OOP in C++

Lab 10

TA : Changmin Jeon, Minji Kim, Hyunwoo Jung,
Hyunseok Oh, Jingyu Lee, Seungwoo Jo

SEOUL NATIONAL UNIVERSITY

Prerequisites for Homework 5

- Exercise 1 - Operator << overloading
  - More to come in Lecture 16 (Polymorphism)
- Exercise 2 - 2D Pointer

Operator << overloading

- How to print Date class

```
class Date {
public:
   Date(int yy, int mm,
        char dd);
   int y;
   int m;
   char d;
} ;
```

```
#include <iostream>
#include “Date.h”
using namespace std;
int main(){
 Date date;
 cout << date.y << "-"
      << date.m << "-"
      << date.d << endl;
}
```

Operator << overloading

- Similar to toString in Java

```
class Date {
public:
   Date(int yy, int mm, char dd);
   friend std::ostream&
   operator<<(std::ostream& os, const Date& date);
   int y; int m; char d;
} ;

std::ostream& operator<<(std::ostream& os, const Date& date) {
   return os << y << “-” << m << “-” << d;
}
```

Operator << overloading

```
#include <iostream>
#include “Date.h”
using namespace std;
int main(){
 Date date(2020, 1, ‘f’);
 cout << date.y << "-"
      << date.m << "-"
      << date.d << endl;
}
```

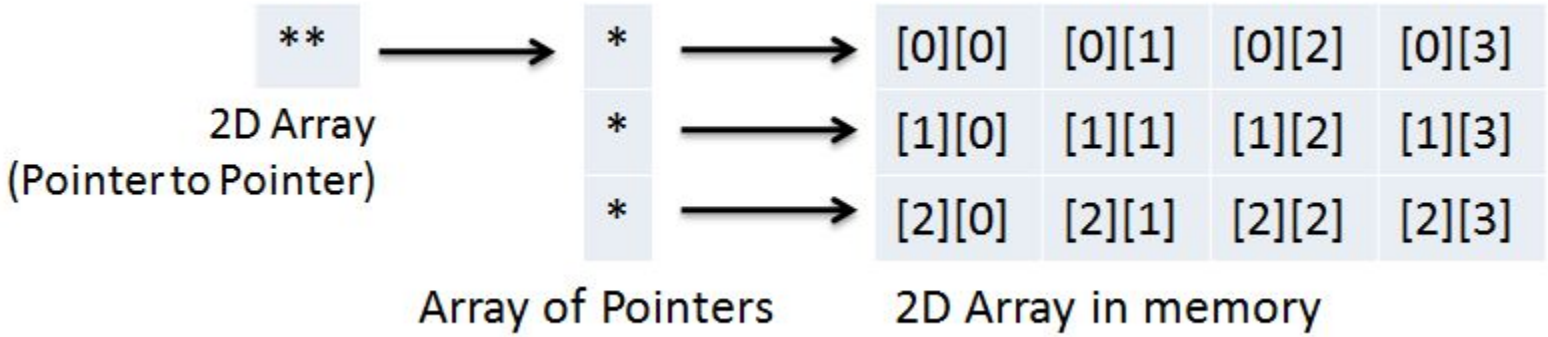
2020-1-f

```
#include <iostream>
#include “Date.h”
using namespace std;
int main(){
 Date date(2020, 1, ‘f’);
 cout << date << endl;
}
```

2020-1-f

2D Pointer

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



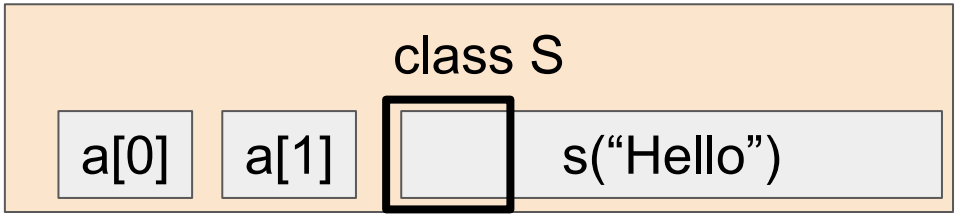
C++ Core Guidelines

- C++ is a very complex language.
  - Syntax with lots of features Java does not have.
    - friends, operator overloading, references, inline, copy constructor, destructor, member initializer lists, etc.
  - The more the syntactic complexity, the more harder to learn, the less the productivity.
    - Study[1] have shown that in development,
    - the C++ will likely generate 2~3 times more bug than Java.
    - Java is 30~200% more productive than C++.

