# OOP in C++

Lab 11

TA : Hyuna Seo, Kichang Yang, Minkyung Jeong, Jaeyong Kim


SEOUL NATIONAL UNIVERSITY

# Announcement

- You should finish the lab practice and submit your job to eTL before the next lab class starts(Wednesday, 7:00 PM).
- The answer of the practice will be uploaded after the due.

# Overview

- Recap: OOP in C++
  - Class Declaration
  - Data Members / Member Functions
  - Constructors / Copy Constructor
  - Destructors
- Problem
  - Problem 1 - Point and Grid

# Class Declaration

- As in Java, the `class` is the fundamental unit to enable object-oriented programming.
  - They can contain attributes and functions.
- An object is an instantiation of a class.

```
class class_name {
 access_specifier:
   member1;
 access_specifier:
   member2;
 ...
};
```

```
class Rectangle {
    int width, height;
 public:
    void set_values (int,int);
    int area (void);
};
```

```
// Create an object
Rectangle rect;
```

# Data Members

- A variable declared in a class is called a data member. It can be accessed with the dot operator(`.`).

```
#include <iostream>

class S {
  public:
    int a[2] = {1, 2};
    std::string s = "Hello";
};
int main() {
    S obj;
    std::cout << obj.s << std::endl;
    std::cout << obj.a[0] << "," << obj.a[1] << std::endl;
}
```

| Output |
| --- |
| Hello<br>1,2 |

# Member Functions

- Functions declared inside the class are called member functions.
- It can be accessed with dot operator(`.`).

```
#include <iostream>

class MyClass {
  public:
    void myMethod() {
      std::cout << "Hello!"
                << std::cout;
    }
};
```

```
int main() {
  MyClass myObj;
  myObj.myMethod();
  return 0;
}
```

```
Hello!
```

# Constructors

- Must be specified with `public` access.
- Initializes data members in various ways.

```
class class_name {
    public:
        class_name(): member_initializer_list {
            // Constructor Body
        }
};
```

```
class S {
    public:
        int n;
        S() { n = 7; }
};
```

```
#include <iostream>

int main() {
    S s;
    std::cout << s.n << std::endl;
}
```

# Destructors

- A destructor is called when the lifetime of an object ends.
  - e.g. program termination, end of scope, etc.
- The destructor is used to free the resources that the object may have acquired during its lifetime.

```
class class_name{
    // Other members
    ~class_name(){
        // Destructor Body
    }
};
```

# Copy Constructor

- It creates an object based on an object of the same class, which has been created previously.

```
// Syntax
class class_name{
  class_name(const class_name&
other) {
    // Copy Constructor Body
  }
};
```

```
class A {
public:
    int n;
    A(int n = 1) : n(n) { }
    A(const A& a) : n(a.n) { }
};
```

```
#include <iostream>
int main(){
  A a1(7); A a2(a1);
  std::cout << a2.n
    << std::endl; // 7
}
```

# Copy Constructor

- Copy constructor is called when an object is initialized from another object of the same type:
  - Initialization:

    `T a = b;` or `T a(b);`, where `b` is of type `T`.

  - Function argument passing by value:

    `f(a);`, where `a` is of type `T` and `f` is `void f(T t)`.

  - Function return by value:

    `return a;` inside a function like `T f()`, where `a` is of type `T`.

# Shallow Copy with Implicit Copy Constructor

- Implicit copy constructor also copies pointer-type attributes, but **simply copy the address (shallow copy), NOT the object (deep copy)**.

```
#include <string>

class Director {
public:
  std::string name;
  Director(std::string name):
    name(name) { }
};
```

```
class Movie {
public:
  Director* director;
  Movie(Director* director):
    director(director) { }
};
```

# Shallow Copy with Implicit Copy

```
#include <iostream>
int main() {
    Director* director = new Director("Bong Joon-ho");
    Movie movie(director);
    std::cout << movie.director->name << std::endl;
    Movie movie_copied(movie);
    delete director;
    std::cout << movie_copied.director->name << std::endl;
}
```

```
Bong Joon-ho
(? x?                    ╝                              ╝        ╝?
@Q쩥              |?
...
```

