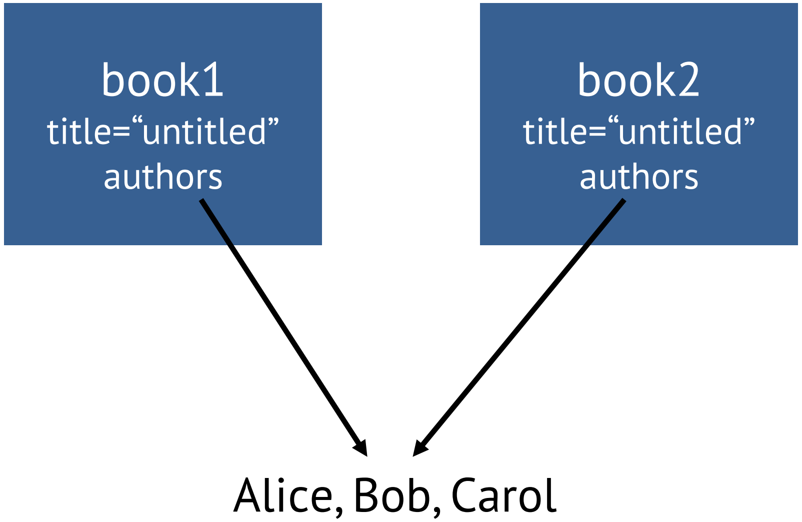
guess why?
You can use reference instead of pointer!!

Copy Constructor

- Okay, finished understanding copy constructor?
- No!!

```
class Book{
private:
   string title;
   string* authors;

public:
   Book() {
      this->title = "untitled";
      this->authors = new string[3];
      this->authors[0] = "Alice";
      this->authors[1] = "Bob";
      this->authors[2] = "Carol";
   }

   Book(const Book& book) {
      this->title = book.title;
      this->authors = book.authors;
   }

   void changeFirstAuthor(string author) {
      this->authors[0] = author;
   }
};
```

```
Book book1;
Book book2(book1)
```



