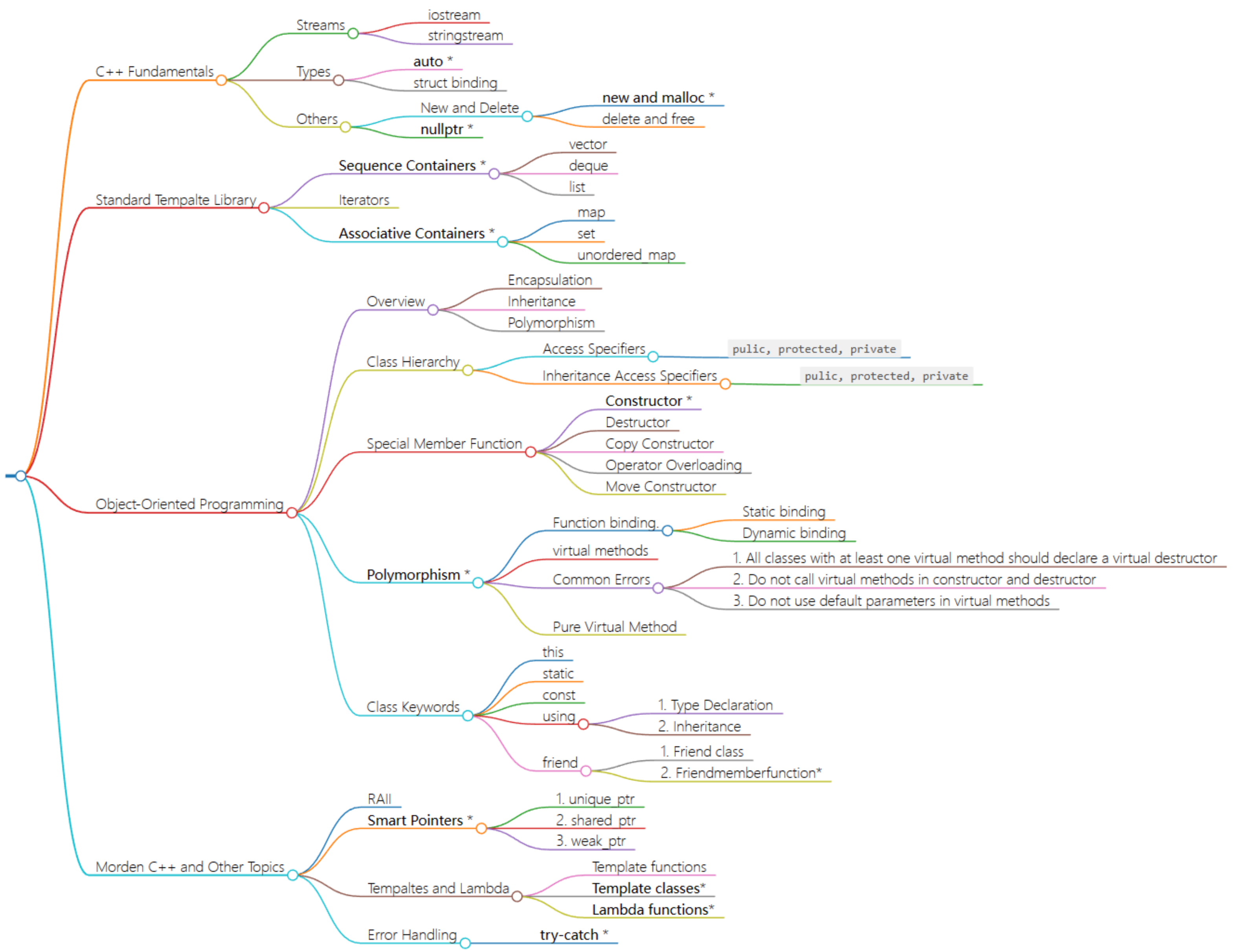
四川大學
SICHUAN UNIVERSITY

Object Oriented Programming—C++
Lecture 12 Final Review

Qijun Zhao
College of Computer Science
Sichuan University

Spring 2023


C++ Fundamentals
Streams
iostream
stringstream
Types
auto *
struct binding
Others
New and Delete
new and malloc *
delete and free
nullptr *
Standard Tempalte Library
Sequence Containers *
vector
deque
list
Iterators
Associative Containers *
map
set
unordered_map
Object-Oriented Programming
Overview
Encapsulation
Inheritance
Polymorphism
Class Hierarchy
Access Specifiers
pulic, protected, private
Inheritance Access Specifiers
pulic, protected, private
Special Member Function
Constructor *
Destructor
Copy Constructor
Operator Overloading
Move Constructor
Polymorphism *
Function binding
Static binding
Dynamic binding
virtual methods
Common Errors
1. All classes with at least one virtual method should declare a virtual destructor
2. Do not call virtual methods in constructor and destructor
3. Do not use default parameters in virtual methods
Pure Virtual Method
Class Keywords
this
static
const
using
1. Type Declaration
2. Inheritance
friend
1. Friend class
2. Friendmemberfunction*
Morden C++ and Other Topics
RAII
Smart Pointers *
1. unique_ptr
2. shared_ptr
3. weak_ptr
Tempaltes and Lambda
Template functions
Template classes*
Lambda functions*
Error Handling
try-catch *

1. C++ Fundamentals and STL

1.1 Stream

Stream: an abstraction for input/output.

Streams convert between *data* and the *string representation of data.*

<iostream>	该文件定义了 **cin**、**cout**、**cerr** 和 **clog** 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。
<fstream>	该文件为用户控制的文件处理声明服务。

Two ways to classify streams

By Direction:

- Input streams: Used for reading data (ex. ‘std::istream’ , ‘std::cin’)
- Output streams: Used for writing data (ex. ‘std::ostream’ , ‘std::cout’)
- Input/Output streams: Used for both reading and writing data (ex. ‘std::iostream’ , ‘std::stringstream’)

By Source or Destination:

- Console streams: Read/write to console (ex. ‘std::cout’ , ‘std::cin’)
- File streams: Read/write to files (ex. ‘std::fstream’ , ‘std::ifstream’ , ‘std::ofstream’)
- String streams: Read/write to strings (ex. ‘std::stringstream’ , ‘std::istringstream’ , ‘std::ostringstream’)

1.2 Types

auto: Keyword used in lieu of type when declaring a variable, tells the compiler to deduce the type.

auto does not mean that the variable doesn’ t have a type. It means that the type is **deduced** by the compiler.

Structured Binding (结构化绑定)

Structured binding lets you **initialize** directly from the contents of a struct

Before

```
auto p =
   std::make_pair("s", 5);
string a = p.first;
int b = p.second;
```

After

```
auto p =
   std::make_pair("s", 5);
auto [a, b] = p;
// a is string, b is int
// auto [a, b] =
      std::make_pair(...);
```

This works for regular structs, too. No nested structured binding.