# Deep Copy with Explicit Copy Constructor

- You can copy the referenced object of pointer-type attribute with your own copy constructor.

```
#include <string>
class Director {
public:
    std::string name;
    Director(std::string name): name(name) { }
    Director(Director const &director): name(director.name) { }
};
class Movie {
public:
    Director* director;
    Movie(Director* director): director(director) { }
    Movie(Movie const &movie): director(new Director(*movie.director)) { }
};
```

# Deep Copy with Explicit Copy Constructor

```
#include <iostream>
int main() {
    Director* director = new Director("Bong Joon-ho");
    Movie movie(director);
    std::cout << movie.director->name << std::endl;
    Movie movie_copied(movie);
    delete director;
    std::cout << movie_copied.director->name << std::endl;
}
```

```
Bong Joon-ho
Bong Joon-ho
```

# Objectives

- Get used to OOP in C++
- Problem 1 - Point and Grid
  - Class Declaration
  - 2D Pointer (Prerequisites for HW5)
  - Copy Constructor
  - Destructor
  - Interface vs Implementation

# Problem Overview

- Problem 1 - Point and Grid
  - 1-1 Construct Point, Grid class (0:10)
  - 1-2 Copy Grid (0:10)
  - 1-3 Clean-up Grid class (0:05)
  - 1-4 Interface vs Implementation (0:10)

10 mins



# Problem 1 : Point and Grid class

- Implement simple Point and Grid class
- `Point` class
  - Implement a point class that represents a point in x and y coordinates.
  - Implement getter to access private variable `x` and `y`.

10 mins

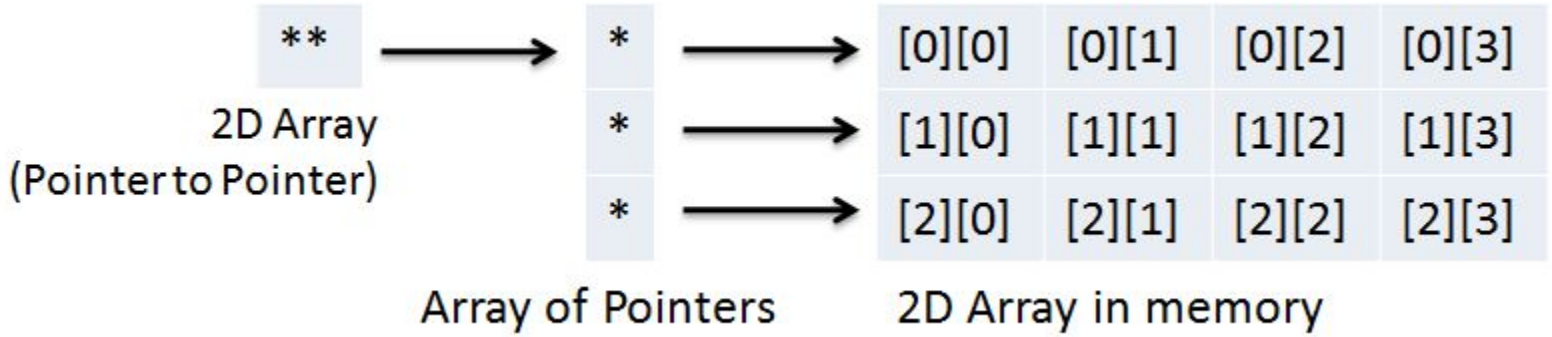


# Problem 1 : Point and Grid class

- Implement simple Point and Grid class
- `Grid` class
  - Implement a 2D grid that represents a 2D array.
    - Initialize the size of the `grid` with `row` and `column`.
    - Initialize all the contents of the grid with zeros.
  - Implement getter/setter to access private data members of the Grid class.
  - Implement a `printGrid` method that prints all the contents of the grid.

# 2D Pointer

- How to represent 2D array in C++?