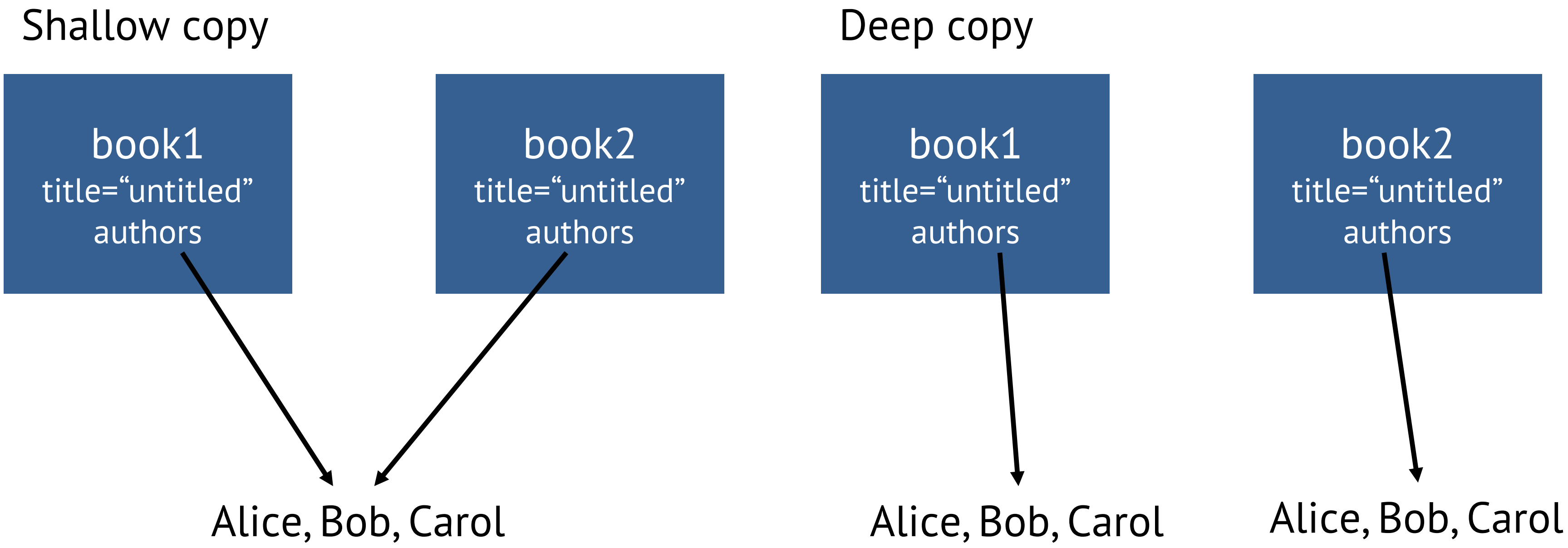
book1
title="untitled"
authors

book2
title="untitled"
authors

Alice, Bob, Carol

If you modify the authors of book1, it also applies to book2

Deep Copy & Shallow copy



should consider the variables inside objects!!

```
Book(const Book& book) {
   this->title = book.title;
   this->authors = new string[3];
   this->authors[0] = book.authors[0];
   this->authors[1] = book.authors[1];
   this->authors[2] = book.authors[2];
}
```

Destructor

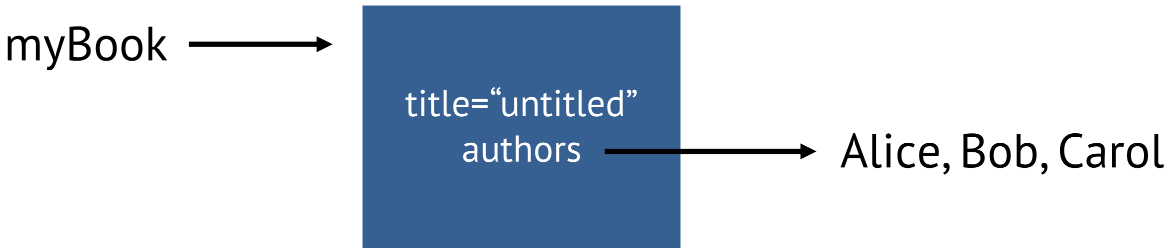
```
class Book{
private:
   string title;
   string* authors;

public:
   Book() {
      this->title = "untitled";
      this->authors = new string[3];
      this->authors[0] = "Alice";
      this->authors[1] = "Bob";
      this->authors[2] = "Carol";
   }

   Book(const Book& book) {
      this->title = book.title;
      this->authors = book.authors;
   }

   ~Book() {
        delete[] authors;
   }
};
```

Destructor

- Calls when the object is destroyed
  - free dynamic allocation
  - end of a function
  - end of a program
- Denoted as ~ + class name
- If the object holds the dynamic-allocated data…
- Destroying object without deallocating the data…
  - is a disaster!!
  - makes dangling memory

Destructor

```
Book* myBook = new Book();
```



myBook

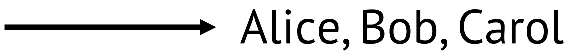
title=“untitled”

authors

Alice, Bob, Carol

if no destructor...default destructor calls!! -> only removes it’s variables

```
delete myBook;
```



Alice, Bob, Carol

with destructor, every memory can be freed safely

```
~Book() {
    delete[] authors;
}
```

Class

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   Human();
   Human(int age, string name, string nationality);
   ~Human();

   int getAge();
   string getName();
   string getNationality();

   void aging();
   void setName(string newName);
   void setNationality(string newNationality);

   string printPersonalInfo();
   Human* makeProduct(string work);

};
```

Class

- You can define / call any objects and it’s method

```
class Human {
private:
   int age;
   string name;
   string nationality;

public:
   ...

   string printPersonalInfo();
   Human* makeProduct(string work);
};

Human :: printPersonalInfo() {
   cout << “Age: ” << age << endl;
   cout << “Name: ” << name << endl;
   cout << “Nationality: “ << nationality << endl;
}

Human human(20, “XiaXia”, ”China”);
human.printPersonalInfo();
```

Thank you!!

contact: jeonhyun97@postech.ac.kr