1.3 STL

C++ STL（标准模板库）是一套功能强大的 C++ 模板类，提供了通用的模板类和函数，这些模板类和函数可以实现多种流行和常用的算法和数据结构，如向量、链表、队列、栈。

容器（Containers）	容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。
算法（Algorithms）	算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。
迭代器（Iterators）	迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。

STL containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!
- **vector, queue, deque, list**

Associative:

- Containers that don’ t necessarily have a sequential order
- More easily searched
- **map, set, unordered_map**

In the STL

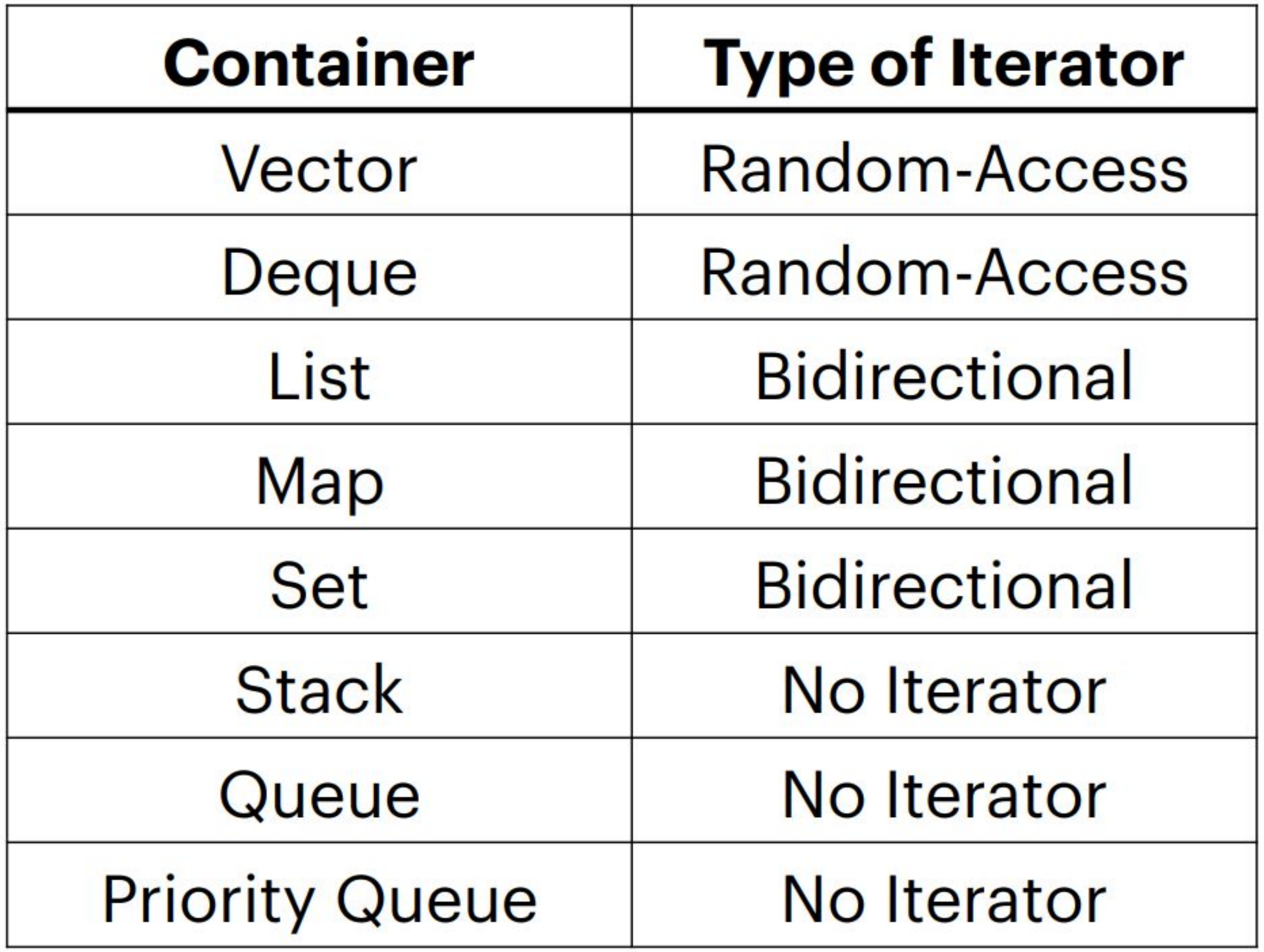
All containers implement iterators, but they’ re not all the same!

- Each container has its own iterator, which can have different behavior.
- All iterators implement a few shared operations:
  - Initializing → `iter = s.begin();`
  - Incrementing → `++iter;`
  - Dereferencing → `*iter;`
  - Comparing → `iter != s.end();`
  - Copying → `new_iter = iter;`

Categorizing STL iterators

Vectors and deques have the most powerful iterators!

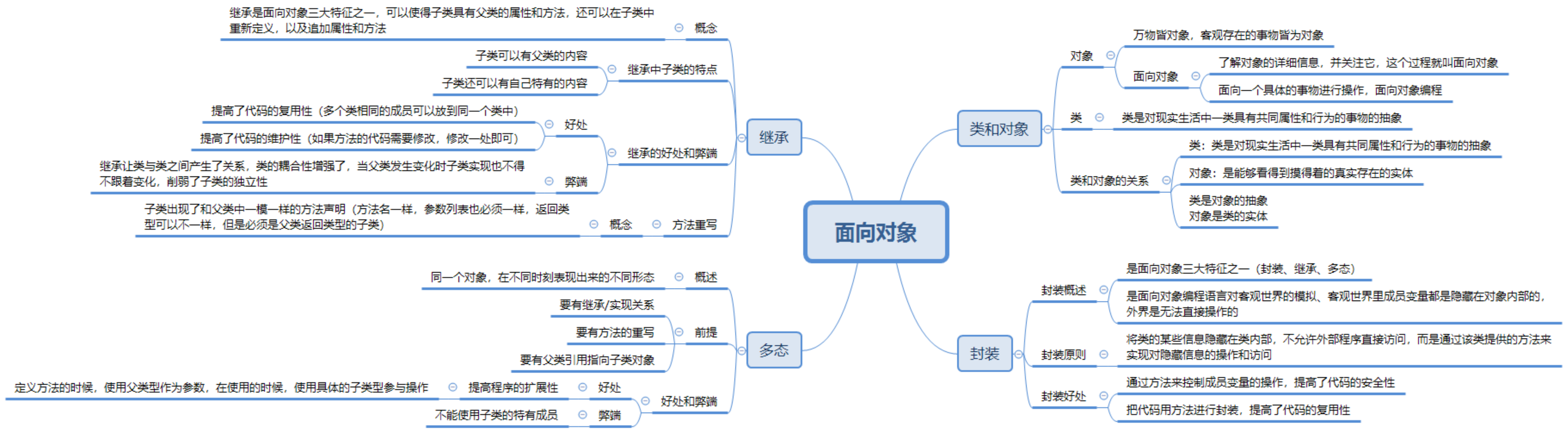
- Creating your own containers means creating their iterators as well.
- You can access elements in stacks and queues one-by-one, but you have to change the container to do so!
- Iteration with iterators is const



Container	Type of Iterator
Vector	Random-Access
Deque	Random-Access
List	Bidirectional
Map	Bidirectional
Set	Bidirectional
Stack	No Iterator
Queue	No Iterator
Priority Queue	No Iterator

New and Delete

- malloc与free是C++/C语言的标准**库函数**，new/delete是C++的**运算符**。它们都可用于申请动态内存和释放内存。
- **对于非内部数据类型的对象而言，光用malloc/free无法满足动态对象的要求**。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于*malloc/free是库函数而不是运算符*，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。
- 因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。 注意new/delete不是库函数。 C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。
- **new可以认为是malloc加构造函数的执行。new出来的指针是直接带类型信息的，而malloc返回的都是void指针。**