[1] Phipps, G. (1999). Comparing observed bug and productivity rates for Java and C++. *Software: Practice and Experience*, *29*(4), 345-358.

Motivation

- C++ is a very complex language.
  - Needs to know the low-level execution mechanism of the C++ (to a certain extent)



a[2] access attempt returns data here

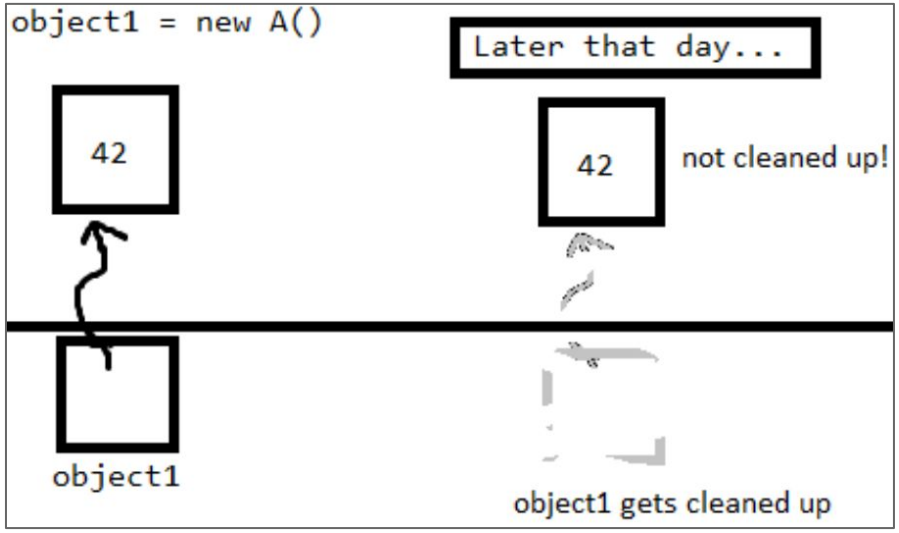
```
#include <iostream>
#include <string>
class S {
   public:
       int a[2] = {1, 2};
       string s = "Hello";
};
```

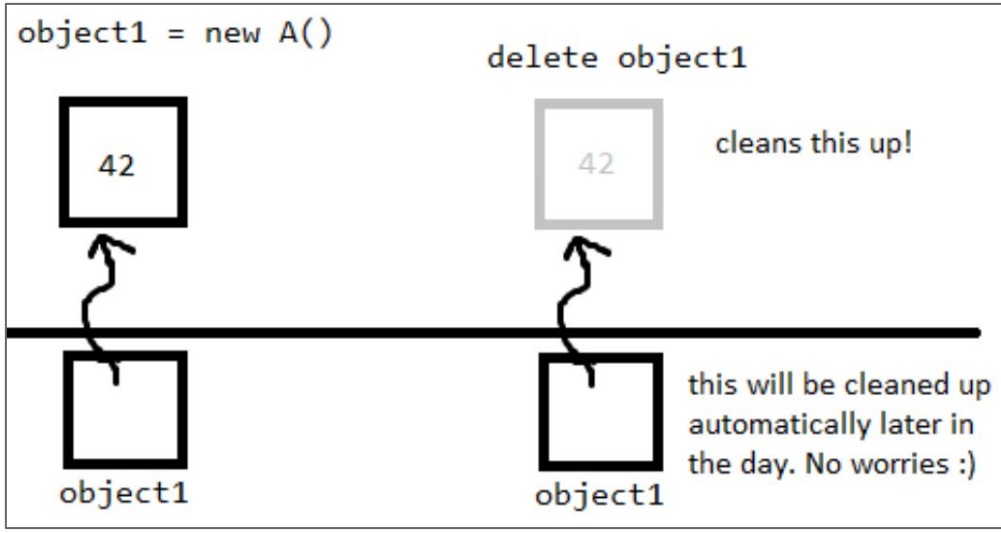
```
int main(){
 S s;
 cout << s.a[0] << ","
      << s.a[1] << endl;
 cout << s.a[2] << endl;
}
```

```
1,2
6487808
```

