C++ OOP & Classes

Lab 09

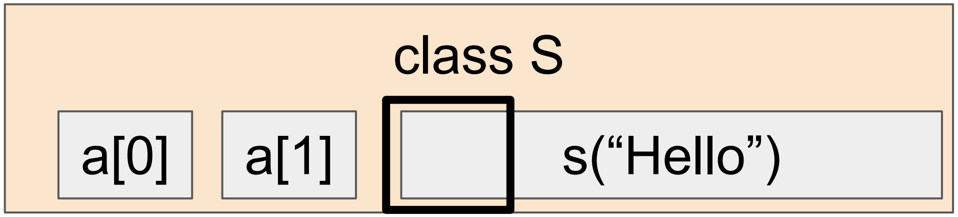
C++ Core Guidelines

- 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

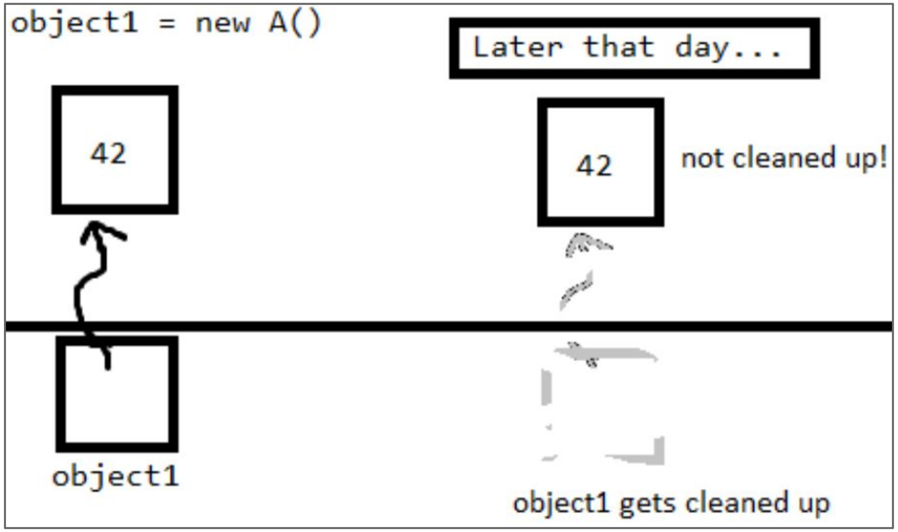
```
#include <iostream>
#include <string>
class S {
   public:
      int a[2] = {1, 2};
      string s = "Hello";
};
```