- 面向对象
  - 继承
    - 概念
      - 继承是面向对象三大特征之一，可以使得子类具有父类的属性和方法，还可以在子类中重新定义，以及追加属性和方法
    - 继承中子类的特点
      - 子类可以有父类的内容
      - 子类还可以有自己特有的内容
    - 继承的好处和弊端
      - 好处
        - 提高了代码的复用性（多个类相同的成员可以放到同一个类中）
        - 提高了代码的维护性（如果方法的代码需要修改，修改一处即可）
      - 弊端
        - 继承让类与类之间产生了关系，类的耦合性增强了，当父类发生变化时子类实现也不得不跟着变化，削弱了子类的独立性
    - 方法重写
      - 概念
        - 子类出现了和父类中一模一样的方法声明（方法名一样，参数列表也必须一样，返回类型可以不一样，但是必须是父类返回类型的子类）
  - 多态
    - 概述
      - 同一个对象，在不同时刻表现出来的不同形态
    - 前提
      - 要有继承/实现关系
      - 要有方法的重写
      - 要有父类引用指向子类对象
    - 好处和弊端
      - 好处
        - 提高程序的扩展性
          - 定义方法的时候，使用父类型作为参数，在使用的时候，使用具体的子类型参与操作
      - 弊端
        - 不能使用子类的特有成员
  - 类和对象
    - 对象
      - 万物皆对象，客观存在的事物皆为对象
      - 面向对象
        - 了解对象的详细信息，并关注它，这个过程就叫面向对象
        - 面向一个具体的事物进行操作，面向对象编程
    - 类
      - 类是对现实生活中一类具有共同属性和行为的事物的抽象
    - 类和对象的关系
      - 类：类是对现实生活中一类具有共同属性和行为的事物的抽象
      - 对象：是能够看得到摸得着的真实存在的实体
      - 类是对象的抽象
        对象是类的实体
  - 封装
    - 封装概述
      - 是面向对象三大特征之一（封装、继承、多态）
      - 是面向对象编程语言对客观世界的模拟，客观世界里成员变量都是隐藏在对象内部的，外界是无法直接操作的
    - 封装原则
      - 将类的某些信息隐藏在类内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问
    - 封装好处
      - 通过方法来控制成员变量的操作，提高了代码的安全性
      - 把代码用方法进行封装，提高了代码的复用性

2.OOP

2.1 RAII and Smart Pointers

RAII Idiom - Resource Acquisition is Initialization

Holding a resource is a class invariant, and is tied to object lifetime

RAII Idiom consists in three steps:

- Encapsulate a resource into a class (constructor)
- Use the resource via a local instance of the class
- The resource is automatically releases when the object gets out of scope

  (destructor)

Implication 1: C++ programming language does not require the garbage collector!!

Implication 2: The programmer has the responsibility to manage the resources.

Smart Pointers

- 智能指针：
  - 管理动态对象。
  - 行为类似常规指针。
  - 负责**自动释放**所指向的对象。
  - 智能指针也是模板。

C++11 引入了 3 个智能指针类型：
1.std::unique_ptr<T> ： **独占**资源所有权的指针。
2.std::shared_ptr<T> ： **共享**资源所有权的指针。
3.std::weak_ptr<T> ： 共享资源的观察者，需要和 std::shared_ptr 一起使用，不影响资源的生命周期。

2.2 Class Hierarchy

Child/Derived Class or Subclass

A new class that inheriting variables and functions from another class is called a derived or child class

Parent/Base Class

The closest class providing variables and functions of a derived class is called parent or base class

Extending a base class refers to creating a new class which retains characteristics of the base class and on top it can add (and never remove) its own members

2.2.1 Access specifiers (访问说明符)

The **access specifiers** define the visibility of inherited members of the subsequent base class. The keywords public , private , and protected specify the sections of visibility

The goal of the access specifiers is to prevent a direct access to the internal representation of the class for avoiding wrong usage and potential inconsistency (access control)

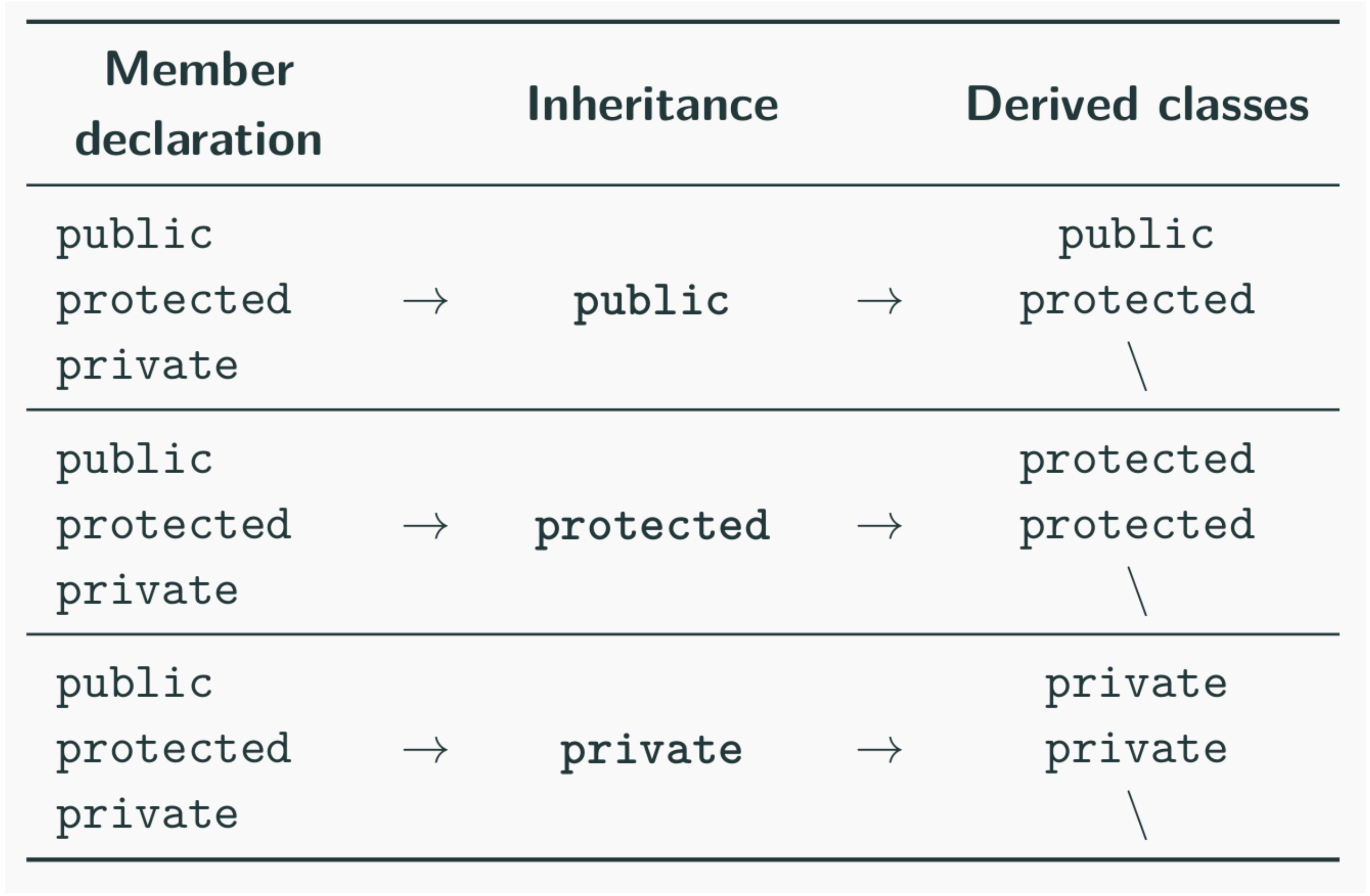
- public: No restriction (function members, derived classes, outside the class)
- protected: Function members and derived classes access
- private: Function members only access (internal)

struct has default public members

class has default private members

2.2.2 Inheritance Access Specifiers

The access specifiers are also used for defining how the visibility is propagated from the base class to a specific derived class in the inheritance



Member declaration		Inheritance		Derived classes
public				public
protected	→	public	→	protected
private				\
public				protected
protected	→	protected	→	protected
private				\
public				private
protected	→	private	→	private
private				\

Class Constructor and Destructor

Constructor [ctor]

A constructor is a special member function of a class that is executed when a new instance of that class is created
Goals: initialization and resource acquisition
Syntax: T(...) same named of the class and no return type