10 mins



# Problem 1 : Point and Grid class

main.cpp

```
#include <iostream>

class Point {
    int x, y;
public:
    //TODO Prob1.1 initialize Point and print x and y
};


class Grid {
    int** grid;
    int row, column;
public:
    //TODO Prob1.1 initialize Grid with zeros
};
```

10 mins



# Problem 1 : Point and Grid class

main.cpp

```
...

 int main() {
     Point p(1,3);
     Grid g(2,3);

     std::cout << "x : " << p.getX() << ", y : " << p.getY() << std::endl;

     g.printGrid();

     return 0;
}
```

Output

```
 x : 1, y : 3
 grid :
 0 0 0
 0 0 0
```

# Q&A

10 mins



# Problem 2 : Copy Grid

- Implement a `printNumberGrid(Grid g)` method that prints a grid with the same shape as the given `Grid g`.
- However, the printed grid is filled with increasing numbers.

**Original Grid g**

```
2 6 5 7
9 1 8 3
```

**Printed Grid**

```
0 1 2 3
4 5 6 7
```

10 mins



# Problem 2 : Copy Grid

- We want to print only a new grid while maintaining the contents of existing grid.
  - Once the method is called, copy constructor is called to construct another object of the `Grid`.
  - However, implicit copy constructor only copies the address of `grid`, not the object.
  - If there's a change in the new grid, it also affects the original grid!!
- Create your own explicit copy constructor!

10 mins



# Problem 2 : Copy Grid

main.cpp

```
...

class Grid {
    int** grid;
    int row, column;
public:
    //TODO Prob1.1 initialize Grid with zeros

    //TODO Prob1.2 create explicit copy constructor
...

};

//TODO Prob1.2 print a grid with increasing numbers in the shape of the given Grid g
```

10 mins



# Problem 2 : Copy Grid

main.cpp

```
...
int main() {
    Point p(1,3);
    Grid g(2,3);

    g.printGrid();

    printNumberGrid(g);

    g.printGrid();

    return 0;
}
```