Motivation

- C++ is a very complex language.
  - Need to be very careful for resource management.
    - Explicit management to prevent resource leak.
    - delete-new, destructor



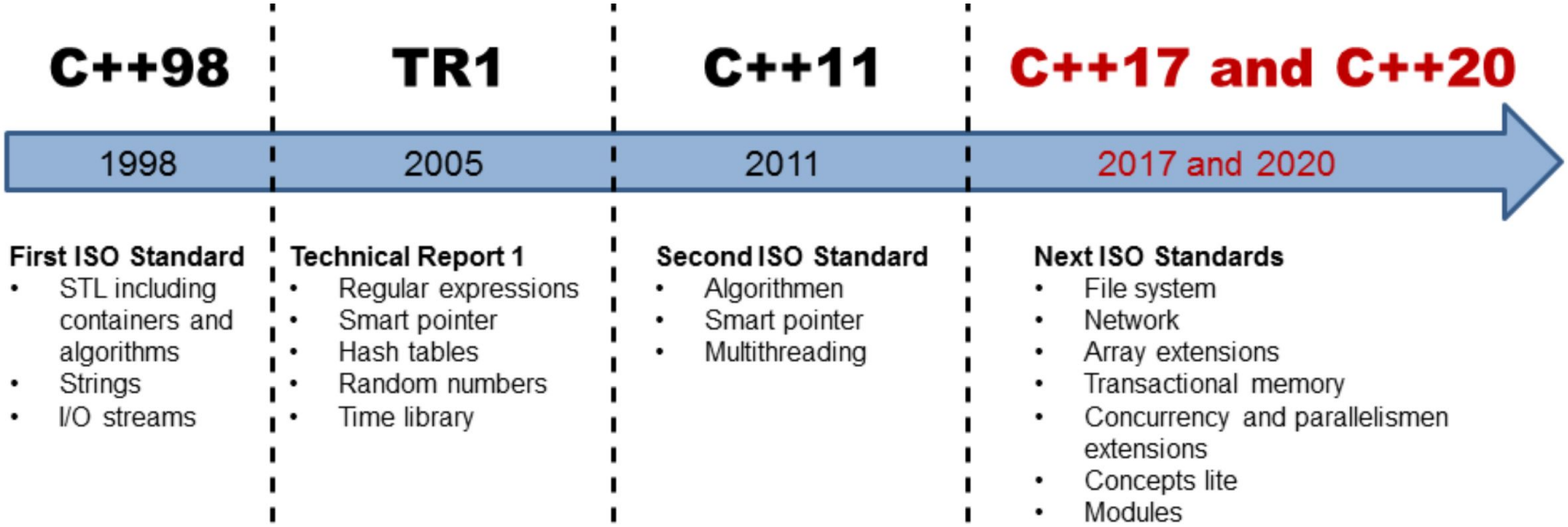


Motivation

- C++ is a very complex language.
  - Complexity due to compatibility with C
    - Pointers, Macro, struct
    - Might mix up C and C++ style code
    - Weaker type safety, overuse of pointer, imperative programming (non-OOP), ...

Motivation

- C++ is a very complex language.
  - Continuous Large-scale changes on C++ standard.
    - C++ -> C++2.0 -> C++98 -> C++03 -> C++11 -> C++14 -> C++17
    - Lots of additional features, for an already complex language.



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
- C++ Core Guidelines (Korean Translation) : https://github.com/CppKorea/CppCoreGuidelines/tree/sync/sections
- Official site : https://github.com/isocpp/CppCoreGuidelines

C++ Core Guidelines

- The content in this document itself will **not** be in final exam.
  - But its content will help your implementation with C++, and improve coding style.
  - Many of the rules are prescriptive. We are uncomfortable with rules that simply state "don't do that!" without offering an alternative.
  - It is your choice to follow this guideline or not, and some of the rules may collide with your own rules.

OOP & Classes

- In this lecture,
  - Only introduce *C : classes and class hierarchies.*

C: Classes and class hierarchies

A class is a user-defined type, for which a programmer can define the representation, operations, and interfaces. Class hierarchies are used to organize related classes into hierarchical structures.

Class rule summary:

- C.1: Organize related data into structures (`struct` s or `class` es)
- C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently
- C.3: Represent the distinction between an interface and an implementation using a class
- C.4: Make a function a member only if it needs direct access to the representation of a class
- C.5: Place helper functions in the same namespace as the class they support
- C.7: Don't define a class or enum and declare a variable of its type in the same statement
- C.8: Use `class` rather than `struct` if any member is non-public
- C.9: Minimize exposure of members

Class definition and instantiation(C.7)

- Don’t define a class and declare a variable of its type in the same statement.
  - Confusing and unnecessary.

```
// BAD
class Date {
public:
   // validate and initialize
   Date(int yy, Month mm,
      char dd);
private:
   int y;  Month m;  char d;
} cur_date;
```

```
// GOOD
class Date {
public:
   // validate and initialize
   Date(int yy, Month mm,
      char dd);
private:
   int y;  Month m;  char d;
} ;
Date curdate;
```

Related data into classes (or struct)

- Ease of comprehension.
- If data is related, that fact should be reflected in code. (C.1)
  - The criteria of ‘related’ data is heuristic.
  - In the below case, the reader do not have to think of implicit relationship of (x,y) and (x2,y2)

```
void draw(int x, int y, int x2, int y2);
// BAD: unnecessary implicit relationships
void draw(Point from, Point to);
// better
```

Problem 1

- Simplify draw function with Point class and Operator << overloading

```
void draw(int x, int y, int x2, int y2) {
    std::cout << "from : " << x << " " << y << std::endl;
    std::cout << "to : " << x2 << " " << y2 << std::endl;
}
```

```
void draw(Point p, Point p2)
```

Minimize exposure of members (C.9)

- Encapsulation. Information hiding.
- Minimize the chance of unintended access.
- This simplifies maintenance.

```
class Distance {
public:
   double meters() const { return magnitude*unit; }
   void set_unit(double u){ // validity check of u
            unit = u;
   } // ...
private:
   double magnitude;
   double unit;    // 1 is meters, 1000 is kilometers,
0.001 is millimeters, etc.
};
```

Problem 2

- Add validity check in `set_unit` function!

```
// TODO: Add validity check
// Condition 1 : u should be non-negative number
// Condition 2 : u should be powers of ten
// Hint: std::log10, std::pow(10, n)
void set_unit(double u) {
    unit = u;
}
```

Interface vs Implementation (C.3)

- 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();
 // validate and initialize
 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; }
```

Problem 3

- Let’s move date class to Date.h and Date.cpp

```
#include <iostream>
#include "Date.h"

int main() {
    Date d(2012, 11, 'f');
    std::cout << d << std::endl;
    return 0;
}       You, seconds ago • Uncomm
```

Class vs Struct (C.2)

- Use class if the class has an invariant;
  - Invariant : data that should not vary with an independent access.
  - Constructor is a way to completely initialize an object.
- Use struct if the data members can vary independently.
- Readability. Ease of comprehension.

```
struct Pair {
// the members can
vary independently
    string name;
    int volume;
};
```

```
class Date {
public:
    // validate and initialize
    Date(int yy, Month mm, char dd);
private:
    int y;  Month m;  char d; // day
};
```

Class vs Struct (C.8)

- Use class rather than struct if any member is non-public.
  - Readability.
  - To make it clear that something is being abstracted and encapsulated.

```
// BAD
struct Date {
 Month m;  char d;
 Date();
 Date(int yy, Month mm,
      char dd);
private:
 int y;
};
```

```
// GOOD
class Date {
 int y;  Month m;  char d;
public:
 Date();
 Date(int yy, Month mm,
      char dd);
};
```

Special Member Functions (C.20)

- If you can avoid defining special member functions(Constructor, Destructor, Copy constructor,...), avoid defining it.
  - Simple, clean semantics.
  - Rule of Zeros.

```
struct Named_map {
public:
    // ... no default operations declared ...
private:
    string name;
    map<int, int> rep;
};
Named_map nm;          // default construct
Named_map nm2 {nm};    // copy construct
```

Constructor (C.41)

- A constructor should create a fully initialized object.
  - A user of a class should be able to assume that a constructed object is usable.

```
class X1 {
 FILE* f;
public:
 void init(); // initialize f
 void read(); // read from f
};
void f(){
 X1 file;
 file.read(); // crash!
 file.init(); // too late
}
```

```
class X1 {
 FILE* f;
public:
 X1() {...} // initialize f
 void read(); // read from f
};
void f(){
 X1 file;
 file.read();
}
```

Copy Constructor / Assignment(C.61)

- Copy operation should copy.
  - Copy operation call are assumed to copy. Nothing less.
  - After the copy, same members from different objects can be
    - Independent (deep copy)
    - Refer to a shared object (shallow copy, through pointer)

Destructor (C.30)

- Define a destructor if a class needs an explicit action at object destruction.
  - A destructor is implicitly invoked at the end of an object's lifetime. If the default destructor is sufficient,

```
// BAD
class Foo {
public:
    // ...
    ~Foo() { s = ""; i = 0; }  // clean up
private:
    string s;
    int i;
};
```

Destructors (C.31)

- All resources acquired by a class must be released by the class's destructor.
  - To prevent resource leaks.

```
class X {
    ifstream f;
    // may own a file
};
// ifstream implicitly closes opened file on its destruction.
```

```
class X2 { // BAD
    FILE* f;
// may own a file
};
// No explicit delete of the FILE, may leak a file handle.
```

Problem 4

- Clean-up Grid class!

```
int main() {
    Grid2d grid(5, 10);

    for (int r = 0; r < grid.getRow(); r++) {
        for (int c = 0; c < grid.getColumn(); c++) {
            grid.setAt(r * grid.getColumn() + c, r, c);
        }
    }

    std::cout << grid << std::endl;

    // Potential memory leak!!!!
    return 0;
}       You, seconds ago • Uncommitted changes
```

```
// TODO : Add proper clean-up code!
// ~Grid2d()
```

C++ Core Guidelines

- ...And more guidelines after that.
  - On Philosophy of coding, resource management, performance,...

P.1: Express ideas directly in code

P.2: Write in ISO Standard C++

P.3: Express intent

P.4: Ideally, a program should be statically type safe

P.5: Prefer compile-time checking to run-time checking

P.6: What cannot be checked at compile time should be checkable at run time

P.7: Catch run-time errors early

P.8: Don't leak any resources

P.9: Don't waste time or space

P.10: Prefer immutable data to mutable data

P.11: Encapsulate messy constructs, rather than spreading through the code

P.12: Use supporting tools as appropriate

P.13: Use support libraries as appropriate

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)

Submission

- Download skeleton files from eTL
- Compress your Project directory into a zip file.
  - It should include problem1.cpp ~ problem4.cpp
- Rename your zip file as 20XX-XXXXX_{name}.zip
  - for example, 2020-12345_KimMinji.zip
- Upload it to eTL - Lab 10 assignment.