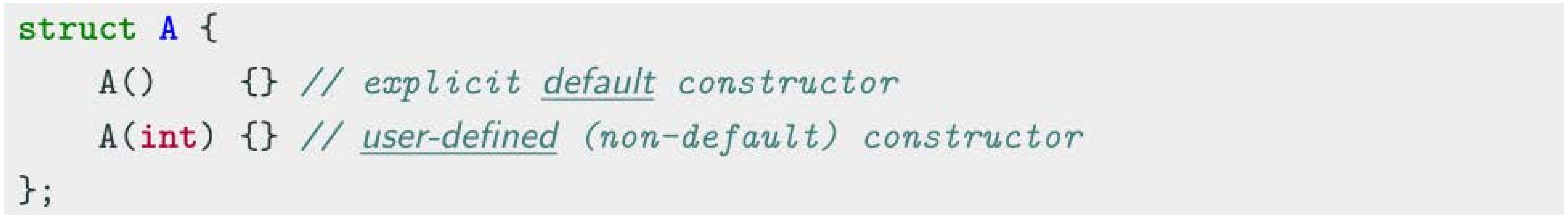
- A constructor is supposed to initialize all data members
- We can define multiple constructors with different signatures
- Any constructor can be constexpr

Default Constructor (默认构造函数)

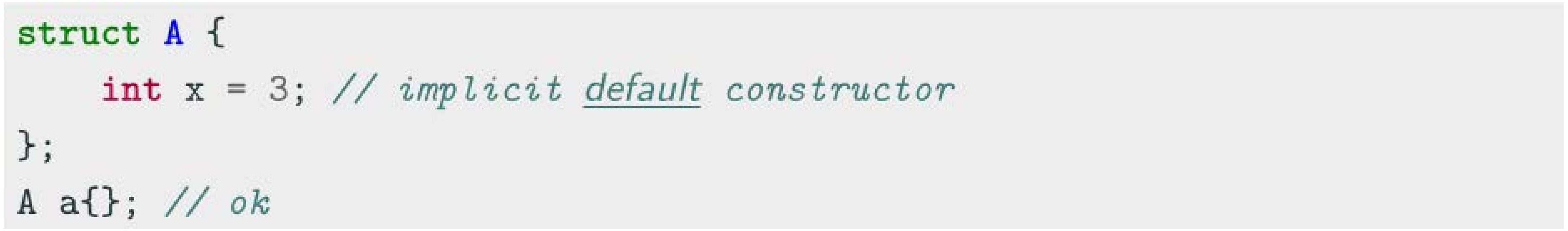
Default Constructor

The default constructor T() is a constructor with <u>no argument</u>

Every class has always either an implicit or explicit default constructor



```cpp
struct A {
    A()    {} // explicit default constructor
    A(int) {} // user-defined (non-default) constructor
};
```



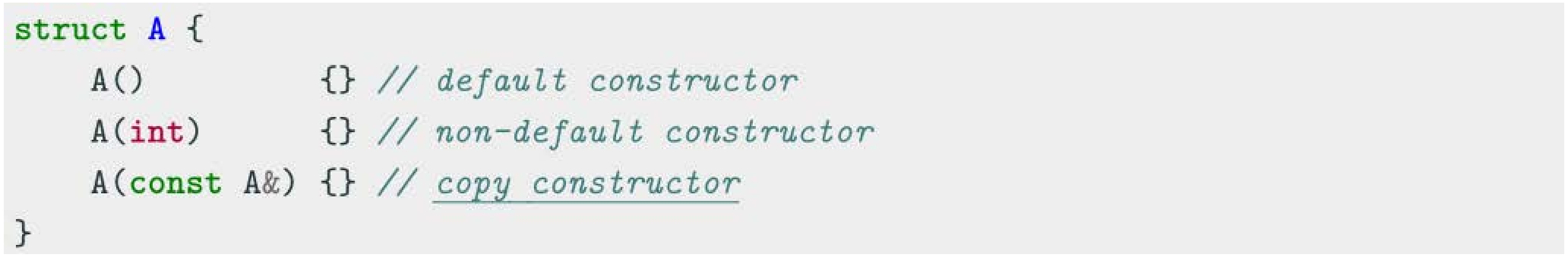
```cpp
struct A {
    int x = 3; // implicit default constructor
};
A a{}; // ok
```

- An implicit default constructor is constexpr

Copy Constructor (拷贝构造函数)

Copy Constructor

A copy constructor T(const T&) creates a new object as a *deep copy* of an existing object



```
struct A {
    A()          {} // default constructor
    A(int)       {} // non-default constructor
    A(const A&)  {} // copy constructor
}
```

- Every class always defines an implicit or explicit copy constructor
- Even the copy constructor implicitly calls the default Base class constructor
- Even the copy constructor is considered a non-default constructor

Operator Overloading (运算符重载)