10 mins



# Problem 2 : Copy Grid

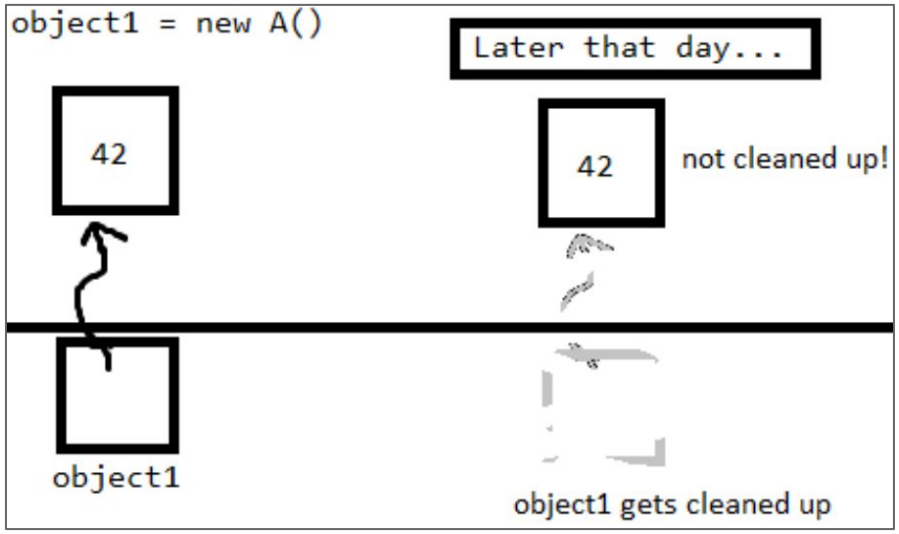
Output

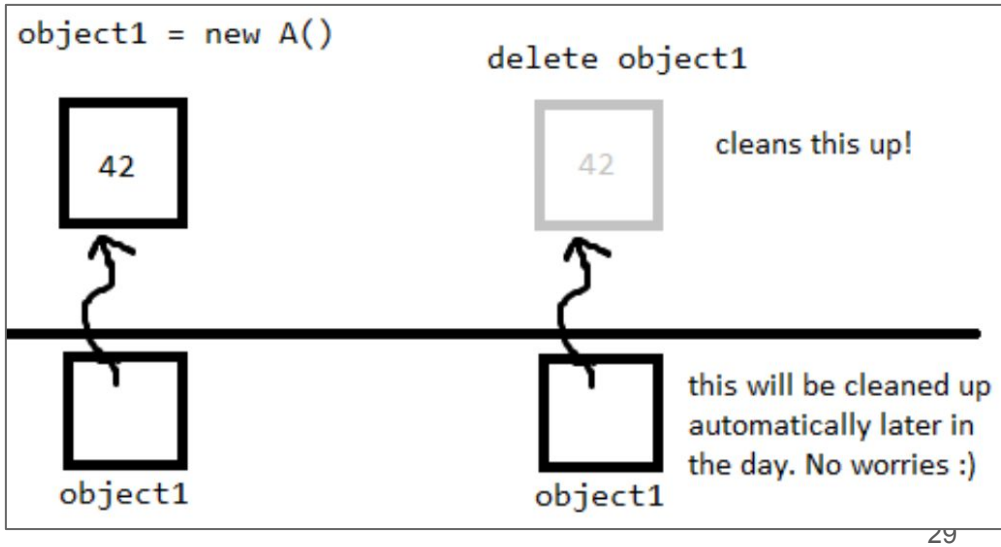
```
grid :
0 0 0
0 0 0
grid :
0 1 2
3 4 5
grid :
0 0 0
0 0 0
```

# Q&A

# Problem 3 : Clean-up Grid class

- Clean-up Grid class!
  - Unlike java, C++ does not automatically clean the memory of objects created with `new`.
  - We need to manage the resource by explicit `delete` at object destruction



5 mins



# Problem 3 : Clean-up Grid class

- Define a destructor to prevent resource leaks
  - A destructor is implicitly invoked at the end of an object's lifetime.
  - Once the destructor is called, print "Clean-up Grid".
  - Use `delete[]` to delete the array of pointers.

5 mins



# Problem 3 : Clean-up Grid class

main.cpp

```
...
class Grid {
    int** grid;
    int row, column;
public:
    //TODO Prob1.1 initialize Grid with zeros

    //TODO Prob1.2 create explicit copy constructor

    //TODO Prob1.3 Add proper clean-up code!
};
...
```

5 mins



# Problem 3 : Clean-up Grid class

main.cpp

```
...
int main() {
    Point p(1,3);
    Grid g(2,3);

    std::cout << "x : " << p.getX() << ", y : " << p.getY() << std::endl;

    g.printGrid();

    return 0;
}
```

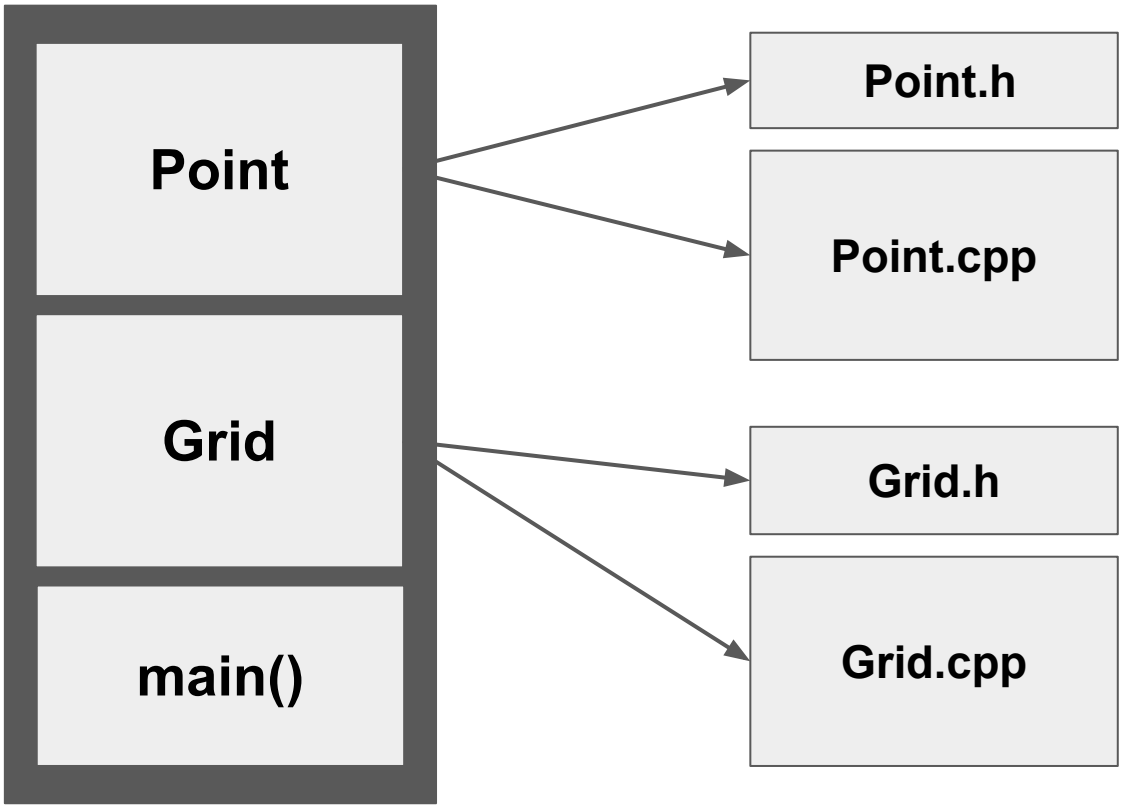
Output

```
x : 1, y : 3
grid :
0 0 0
0 0 0
Clean-up Grid
```