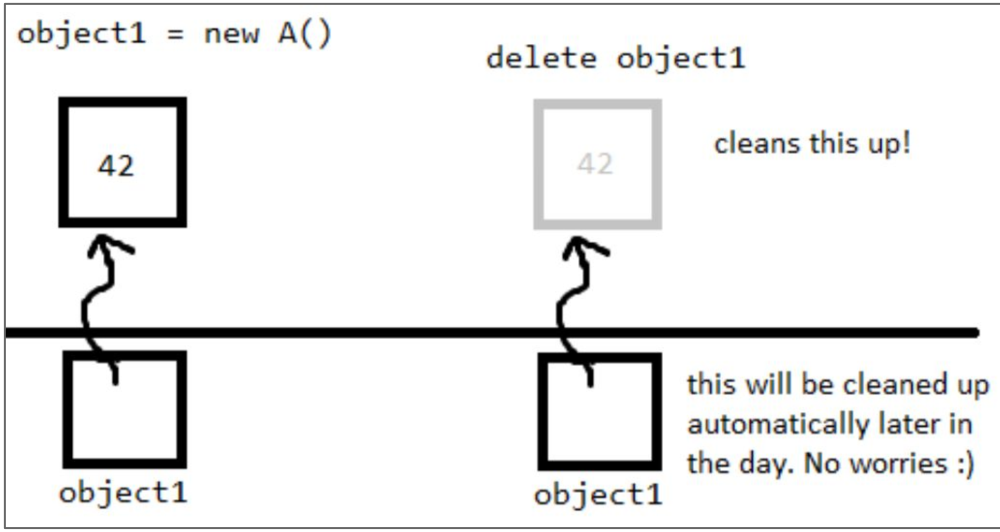
```
int main(){
 S s;
 cout << s.a[0] << ","
      << s.a[1] << endl;
 cout << s.a[2] << endl;
}
```

```
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



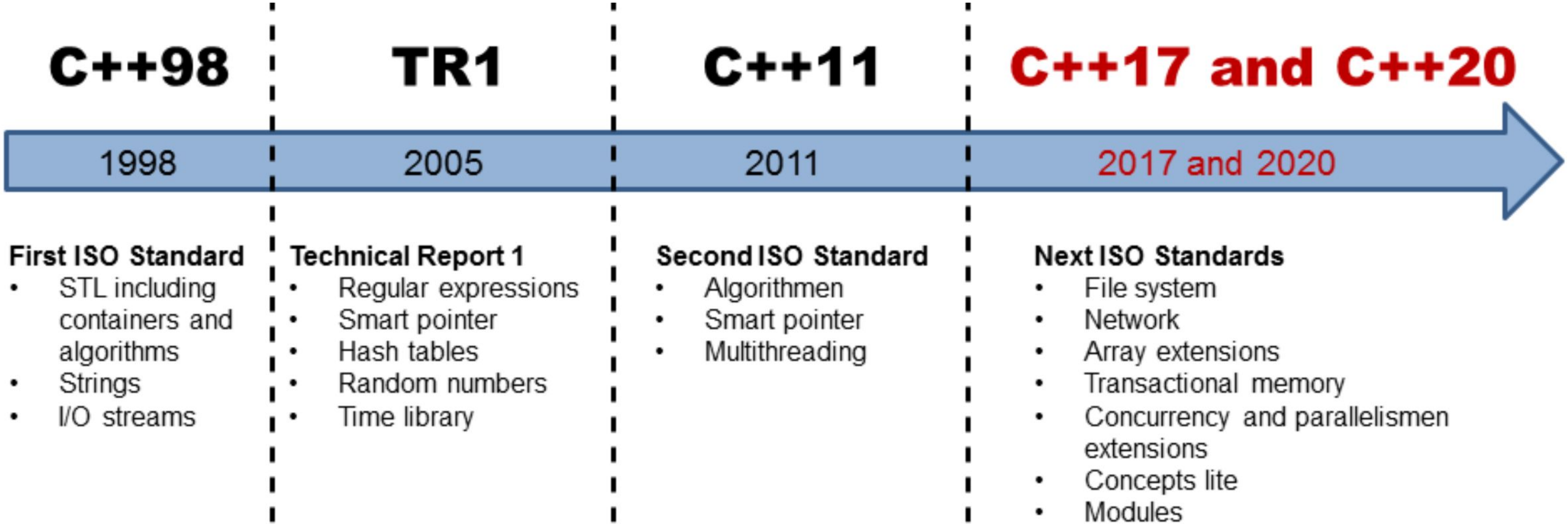

object1 = new A()
delete object1
42
42
cleans this up!
object1
object1
this will be cleaned up automatically later in the day. No worries :)

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++98 — 1998

First ISO Standard
- STL including containers and algorithms
- Strings
- I/O streams

TR1 — 2005

Technical Report 1
- Regular expressions
- Smart pointer
- Hash tables
- Random numbers
- Time library

C++11 — 2011

Second ISO Standard
- Algorithmen
- Smart pointer