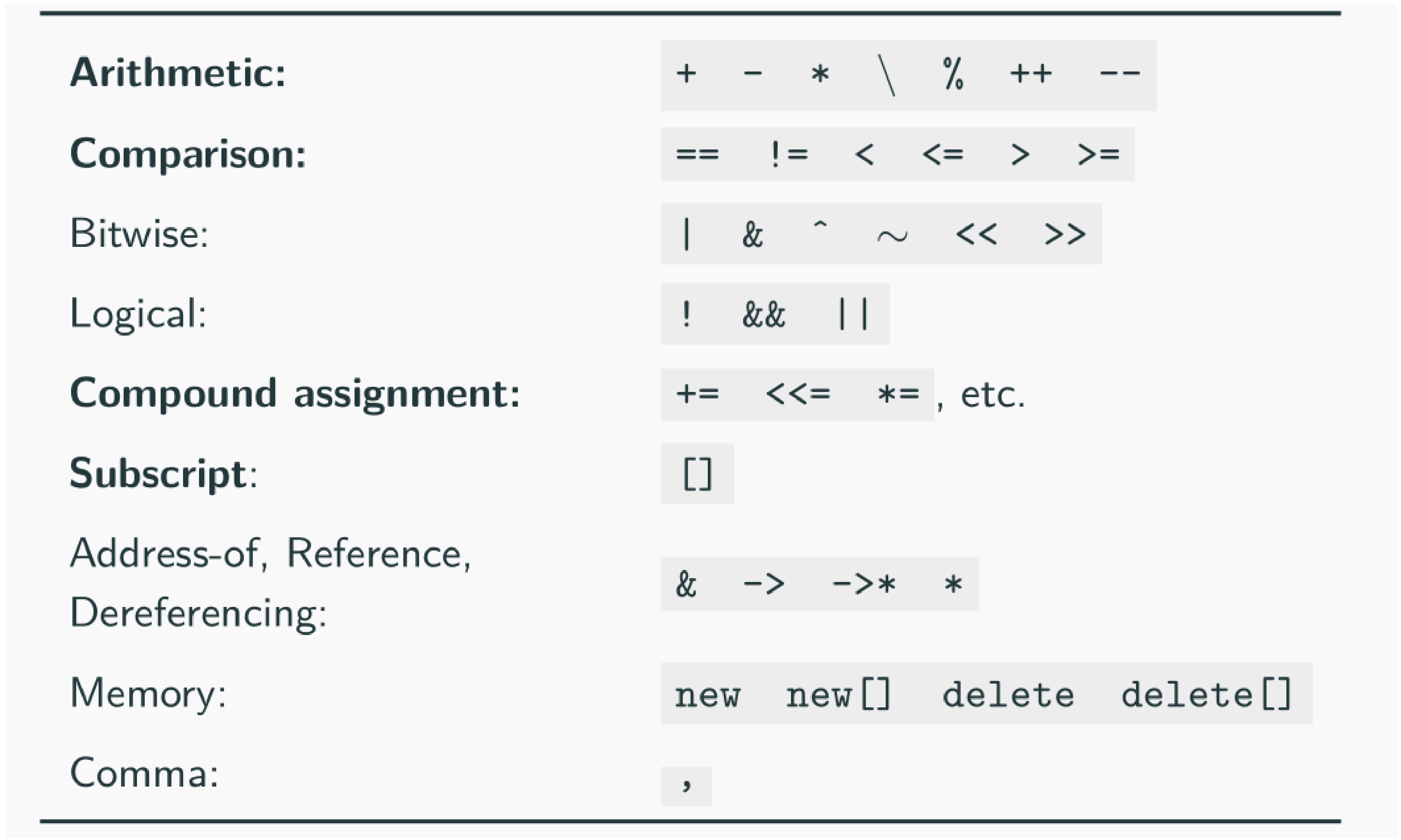
Operator Overloading

Operator overloading is a special case of polymorphism in which some *operators* are treated as polymorphic functions and have different behaviors depending on the type of its arguments

```
struct Point {
    int x, y;

    Point operator+(const Point& p) const {
        return {x + p.x, y + p.x};
    }
};

Point a{1, 2};
Point b{5, 3};
Point c = a + b; // "c" is (6, 5)
```

Operator Overloading

Categories not in bold are rarely used in practice



Arithmetic:	`+` `-` `*` `\` `%` `++` `--`
Comparison:	`==` `!=` `<` `<=` `>` `>=`
Bitwise:	`|` `&` `^` `∼` `<<` `>>`
Logical:	`!` `&&` `||`
Compound assignment:	`+=` `<<=` `*=`, etc.
Subscript:	`[]`
Address-of, Reference, Dereferencing:	`&` `->` `->*` `*`
Memory:	`new` `new[]` `delete` `delete[]`
Comma:	`,`

Operators which cannot be overloaded: `?` `.` `.*` `::` `sizeof` `typeof`

2.4 Polymorphism (多态)

Polymorphism

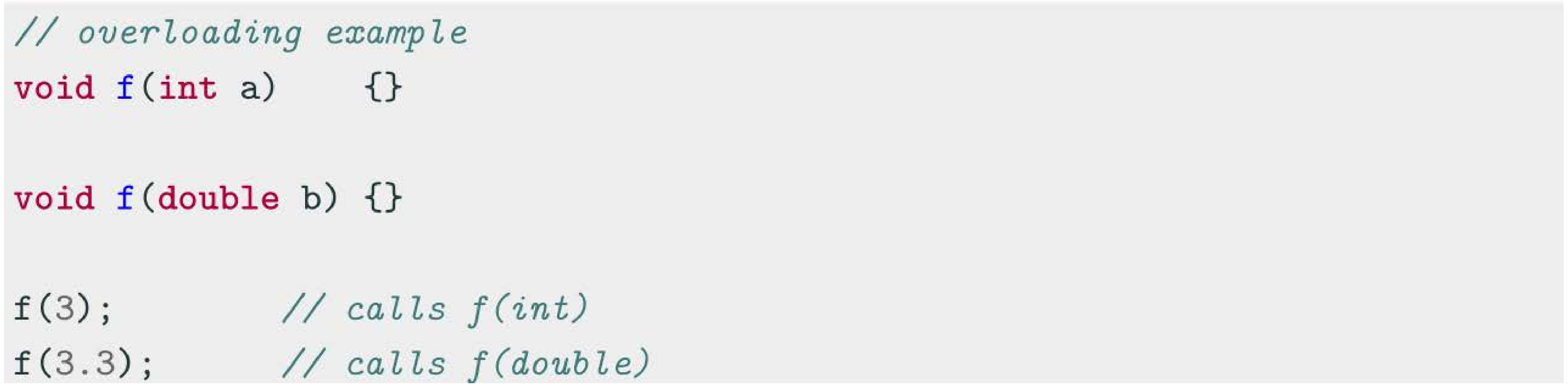
In Object-Oriented Programming (OOP), polymorphism (meaning “having multipleforms”) is the capability of an object of mutating its behavior in accordance with the specific usage *context*

- At run-time, objects of a *base class* behaves as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

Polymorphism vs. Overloading

Overloading is a form of static polymorphism (compile-time polymorphism)

In C++, the term **polymorphic** is strongly associated with dynamic polymorphism (*overriding*)



```
// overloading example
void f(int a)    {}

void f(double b) {}

f(3);          // calls f(int)
f(3.3);        // calls f(double)
```

Function Binding (函数绑定)

Connecting the function call to the function body is called *Binding*

• In **Early Binding** or Static Binding or Compile-time Binding, the compiler identifies the type of object at compile-time

- the program can jump directly to the function address

• In **Late Binding** or Dynamic Binding or Run-time binding, the run-time identifies the type of object at execution-time and then matches the function call with the correct function definition

- the program has to read the address held in the pointer and then jump to that address (less efficient since it involves an extra level of indirection)

C++ achieves **late binding** by declaring a virtual function

Polymorphism - virtual method (虛方法)

```
struct A {
    virtual void f() { cout << "A"; }
}; // now "f()" is virtual, evaluated at run-time

struct B : A {
    void f() { cout << "B"; }
}; // now "B::f()" overrides "A::f()", evaluated at run-time

void g(A& a) { a.f(); } // accepts A and B

A a;
B b;
g(a); // print "A"
g(b); // NOW, print "B"!!!
```

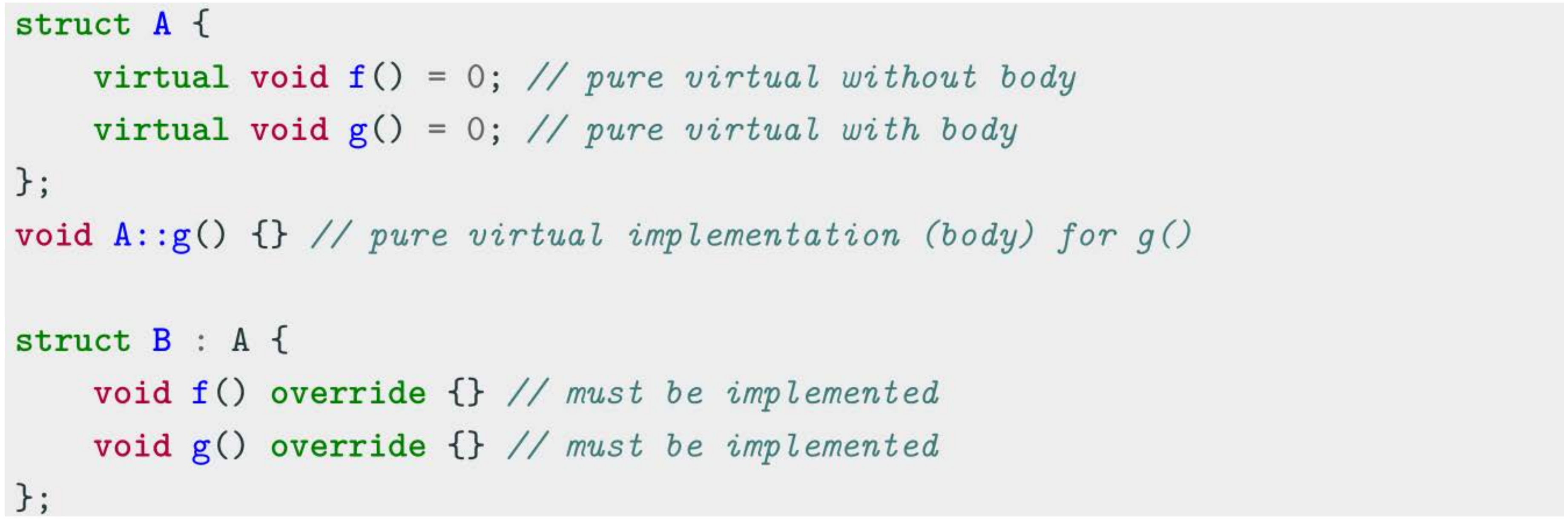
The virtual keyword is not *necessary* in derived classes, but it improves *readability* and clearly advertises the fact to the user that the function is virtual