# Q&A

# Problem 4 : Interface vs Implementation

- Let's move Point / Grid class to Point.h / Grid.h and Point.cpp / Grid.cpp



10 mins



# Problem 4 : Interface vs Implementation

- Distinguish between an interface and its implementation “details” using a class.
- Readability and simpler maintenance.

```
// Interface
class Date {
 int y;  Month m;  char d;
public:
 Date();
 Date(int yy, Month mm, char
dd);
 char day() const;
 Month month() const;
 int year() const;
};
```

```
// Implementation Detail
Date::Date(int yy, Month
mm, char dd):
y(yy), m(mm), d(dd){}
Date::day(){ return d; }
Date::month(){ return m; }
Date::year(){ return y; }
```

10 mins



# Problem 4 : Interface vs Implementation

## main.cpp

```
#include <iostream>
#include "Point.h"
#include "Grid.h"

//TODO Prob1.2 print a grid with increasing numbers in the shape of the given Grid g

int main() {
    Point p(1,3);
    Grid g(2,3);

    std::cout << "x : " << p.getX() << ", y : " << p.getY() << std::endl;

    g.printGrid();

    return 0;
}
```

## Output

```
 x : 1, y : 3
 grid :
 0 0 0
 0 0 0
Clean-up Grid
```

# Q&A

# Exercise

- Let's create a method that marks the given point in the grid!
  - Currently, the contents of the grid are all initialized to zero.
  - Mark the given point with the increasing counter
    - Mark Counter
      The counter number starts from 1
      Use static members to track the current number of marker
    - Check the validity of the point to mark within the grid
      Ignore the invalid points (eg. out of range)

# Exercise

main.cpp

```
...
int main() {
    Grid g(2,3);

    g.printGrid();

    Point p1(1,0);
    Point p2(0,1);
    Point p3(3,3); // Invalid point

    g.mark_point(p1);
    g.mark_point(p2);
    g.mark_point(p3);

    g.printGrid();

    return 0;
}
```

# Exercise

Output

```
grid :
0 0 0
0 0 0
grid :
0 2 0
1 0 0
Clean-up Grid
```

# Submission

- Download skeleton files from eTL
- Compress your Project directory into a zip file.
- Rename your zip file as 20XX-XXXXX_{name}.zip
  - for example, 2021-12345_JeongMinkyung.zip
- Upload it to eTL - Lab 11 assignment.

# C++ Core Guidelines


- Need a coding guideline to rely on, and effectively use this complex language.
  - Similar to design pattern in Java, but official.
    - Made by the creator of the C++ (Bjarne Stroustrup) himself. Maintained by experts at CERN, Microsoft, etc like Herb Sutter.
  - Aims simplicity and safety. (type-safe, no resource leak)
  - To help someone who is less experienced or coming from a different background or language.

# C++ Core Guidelines

- C++ Core Guidelines : https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-class

- Official site : https://github.com/isocpp/CppCoreGuidelines

# C++ Core Guidelines

- Guideline does not teach you the syntax itself, but rather how to use it effectively.
- GSL (Guided Support Library) : C++ library to support this guidelines (but not useful currently)

# Thank You!!!