- Multithreading

C++17 and C++20 — 2017 and 2020

Next ISO Standards
- File system
- Network
- Array extensions
- Transactional memory
- Concurrency and parallelismen extensions
- Concepts lite
- Modules

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

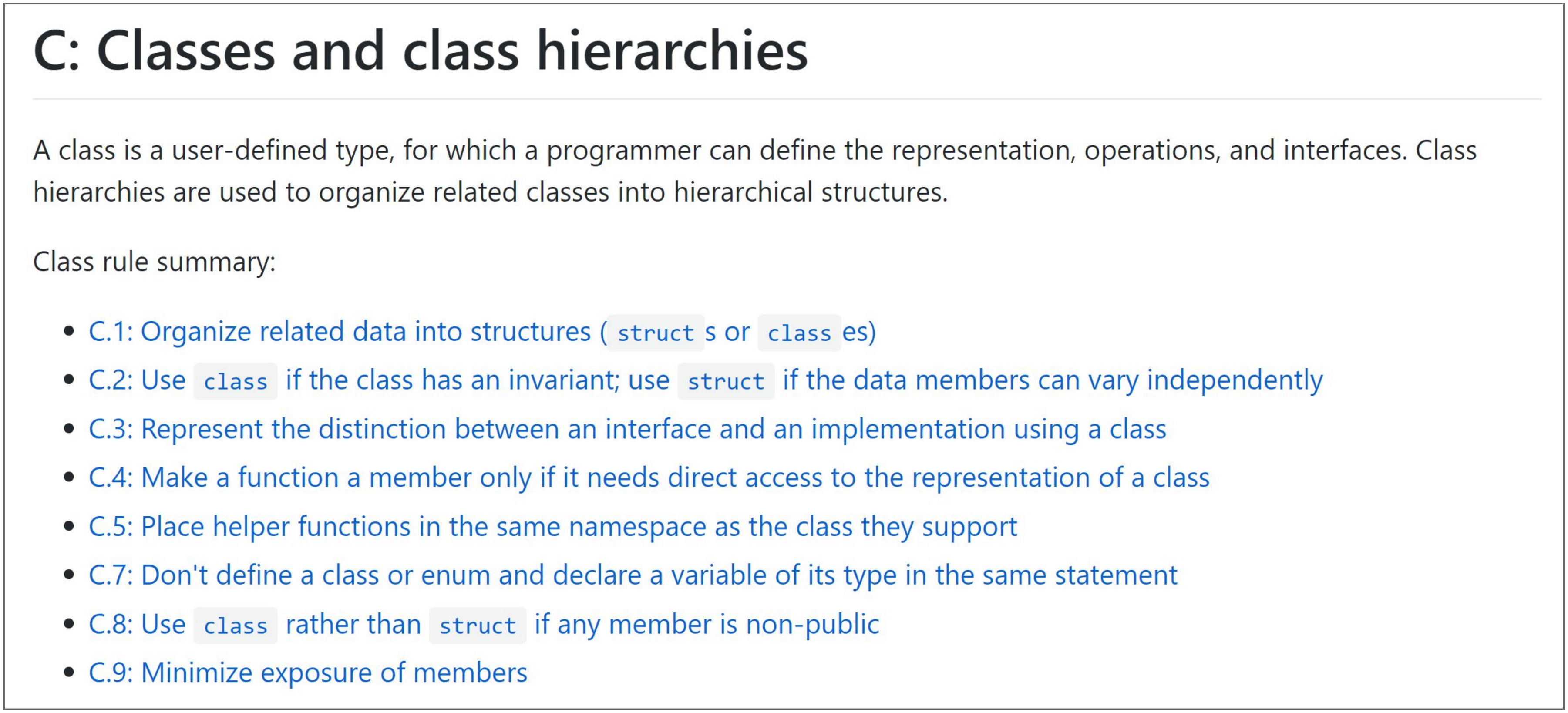
- C++ Core Guidelines : https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-class
- C++ Core Guidelines (Korean Translation) : https://github.com/CppKorea/CppCoreGuidelines/tree/sync/sections
- Official site : https://github.com/isocpp/CppCoreGuidelines

C++ Core Guidelines

- The content in this document itself will **not** be in final exam or lab tests.
    - 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
```