Pure Virtual Method (纯虚方法)

Pure Virtual Method

A **pure virtual method** is a function that <u>must</u> be implemented in derived classes (concrete implementation)

Pure virtual functions can <u>have</u> or <u>not have</u> a body



```cpp
struct A {
    virtual void f() = 0; // pure virtual without body
    virtual void g() = 0; // pure virtual with body
};
void A::g() {} // pure virtual implementation (body) for g()

struct B : A {
    void f() override {} // must be implemented
    void g() override {} // must be implemented
};
```

2.5 Class Keywords (类关键字)

this

Every object has access to its own address through the const pointer this

Explicit usage is not mandatory (and not suggested)

this is necessary when:

- The name of a local variable is equal to some member name
- Return reference to the calling object

```
struct A {
    int x;
    void f(int x) {
        this->x = x; // without "this" has no effect
    }
    const A& g() {
        return *this;
    }
};
```

Class Keywords——static

static (静态)

The keyword static declares members (fields or methods) that are not bound to class instances. A static member is shared by all objects of the class

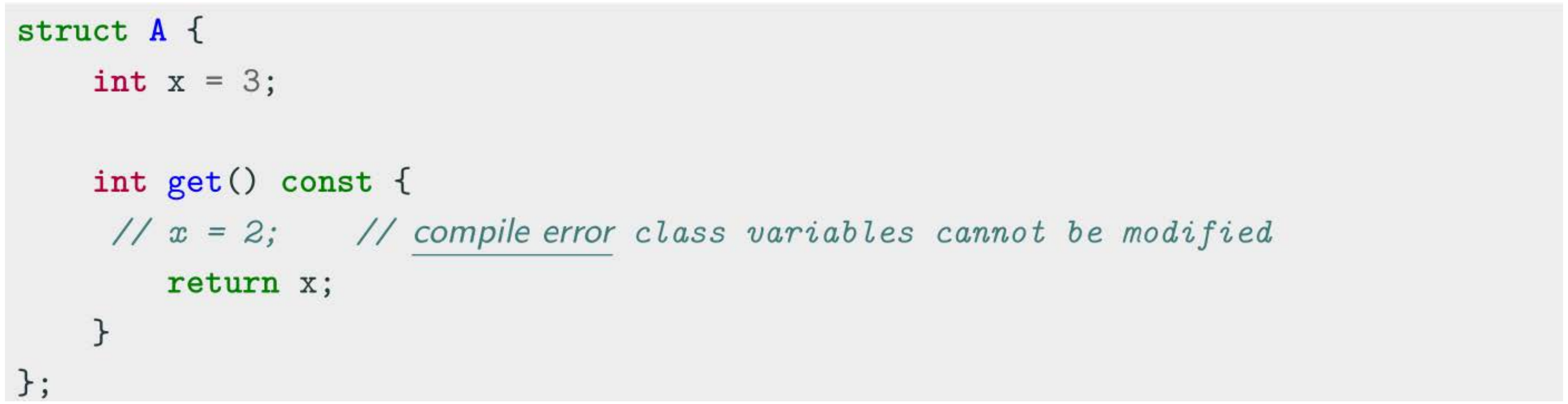
- A static member function can only access static class members
- A non- static member function can access static class members
- Non-const static data members cannot be directly initialized inline ...before C++17

Class Keywords——const

Const member functions (常量成员函数)

Const member functions (inspectors or observer) are functions marked with const that are not allowed to change the object state

Member functions without a const suffix are called non-const member functions or **mutators**. The compiler prevents from inadvertently mutating/changing the data members of observer functions



```
struct A {
    int x = 3;

    int get() const {
     // x = 2;     // compile error class variables cannot be modified
        return x;
    }
};
```

Class Keywords——friend (友元)

friend Class

A friend class can access the private and protected members of the class in which it is declared as a friend

Friendship properties:

- **Not Symmetric**: if class A is a friend of class B, class B is not automatically a friend of class A
- **Not Transitive**: if class A is a friend of class B, and class B is a friend of class C, class A is not automatically a friend of class C
- **Not Inherited**: if class Base is a friend of class X, subclass Derived is not automatically a friend of class X; and if class X is a friend of class Base, class X is not automatically a friend of subclass Derived

Class Keywords——friend

friend Method

A ***non-member*** ***function*** can access the private and protected members of a class if it is declared a friend of that class

```
class A {
    int x = 3;  // private

    friend int f(A a); // friendship declaration, no implementation
};

//'f' is not a member function of any class
int f(A a) {
    return a.x;  // A is friend of f(A)
}
```

friend methods are commonly used for implementing the stream operator operator<<

2.6 Templates and Lambda

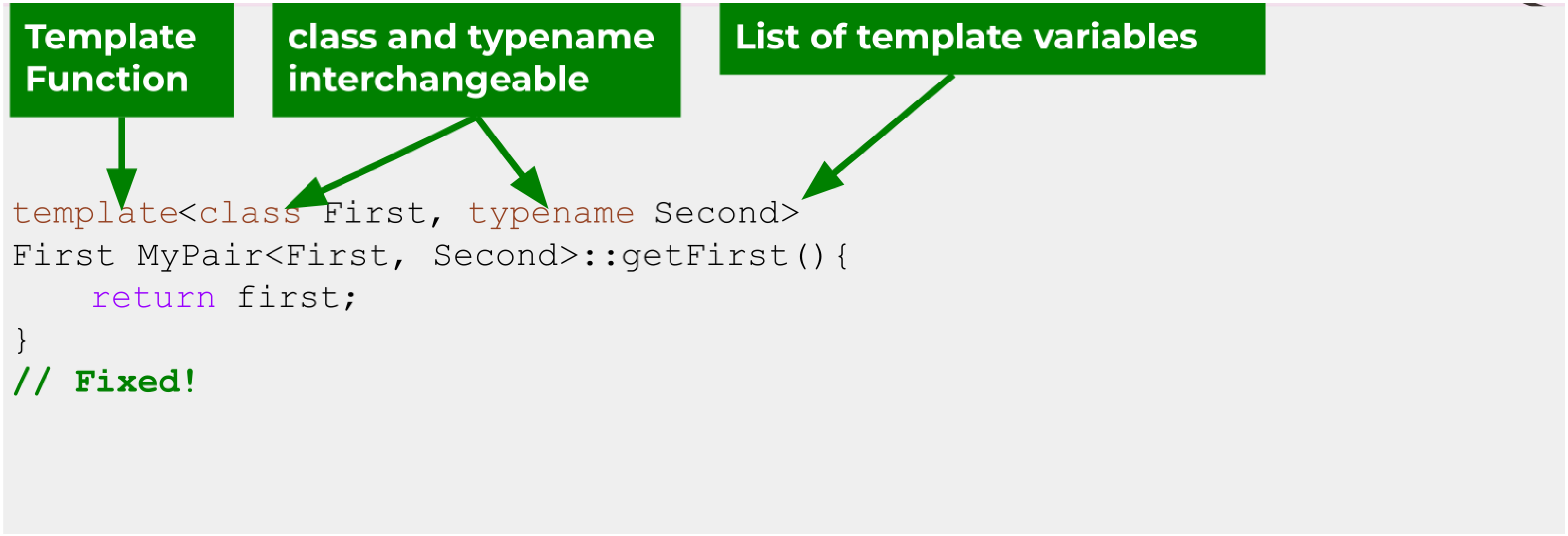
Template Class: A class that is parametrized over some number of types. A class that is comprised of member variables of a general type/types.