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.
};
```

Fewer member functions (C.4)

- Make function a member only if it needs direct access to the representation of a class. (privates)
  - Fewer functions that can cause trouble by modifying object state.
  - Reduces the number of functions that needs to be modified after a change in representation.

```
class Date {
 // ... relatively small interface ...
};
// helper functions:
Date next_weekday(Date);
bool operator==(Date, Date);
```

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; }
```

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.

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

```cpp
// 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();
}
```

Delegating Constructor (C.51)

- Use delegating constructors to represent common actions for all constructors of a class.
  - To avoid repetition and accidental differences.

```
class Date {
   int d; Month m; int y;
public:
Date(int dd, Month mm, year yy)
      :d{dd}, m{mm}, y{yy}{
   if (!valid(d, m, y))
      throw Bad_date{}; }
Date(int dd, Month mm)
:d{dd},m{mm} y{current_year()}{
   if (!valid(d, m, y))
      throw Bad_date{}; }
};
```

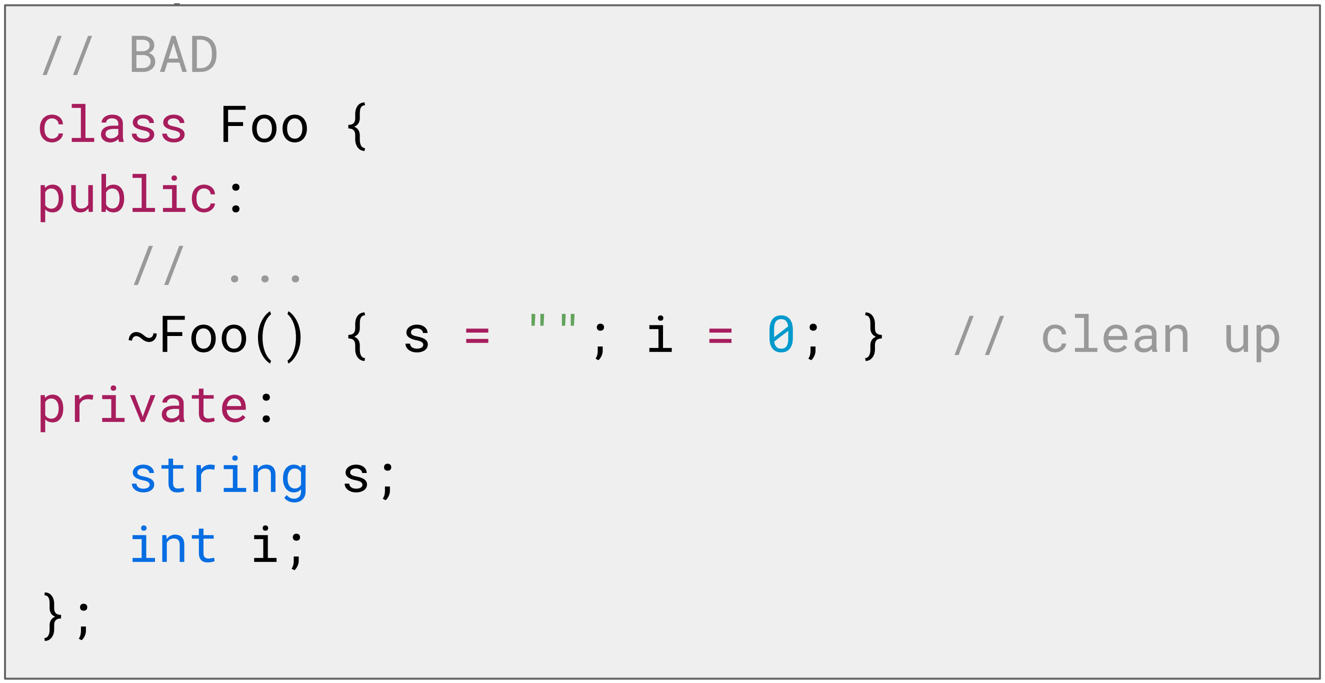
```
class Date2 {
  int d; Month m; int y;
public:
Date2(int dd, Month mm, year yy)
:d{dd}, m{mm}, y{yy}
   { if (!valid(d, m, y))
      throw Bad_date{}; }
Date2(int dd, Month mm)
:Date2{dd, mm, current_year()}{}
};
```

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,



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

Destructors (C.31)

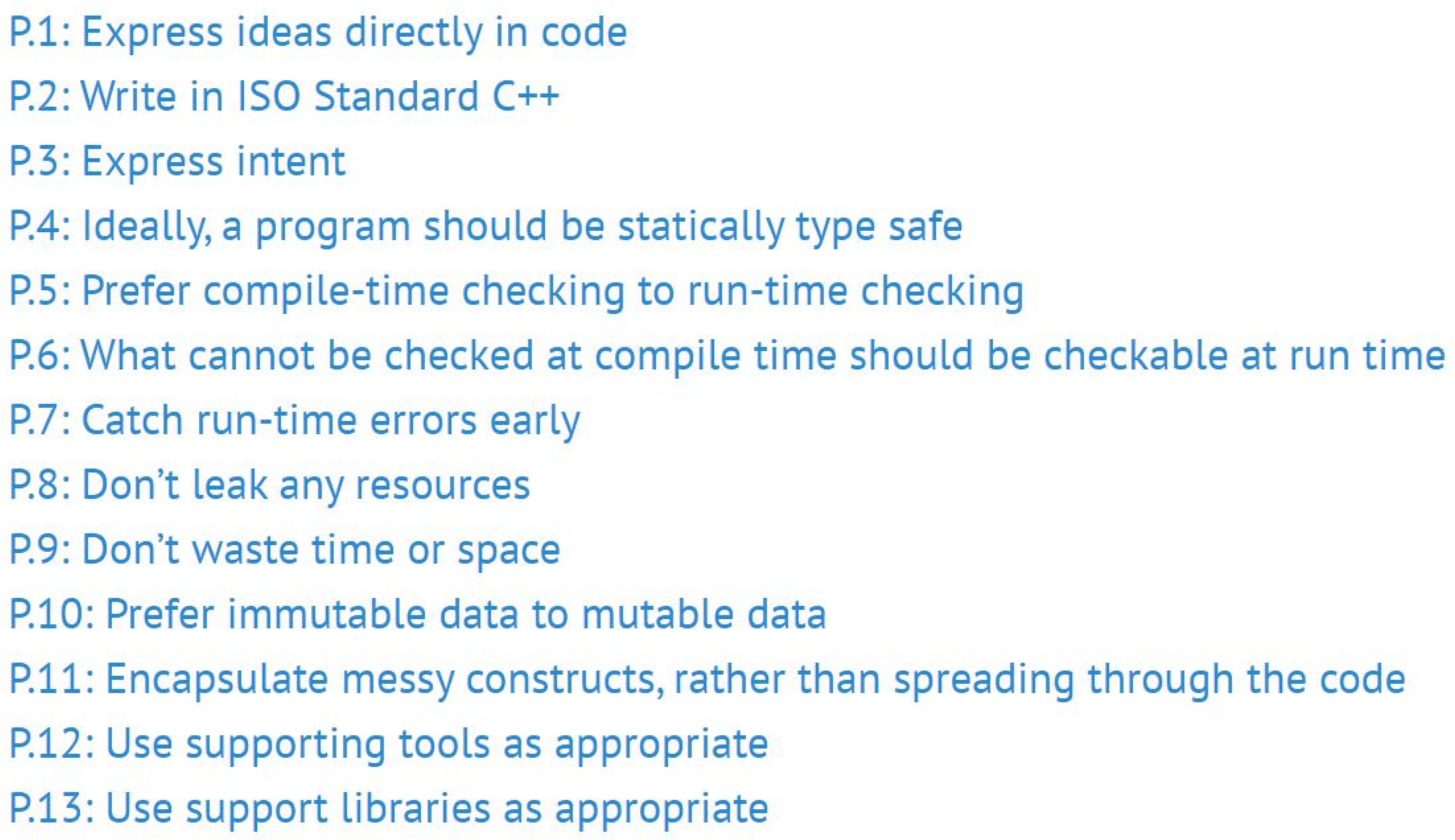
- All resources acquired by a class must be released by the class's destructor.
  - To prevent resource leaks.

```
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.
```

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)