Writing a Template Class: Syntax

```
//mypair.h
template<typename First, typename Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

        void setFirst(First f);
        void setSecond(Second f);
    private:
        First first;
        Second second;
};
```

Use generic typenames as placeholders!

Implementing a Template Class: Syntax


Template Function
class and typename interchangeable
List of template variables

```
template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
// Fixed!
```

Lambdas Function

Lambdas are **inline**, **anonymous** functions that can know about functions declared in their same scope!

Outside parameters go here

Specifies that Type is generic

```
auto var = [capture-clause] (auto param) -> bool
{
        ...
}
```

Function body goes here!

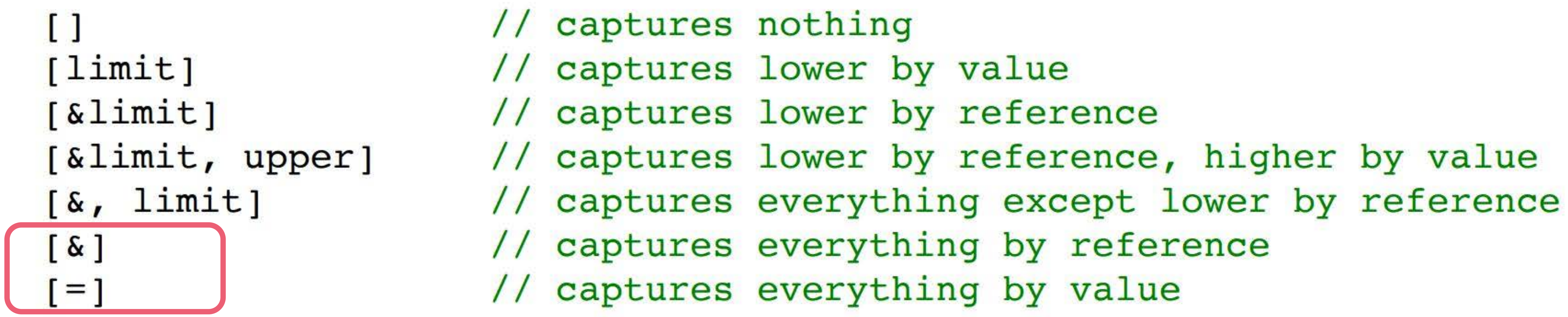
```
int limit = 5;
auto isMoreThan = [limit] (int n) { return n > limit; };
isMoreThan(6); // true
```

Capture Clauses

You can capture any outside variable, both by reference and by value.

- Use just the = symbol to capture everything by value,

and just the & symbol to capture everything by reference



```
[]                  // captures nothing
[limit]             // captures lower by value
[&limit]            // captures lower by reference
[&limit, upper]     // captures lower by reference, higher by value
[&, limit]          // captures everything except lower by reference
[&]                 // captures everything by reference
[=]                 // captures everything by value
```

Using Lambdas

Lambdas are pretty computationally cheap and a great tool!

- Use a lambda when you need a *short* function or to *access local variables* in your function.
- If you need more logic or overloading, use function pointers.

C++ Exception Basics

C++ provides three keywords for exception handling:

throw Throws an exception

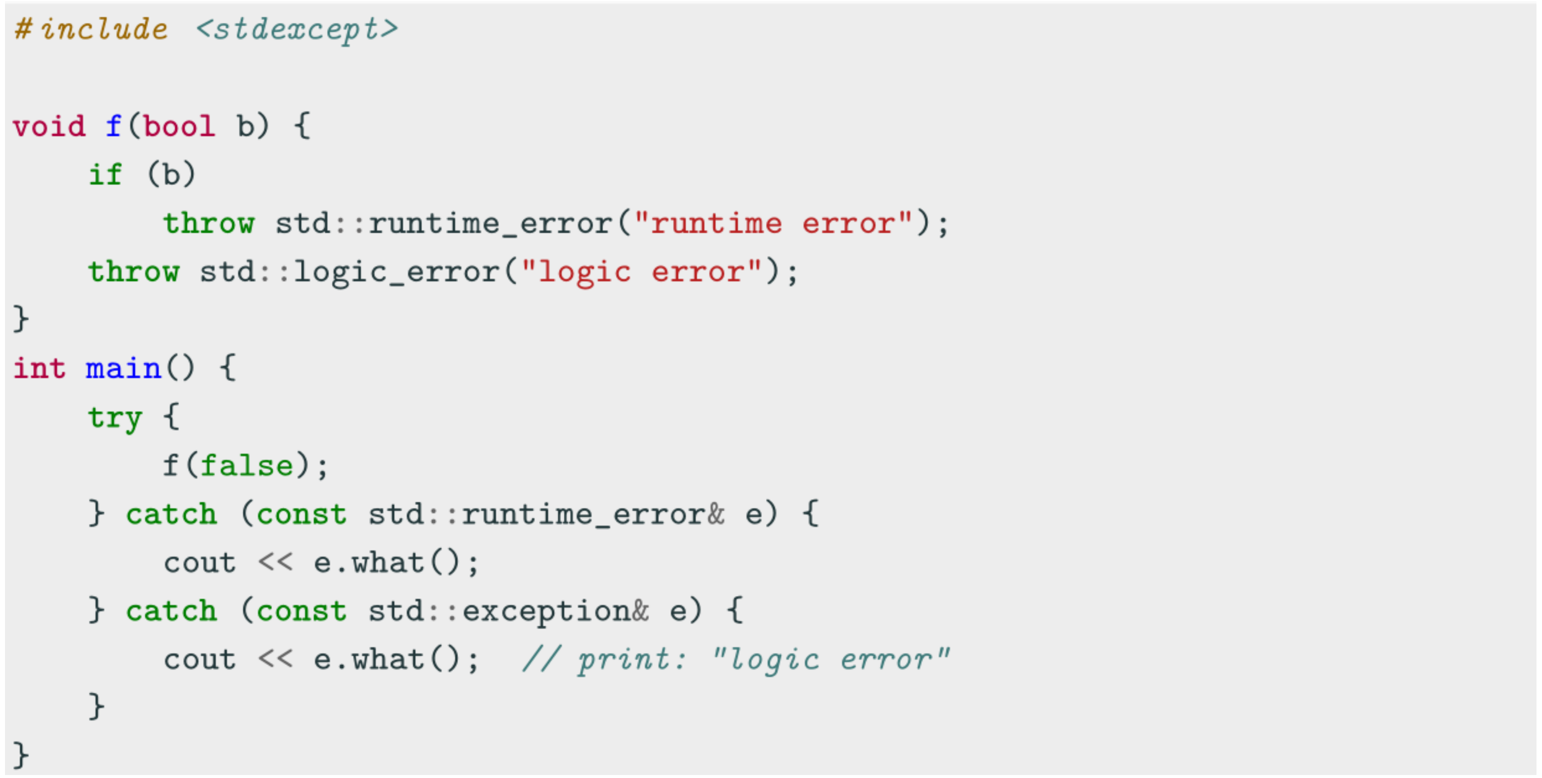
try Code block containing potential throwing expressions

catch Code block for handling the exception

```
void f() { throw 3; }

int main() {
    try {
        f();
    } catch (int x) {
        cout << x; // print "3"
    }
}
```

std Exceptions

throw can throw everything such as integers, pointers, objects, etc. The standard way consists in using the std library exceptions <stdexcept>



```cpp
#include <stdexcept>

void f(bool b) {
    if (b)
        throw std::runtime_error("runtime error");
    throw std::logic_error("logic error");
}
int main() {
    try {
        f(false);
    } catch (const std::runtime_error& e) {
        cout << e.what();
    } catch (const std::exception& e) {
        cout << e.what();  // print: "logic error"
    }
}
```

Exception Propagation

Exceptions are automatically propagated along the call stack. The user can also control how they are propagated

```
int main() {
    try {
        ...
    } catch (const std::runtime_error& e) {
        throw e; // propagate a copy of the exception
    } catch (const std::exception& e) {
        throw;   // propagate the exception
    }
}
```

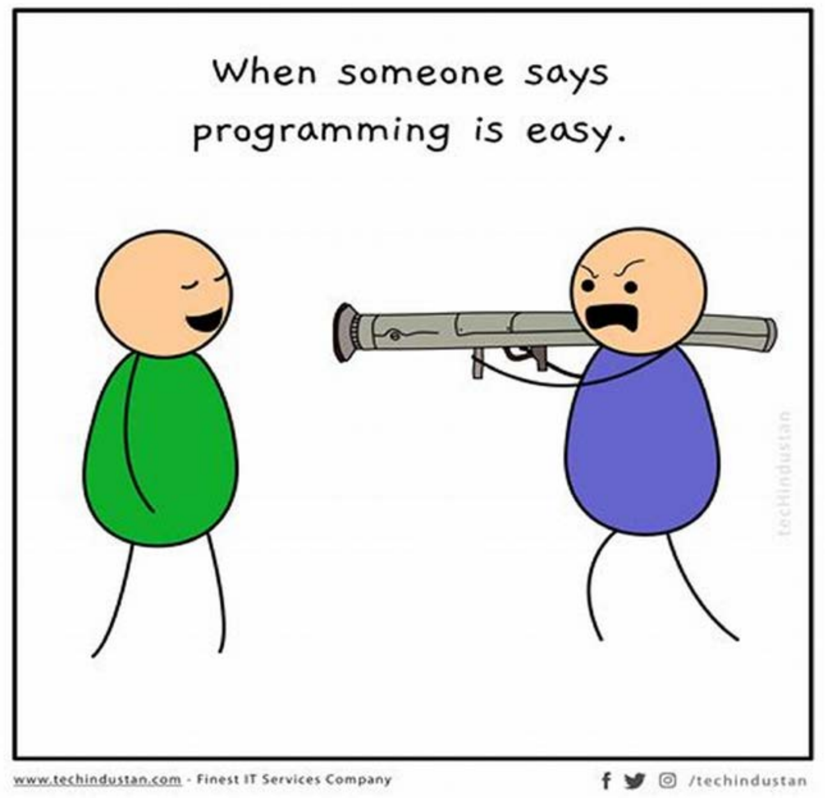
Defining Custom Exceptions

```
#include <exception> // to not confuse with <stdexcept>

struct MyException : public std::exception {
    const char* what() const noexcept override { // could be also "constexpr"
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch (const std::exception& e) {
        cout << e.what(); // print "C++ Exception"
    }
}
```

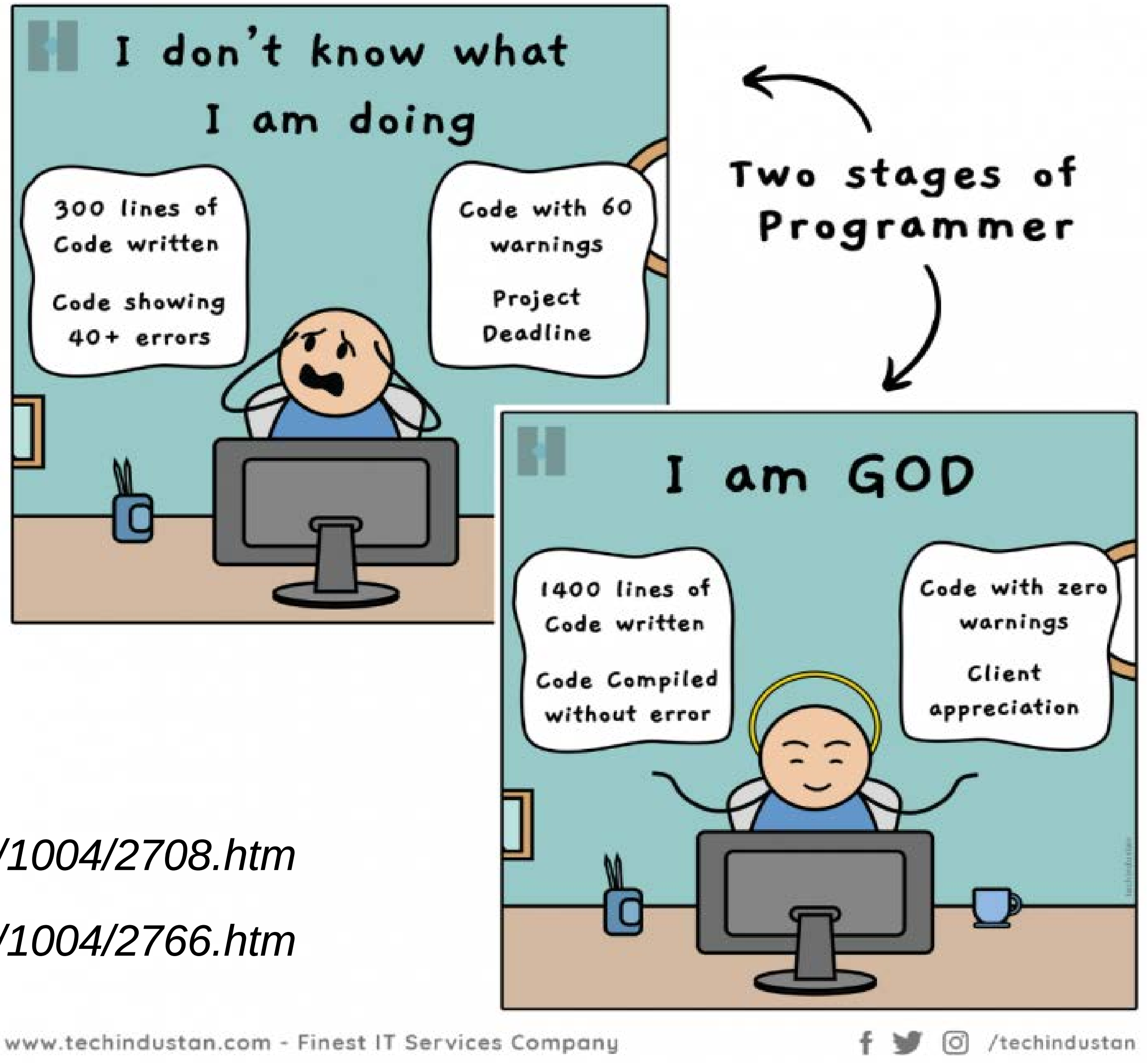
期末考试

- 时间：第17周星期二（6月13日）下午14:00—16:00
- 地点：江安综合楼B座B504
- 形式：闭卷（中文）

- 题型与分值：
  1. 单选题，10道、20分
  2. 填空题，5道、10分
  3. 程序阅读题，8道、40分
  4. 程序填空题，10小问，20分
  5. Lambda函数设计题，2道，10分


When someone says programming is easy.
www.techindustan.com - Finest IT Services Company
/techindustan

谢谢你们一学期的陪伴和共同进步，祝你们早日成为God Progammer!


I don't know what I am doing
300 lines of Code written
Code showing 40+ errors
Code with 60 warnings
Project Deadline
Two stages of Programmer
I am GOD
1400 lines of Code written
Code Compiled without error
Code with zero warnings
Client appreciation
www.techindustan.com - Finest IT Services Company
/techindustan

https://syl.scu.edu.cn/info/1004/2708.htm

https://syl.scu.edu.cn/info/1004/2766.htm

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn