g++ -c test.cpp -std=c++11

g++ -c main.cpp -std=c++11

g++ test.o main.o -o main -std=c++11

gcc -Wall -Werror -std=c++11 -c main.c

bazel build //assignments/ev:client

bazel run //assignments/wl:main

./bazel-bin/assignments/ev/client | diff ./assignments/ev/client.sampleout -

～～WEEK1 Intro & types

\*\*types:

enum:来自有限，常数集的单个选项

\*\*Program errors

During the course we will talk about different types of errors:

Compile time:no type 因为编译时候最看重类型匹配

Runtime (Exception)： 出界，等等

(Runtime) Undefined behaviour： 越界访问a[0]没有定义这个行为是什么

Link time：在连接函数（文件）时候出现问题

Runtime (Logic)：人为错误类似，由于逻辑错误在运行时不对

\*\*Literals：一些文字直接嵌入到机器代码指令中。其他文字作为编译代码的一部分存储在只读数据中

有Boolean，Character，Integer，Floating-point ，String (these are not std::strings) eg："Healthy Harold", "a"

\*\*Declarations vs Definitions:

声明会告知变量的类型和名称

定义是一个声明，但也做了额外的事情:

变量定义为存储器分配存储，并构造变量。

类定义允许您创建类'类型的变量。

您可以仅使用声明调用函数，但必须在以后提供定义

所有内容必须具有一个定义

\*\* const:

Make everything const unless you know it will be modified

This course will have a heavy focus on const-correctness

Why const:

1.更清晰的代码您可以知道一个函数不会尝试和修改只是签名的东西，不可变对象更容易推理

2.编译器可能能够进行某些优化

3.不可变对象在多线程中更容易使用不必担心线程之间的竞争条件

4.防止错误改变你不应该做的事情

\*\*References: 一个变量的别名，两个说的是一个意思

References to const：对const的引用意味着您无法使用引用修改对象，该对象仍然可以修改，而不是通过此引用

\*\*auto:

自动扣除：取右边的类型，但剥离顶层const和＆。

让编译器知道类型。

\*\*Functions: Pass by value：复制值，之后函数完成后 销毁

\*\*Functions: Pass by reference： 传入左值，可以对原始修改

\*\*Lvalue and Rvalue：

Lvalue: lvalue,xvalue(基本在生命周期结束前后，所以他的资源有可能被移交)

Rvalue: xvalue(&& 临时给没有索引（变量）地址的值一个地址)

prvalue(std::move后的右值一般是prvalue)

按值调用：传递实际参数的rvalue

无法访问/修改被调用者中的实际参数

通过引用调用：

传递实际参数的左值可以直接访问/修改实际参数消除传递大对象的开销

\*\* reference

int& fin(int& a){

++a;

int& b = a;

return b;

}

int main() {

int c=0;

int& d{fin(c)}; //初始化后不改变c和d代表一个 c穿进去是reference 就是本身

std::cout<<c<<'\n';

int ee = 2;

d=ee; //改变的是d值也就是c值

std::cout<<c<<'\n';

return 0;

}

\*\*Function Overloading：

相同名字但是不同实现：可以按照传入形式进行选择

Step 1: Find candidate functions: Same name

Step 2: Select viable ones: Same number arguments + each argument

convertible

Step 3: Find a best-match: Type much better in at least one argument

在按值调用时，顶级const对传递给函数的对象没有影响。具有顶级const的参数与没有顶级const的参数无法区分

\*\*顶层和底层：

#include <iostream>

#include <cstdlib>

int main(){

// int i = 2;

// int j = 1;

// int& k = i;

// k = j; // This does not make k reference j instead of i. It just changes the value

// std::cout << "i = " << i << ", j = " << j << ", k = " << k << '\n';

int i = 1;

int& ref = i;

std::cout << ref << '\n';

i++; // This is fine

std::cout << ref << '\n';

ref++; // This is not

std::cout << ref << '\n';

std::cout << i << '\n';

const int j = 1;

const int& jref = j; // this is allowe

// int& ref = j; // not allowed

std::cout << jref << '\n';

int a = 42;

int a1 = 89;

const int\* p1 = &a;

std::cout << \*p1 << std::endl;

// (\*p1) = 8; 不正确是因为\*P是一个整体看，这时候是常量，不能修改指针指向的位置的值进而改变原始值（a=42）底层；

p1 = &a1;

std::cout << \*p1 << std::endl;

int b = 42;

int\* const q1 = &b;

std::cout << \*q1 << std::endl;

(\*q1) = 8;

//q1 = &a1; // 因为q1是const,所以首先q1本身的值是不能变的也就是不能重新赋予新的值，

//但是\*q1 这个指针不受限制是可以变的。也就是改变了b得值 这个是顶层常量 普通reference 是顶层因为不可以改变自己的指向，而const & 都是因为既不能改变自己的引用，又不能改表引用对象的值

std::cout << \*q1 << std::endl;

std::cout << b << std::endl;

const int\*const p3= &b; //这个右边是顶层，因为是常量指针也就是本身值不能改变（地址），

//而左边是底层的，也就是地址可以改，但是不能通过该指针指向的值进而改指向变量的值 类似于 int& i

// // Top-level const ignored

// Record Lookup(Phone p);

// Record Lookup(const Phone p); // redefinition

// // Low-level const not ignored

// Record Lookup(Phone &p); (1)

// Record Lookup(const Phone &p); (2)

// Phone p;

// const Phone q;

// Lookup(p); // (1)

// Lookup(q); // (2)

//这里面首先由于函数对于普通的形参，调用参数是复制一个实际参数样本。而对于引用形参，是直接用实际参数的各种数据（包含地址），

//所以调用过程不一样，导致了不可被忽略当重载时，因为const代表传入进来一定要是const，但是普通的代表我只复制一个值，之后改不改变

//是函数内部问题。

}

\*\*Constexpr：以在编译时计算的变量如果在编译时已知其输入，则可以在编译时运行该函数。一行代码

好处：可以在编译时确定的值意味着在运行时需要更少的处理，

从而导致整体更快的程序执行将潜在的错误源转移到编译时而不是运行时（更容易调试）

\*\*bazel:

Each build rule has:

A name

A list of sources (srcs)

A list of headers (hdrs)

A list of dependencies (dep)

Potentially many other arguments

cc\_library(

name = "hello\_world",

srcs = ["hello\_world.cpp"],

hdrs = ["hello\_world.h"],

deps = []

)

cc\_library(

name = "printer",

srcs = ["printer.cpp"]

hdrs = ["printer.h"],

deps = [

# If it's declared within the same build

# file, we can skip the directory

":hello\_world"

]

)

cc\_binary(

name = "main"

srcs = ["main.cpp"],

deps = [

"//path/to:hello\_world"

]

)

cc\_library: (编写文件)

A piece of code that can't run on its own, but can be depended upon by

other files

cc\_binary: (main)

The srcs should have a main function

Has no headers

Cannot be tested

cc\_test:

Works very similar to a binary

Semantic difference

\*\*迭代器例子：

const std::vector<int> vec;

std::list<int> li;

std::forward\_list<double> forward\_li;

std::string s;

vec.begin(); // const random access iterator

vec.cbegin(); // const random access iterator

(\*vec.begin())++; // Fails to compile (vector is const, attempt to access non-const iterator)

li.cbegin(); // const bidirectional iterator

li.rbegin(); // non-const bidirectional iterator

forward\_li.cbegin(); // const forward iterator

(\*forward\_li.begin())++; // forward iterator

(\*forward\_li.cbegin())++; // fails to compile (attempt to modify non-const iterator)

forward\_li.crbegin(); // Fails to compile (cannot get reverse iterator for forward iterators)

s.begin(); // non-const random access iterator

std::back\_inserter(vec); // Compiles. Can't invoke the dereference operator to insert anything because you get a compile time error.

std::back\_inserter(li); // Output iterator

std::istream\_iterator<int>(std::cin); // Input iterator

std::ostream\_iterator<int>(std::cout, " "); // Output iterator

~~WEEK 2.1 STL Containers & I tainers & Iterators

cast: std::static\_cast<int>(a)

为什么STL好： 优化，API，template，等等

\*\*iterator:迭代器是指针的抽象概念迭代器是将容器数据抽象为对象序列的类型容器

和算法之间的粘合剂

\*\*Iterator Categories:

output input forward bidirectional randonm access

read a=\*p =\*p =\*p =\*p

access -> -> -> ->[]

write \*p=2 \*p \*p \*p

iteration ++ ++ ++ ++,-- ++,--,+=,-=

compare ==,!= ==,!= ==,!= ==,!=,<,>,<=,>=

\*\*Sequential Containers:

std::vector: Dynamically sized array

std::list: Doubly linked list

std::deque :<vector> with fast operations for element at beginning

\*\*数据结构操作：

1.vector：std::vector<int> a{1,2}; //初始化

a.push\_back(3); //

a.pop\_back();//

a.at(3),a[3];//

a.front()，a.back();

a.erase;//

a.insert();a.emplace();//

a.clear；//

2.pari: pair<T1, T2> p1; //创建一个空的pair对象（使用默认构造），它的两个元素分别是T1和T2类型，采用值初始化。

pair<T1, T2> p1(v1, v2); //创建一个pair对象，它的两个元素分别是T1和T2类型，其中first成员初始化为v1，second成员初始化为v2。

p1.first; // 返回对象p1中名为first的公有数据成员

p1.second;

pair<int, double> p1(1, 1.2);

pair<int, double> p2 = p1;

pair<int, double> p1;

p1 = make\_pair(1, 1.2);

~~WEEK 2.2 STL Algorithms

\*\*Principles of testing：

测试API，而不是实现不要让测试变得脆弱如果您的代码发生变化，

您的测试应该最小化更改使测试变得简单明显出现的问题

不要在测试中放置if语句或循环任何复杂的代码都应放入名字很好的功能

\*\*Common algorithms

template <typename T>

T sum(iterable<T> cont) {

T total;

for (auto it = std::begin(cont); std::end(cont); ++

total += \*it;

}

return total

}

\*\*求和：

auto midpoint = v.begin() + (v.size() / 2);

// This looks a lot harder to read. Why might it be better?

auto midpoint = std::next(v.begin(), std::distance(v.begin(), v.end()) / 2);

int sum = std::accumulate(v.begin(), midpoint, 0);

int sum = std::accumulate(v.begin(), v.end(), 0);

\*\* 排序：

std::vector<int> sortedVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::lower\_bound(sortedVec.begin(), sortedVec.end(), 5);

std::list<int> sortedLinkedList{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::lower\_bound(sortedLinkedList.begin(), sortedLinkedList.end(), 5);

\*\*Lambda captures

std::for\_each(v.begin(), v.end(), [] (int& val) { val = val + n; });

By Value:

void AddN(std::vector<int> vec, int n)

std::for\_each(vec.begin(), vec.end(),

[=] (int& item) { item += n; });

}

By reference:

std::map<std::string, int> m;

auto emplace = [&] (

const auto& key,

const auto& value) {

m.emplace(key, value);

};

emplace("hello", 5); 这个emplace相当于包含一个lamda函数

by expression:

std::vector<int> vec{1, 2, 3};

int n = 10;

auto fn = [vec{std::move(vec)}, y=n + 1] () {

std::cout << vec.size() << '\n' << y;

};

// Should be 0

std::cout << vec.size() << '\n';

fn();

~~WEEK3 Object-Oriented Programming ，constructor

\*\*Scope: 变量的范围是可以访问的程序的一部分。

范围从变量定义开始。

范围（通常）在下一个结束“}”即使您从未见过将变量定义为关闭，

您也可能对此很熟悉尽可能第一次使用

Object Lifetimes:

个对象是一个特定类型的内存，它包含一些数据。

所有变量都是对象。

与许多其他语言不同，这不会增加开销。

当它进入范围时，对象的生命周期开始“构造”对象每个类型都有一个或多个构造函数如何构造它

对象生命周期在超出范围时结束“Destructs”对象每个类型都有一个不同的“析构函数”，告诉编译器如何销毁它

\*\*Construction: 一般用use（）来调用函数，而{}来构造对象

（）只能用于函数，而{}可以用于任何一种罕见的情况这些都是不同的有时它在构造函数和初始化列表之间是不明确的

4种，默认，有参数，copy,move。

friend constructor, destructor, opeator ,move ass, copy ass 练习:

class

~~WEEK 4.1 Operator Overloading

重载操作符要注册表格

Type Operator(s) Member / friend

\*\*I/O <<,>> friend

friend std::ostream& operator<<(std::ostream& os, const EuclideanVector& v);

std::ostream&operator<<(std::ostream&os,constPoint&p){

os << "(" << p.x\_ << ", "<< p.y\_ << ")" ;

return os;

}

friend std::istream& operator>>(std::istream& is, const EuclideanVector& v);

\*\*Arithmetic +, -, \*, / friend

friend EuclideanVector operator\*(const EuclideanVector& lhs, const double& rhs);

\*\*Relational, Equality >,<,>=,<=,==,!= friend

friend bool operator==(const Point& p1, const Point& p2) {

return p1.x\_ == p2.x\_ && p1.y\_ == p2.y\_; // return std::tie(p1.x\_, p1.y\_) == std::tie(p2.x\_, p2.y\_);

}

\*\*assignments = member (non-const)

EuclideanVector& operator=(const EuclideanVector&); //copy assignment 正常

EuclideanVector& operator=(EuclideanVector&&) noexcept; //move assignment

//copy assignment

EuclideanVector& EuclideanVector::operator=(const EuclideanVector& orig) {

if (this != &orig) {

EuclideanVector tmp{orig};

std::swap(tmp, \*this);

}

return \*this;

}

// move assignment

EuclideanVector& EuclideanVector::operator=(EuclideanVector&& orig) noexcept {

// because the pointer is unique pointer, so we just change the pointing position

// and the memory which the unique pointer pointed will be destructed automatically

if (this != &orig) {

magnitudes\_ = std::move(orig.magnitudes\_);

num\_dim = orig.num\_dim;

orig.num\_dim = 0;

}

return \*this;

}

\*\*Compound assignments +=,-=,\*=,/= member (non-const)

EuclideanVector& operator+=(const EuclideanVector& p);

Point& Point::operator+=(const EuclideanVector& p) {

this->x\_ += p.x\_;

this->y\_ += p.y\_;

return \*this;

}

\*\*Subscript [] member (both)

double& operator[](int index); //得到reference

double operator[](int index) const; //正常

double& EuclideanVector::operator[](int index) {

assert(index >= 0 && index < num\_dim);

return magnitudes\_[index];

}

\*\*Increment/Decrement ++,-- member (non-const)

RoadPosition& operator++(); //prefix ++x

RoadPosition& operator++(int); //postfix x++

RoadPosition& RoadPosition::operator++() {

this->tick\_(); //++x

return \*this;

}

RoadPosition RoadPosition::operator++(int) {

RoadPosition rp = \*this; //copy

this->tick\_(); //++x

return rp; //old

}

void RoadPosition::tick\_() {

++(this->km\_from\_sydney\_);

}

\*\*Arrow, Deference ->,\* member (both)

std::string\* operator->() { return ptr; }

std::string& operator\*() { return \*ptr; }

\*\*Call () member

// Type conversion operators

operator std::vector<int>() { //强制转换

std::vector<int> vec;

vec.push\_back(x\_);

vec.push\_back(y\_);

return vec;

}

在特定实例的上下文中调用操作时使用members

在没有任何特定实例的情况下调用操作时使用朋友(即使他们不需要访问私人信息)

\*\* 指针操作重载 \* 和 -> :

因为ptr指向一个实例或heap 所以返回指针就是可以把class里面private指针拿出来用；

std::string\* operator->() { return ptr; } // 返回指针类型 \*

相当把指向的区域解读出来。

std::string& operator\*() { return \*ptr; } // 返回解引用后数据 对象的&

\*\*隐性类型转换重载：也就是类似于 double a= 0.0; int b= 1; a=b;

operator std::vector<int>() {

std::vector<int> vec;

vec.push\_back(x\_);

vec.push\_back(y\_);

return vec;

}

Point p{1,2};

std::vector<int> vec = static\_cast<std::vector<int>>(p);

或者 std::vector<int> vec = p;

～～WEEK5.1 SMART POINTER

#include <memory>

\*\*生命周期：要在C ++中创建安全的对象生存期，我们总是将一个对象的生命周期附加到其他对象的生命周期

命名对象： 函数中的变量与其范围相关联

数据成员与类实例的生命周期相关联

std :: vector中的元素与向量的生命周期相关联

未命名的对象：

堆对象应该与创建它的任何对象的生命周期联系在一起

糟糕的编程实践示例拥有的原始指针与任何东西都没有关系

C样式的数组与任何东西都没有联系

void fn() {

// Similar to C's malloc

MyIntPointer p{new int{5}};

// Copy the pointer;

MyIntPointer q{p.GetValue()};

// p and q are both now destructed.

// What happens?

}

这个时候 会删除两次，因为在p结束时候已经释放了堆内存，q会出错

所以如果使用构建时候不要把指针p{q}这样不会只是多加一个指针，而是要创建新的heap

\*\*Smart Pointers

Type Shared access Shared ownership Take ownership

std::unique\_ptr<T> No No YES

std::shared\_ptr<T> YES YES YES

std::weak\_ptr<T> YES NO NO

std::experimental::observer\_ptr<T> YES NO NO

\*\*std::unique\_ptr<T>:

建立最好方式：

std::unique\_ptr<std::string> up3 = std::make\_unique<std::string>("Hello");

std::cout << \*(up3.get()) << "\n";

up3.realease()

\*\*std::shared\_ptr<T>, std::weak\_ptr<T>:

int\* i = new int;

\*i = 5;

std::shared\_ptr<int> x{i};

std::shared\_ptr<int> y = x; // Both now own the memory

std::cout << "use count: " << x.use\_count() << "\n";

std::cout << "value: " << \*x << "\n";

x.reset(); // Memory still exists, due to y.

std::cout << "use count: " << y.use\_count() << "\n";

std::cout << "value: " << \*y << "\n";

y.reset(); // Deletes the memory, since

// no one else owns the memory

std::cout << "use count: " << x.use\_count() << "\n";

std::cout << "value: " << \*y << "\n";

建立方式：

std::shared\_ptr<int> x = std::make\_shared<int>(1);

std::weak\_ptr<int> wp = x; // x owns the memory

{

std::shared\_ptr<int> y = wp.lock(); // x and y own the memory 检查wp所指向的的数据owner(share pointer)还存在不

if (y) {

// Do something with y

std::cout << "Attempt 1: " << \*y << '\n';

}

} // y is destroyed. Memory is owned by x

\*\*什么时候用什么指针：

几乎总是希望用unique超过shared， 也就是知道生命周期 最长的为unique，unique拥有 raw观察

如果你有几个指针，你不知道哪一个会留在最长的，则用shared拥有，weak观察加临时拥有（share 释放后）

～～WEEK5.2 EXCEPETION

exception 最常见的问题

\*\*runtime\_error 只有在运行时才能检测出的问题

range\_error 运行时错误：生成的结果超出了有意义的值域范围

overflow\_error 运行时错误：计算上溢

underflow\_error 运行时错误：计算下溢

\*\*logic\_error 程序逻辑错误

domain\_error 逻辑错误：参数对应的结果值不存在

invalid\_argument 逻辑错误：参数无效

length\_error 逻辑错误：试图创建一个超过该类型最大长度的对象

out\_of\_range 逻辑错误：使用一个超出有效范围的值

\*\*Exceptions are treated like lvalues

Throw by value, catch by const reference & ： more efficient, less copying (exploring today)

no slicing problem (related to polymorphism, exploring later)

Rethrow: 如果当前层捕获异常后处理，需要把异常再抛出到上一层，则在catch最后再次throw。之后上一层继续处理异常

Stack unwinding：栈展开是如果函数调用出现异常，则在最里面先处理，如果没有catch或者处理不了，则把函数从栈堆先移除，之后

继续下一个，直到处理为止。而处理不了异常的函数在throw之后不会继续执行，临时变量会调用对应析构函数，指针需要手动删除

堆栈展开是退出堆栈帧的过程，直到找到函数的异常处理程序

在堆栈展开期间，如果异常从析构函数中离开（也就是析构函数有异常但是处理不了），将调用std :: terminate（）不会继续栈展开。因为如果异常离开析构函数，则可能无法正确释放资源。

析构函数内部发生的所有异常都应该在析构函数内处理析构函数通常不抛出，并且需要显式选择抛出STL类型（不要那样做）

RALL：Resource acquisition is initialisation（资源获取是初始化）：构建是获取，析构是释放。new ,delete 等等。

Partial construction：当构建时候出现异常，会暂停构建解决异常或者终止程序，

这时候如果析构函数函数有操作或者释放heap，那么会释放和操作不了。

Partial construction solution:使用smart pointer。 这时候当停止时候会自动释放heap。 或者在构建时候捕获，并且处理

Exception safety levels：Operations performed have various levels of safety

No-throw (failure transparency) 最好

Strong exception safety (commit-or-rollback) 强

Weak exception safety (no-leak)。弱

No exception safety 无

C++异常安全一般有四个等级的要求(异常安全等级由低到高): 1. 没有任何异常安全保证, 也就是异常一旦发生, 可能造成程序行为的未定义; 2. 基本保证, 也就是异常发生的时候, 程序的行为还是合法的, 状态也都是有效的, 行为是有定义的, 但是程序实例的状态有可能改变(仍旧合法) 3. 强保证(回滚保证), 这个等级就要求异常一旦发生然后进行处理了以后, 要么一次性全部成功, 要么就回滚到异常钱的原始状态(程序状态和异常发生以前一模一样). 4. 保证不会有任何一方的发生.

这里面4是最不安全的, 不可取. 1基本上等级最强, 但是一般情况下不可能满足. 所以异常安全往往在2和3这两个等级间取舍. 等级3有可能会有额外的负担, 资源消耗等. 具体情况根据程序逻辑和实际情况判断取舍

函数提供的异常安全性的最高等级只能是你实现中调用的各个函数中异常安全性等级最低的那个

从initial list中抛出异常，也就是在没有构建的时候先检查，分配资源前抛出异常

ThrowException::ThrowException(string m, int f) throw(string):msg(m),b(f) { //B

if(b==1)

throw "b=1 not allowed.";

}

～～WEEK6： TEMMPLATE

Type parameter: Unknown type with no value

Nontype parameter: Known type with unknown value

\*\*Function TEMMPLATE:

using S = int; // 把int类型名称换成S 自定义类型名称

template <typename T> // template parameter list

T min(T a,T b) { //template type parameters

return a;

}

\*\*Class TEMMPLATE:

template <typename T, int size> 包含nontype 参数

在函数中使用要传入到参数里面

T findmin(const std::array<T, size> a)

在class中使用，要传入到基本constructor里面

explicit Queue<T,max\_size>() : queue\_{std::make\_unique<T[]>(max\_size)} {}

Queue<T,max\_size>(int i) : ... {}

Advantage: Improved performance. Size of a queue is determined at compile time requiring less to be done / processed at runtime.

Disadvantage: (1) Code explosion - instantiation created for a queue of every size. (2) Unable to copy construct easily

template <typename T>

class Stack {

public:

friend std::ostream& operator<<(std::ostream& os, const Stack& s) {

for (const auto& i : s.stack\_)

os << i << " ";

return os;

}

void push(T);

void pop();

T& top();

const T& top() const;

bool empty() const;

private:

std::vector<T> stack\_;

};

\*\*.tpp file like .cpp file

template <typename T>

void Stack<T>::push(const T item) {

stack\_.push\_back(item);

}

template <typename T>

void Stack<T>::pop() {

stack\_.pop\_back();

}

template <typename T>

T& Stack<T>::top() {

return stack\_.back();

}

template <typename T>

const T& Stack<T>::top() const {

return stack\_.back();

}

template <typename T>

bool Stack<T>::empty() const {

return stack\_.empty();

}

~~WEEK7 Templates, Part 2

\*\*Inclusion compilation model 包含编译模型

为什么.h在.cpp上面, .tpp在.h下面。 因为 在编译时候时候，.cpp是固定的参数类型 参数都是有类实例的，编译器知道。

关键在于main.cpp是插入的.h文件 而.h在没有模版时候 是由固定类型的，所以.cpp的也是同类型。所以编译可以知道具体实例的类型匹配不，之后编译。

而对于模版的话，分开编译.cpp和.main时候。.cpp不知道实例，所以无法确定T的类型。所以需要把cpp放到.h后面 可以直接编译main.cpp 之后在-o

（也就是原始.cpp 和 main.cpp可以分开在连接， 现在需要放到一起，而插入头文件是 所以头文件要包含.cpp文件，所以放到下面）

而模版定义不是，模板定义需要在编译时知道（模板定义不能在链接时实例化，因为这需要对所有类型进行实例化）

…………另一种是显示定义所有可能在.cpp中

template <typename T>

T min(T a, T b) {

return a < b ? a : b;

}

template int min<int>(int, int);

template double min<double>(double, double);

\*\*Static Members：每个模板实例化都有自己的静态成员集

\*\*Friends:每个堆栈实例化都有一个朋友的唯一实例化

\*\*Default Members:template <typename T, typename CONT = std::vector<T>>

~~WEEK7.2 Custom Iterators

\*\*迭代器：

迭代器是指针的抽象概念

迭代器是将容器数据抽象为对象序列的类型

容器和算法之间的粘合剂

算法设计者不关心数据结构的细节

数据结构的设计者不必提供广泛的访问操作

\*\*Iterator invalidation：迭代器失效

不能再循环时候简单的对迭代器做删除插入的操作

push back 还有 earse失效

push back因为会增加size如果大雨容量，那么所有的迭代器会失效（undefined behavior)

对于节点式容器(map, list, set)元素的删除，插入操作会导致指向该元素的迭代器失效，其他元素迭代器不受影响

对于顺序式容器(vector，string，deque)元素的删除、插入操作会导致指向该元素以及后面的元素的迭代器失效

std :: vector在添加元素时使一切无效

std :: unordered\_（map / set）在添加元素时使所有内容无效

std :: map / set在插入时不会使迭代器失效(红黑树)

\*\*正确方法

节点容器：

std::set<int> myset = {1, 2, 3, 4, 5, 6};

    std::set<int>::iterator it = myset.begin();

    for (; it != myset.end() ;) {

        if (\*it == 3) {

            myset.erase(it++);

        } else {

          ++it;

        }

    }

顺序容器：

std::vector v{1, 2, 3, 4, 5};

// Erase all even numbers (C++11 and later)

for (auto it = v.begin(); it != v.end(); ) {

if (\*it % 2 == 0) {

it = v.erase(it);

} else {

++it;

}

}

\*\*迭代器特征(编译自己的迭代器需要做的）:

类别（输入，输出，转发，双向，随机访问）

Value type (T)

Reference Type (T& or const T&)

Pointer Type (T\* or T\* const)

差异类型（用于计算迭代器之间距离的类型

\*\*迭代器要求或要求:

class Iterator {

public:

using iterator\_category = std::forward\_iterator\_tag;

using value\_type = T;

using reference = T&;

using pointer = T\*; // Not strictly required, but nice to have.

using difference\_type = int;

reference operator\*() const;

Iterator& operator++();

Iterator operator++(int) {

auto copy{\*this};

++(\*this);

return copy;

}

// This one isn't strictly required, but it's nice to have.

pointer operator->() const { return &(operator\*()); }

friend bool operator==(const Iterator& lhs, const Iterator& rhs) { ... };

friend bool operator!=(const Iterator& lhs, const Iterator& rhs) { return !(lhs == rhs); }

};

\*\*Container requirements:

All a container needs to do is to allow std::[cr]begin / std::[cr]end

(这允许在range-for循环和std算法中使用)

最简单的方法是定义begin / end / cbegin / cend方法

By convention, we also define a type Container::[const\_]iterator

\*\*Dissecting IntStack(解析):

迭代器特征,重载运算符（\*， - >）,等于运算符,构造函数（defualt to nullptr）

私有数据:

迭代器是在类中定义的，因此可以访问私有数据

Iterator将容器定义为构造函数的友元类

List类中的关键点:

begin（） - 返回一个Iterator对象

end（） - 返回一个Iterator对象（使用nullptr作为私有数据）

注意：除非通过返回引用，否则Iterator类不会修改List / Node数据。

\*\*自定义bidirectional双向迭代器:

Need to define operator--() on your iterator

Need to move from c.end() to the last element

c.end() can't just be nullptr

Need to define the following on your container:

class Container {

// Make the iterator

class reverse\_iterator {...};

// or

using reverse\_iterator = ...;

// Need to define these.

reverse\_iterator rbegin();

reverse\_iterator rend();

// If you want const reverse iterators (hint: you do), define these.

const\_reverse\_iterator rbegin() const { return crbegin(); }

const\_reverse\_iterator crbegin();

const\_reverse\_iterator rend() const { return crend(); }

const\_reverse\_iterator crend() const;

};

++操作 前向iterator 是 node\_=node\_->next.get(); return \*this(iterator); 来把现在变为下一个

\*\*Automatic reverse iter verse iterators

class Container {

// Make the iterator using these.

using reverse\_iterator = std::reverse\_iterator<iterator>;

using const\_reverse\_iterator = std::reverse\_iterator<const\_iterator>;

// Need to define these.

reverse\_iterator rbegin() { return reverse\_iterator{end()}; }

reverse\_iterator rend() { return reverse\_iterator{begin()}; }

// If you want const reverse iterators (hint: you do), define these.

const\_reverse\_iterator rbegin() const { return crbegin(); }

const\_reverse\_iterator rend() const { return crend(); }

const\_reverse\_iterator crbegin() const { return const\_reverse\_iterator{cend()}; }

const\_reverse\_iterator crend() const { return const\_reverse\_iterator{cbegin()}; }

};

Reverse iterators can be created by std::reverse\_iterator

rbegin() stores end(), so \*rbegin is actually \*(--end())

\*\*Random access iterators 随机访问：

class Iterator {

...

using iterator\_category = std::bidirectional\_iterator\_tag

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = int;

Iterator& operator+=(difference\_type rhs) { ... }

Iterator& operator-=(difference\_type rhs) { return \*this += (-rhs); }

reference operator[](difference\_type index) { return \*(\*this + index); }

friend Iterator operator+(const Iterator& lhs, difference\_type rhs) {

Iterator copy{\*this};

return copy += rhs;

}

friend Iterator operator+(difference\_type lhs, const Iterator& rhs) { return rhs + lhs; }

friend Iterator operator-(const Iterator& lhs, difference\_type rhs) { return lhs + (-rhs); }

friend difference\_type operator-(const Iterator& lhs, const Iterator& rhs) { ... }

friend bool operator<(Iterator lhs, Iterator rhs) { return rhs - lhs > 0; }

friend bool operator>(Iterator lhs, Iterator rhs) { return rhs - lhs < 0; }

friend bool operator<=(Iterator lhs, Iterator rhs) { !(lhs > rhs); }

friend bool operator>=(Iterator lhs, Iterator rhs) { !(lhs < rhs); }

}

~~WEEK8.1 Custom Iterators

\*\*default member:

就是template <typename T, typename Cont = std::vector<T>> 当cont没有指定时候 会用默认的类型，这样传入参数会简便很多。

\*\*partial spectialisation:

在 kind层次上的，对于正常的class template来说，所有的类型的操作是一样的，而有的类型在需要特别操作对于。 例如,加法对于pointer和int是不一样的。

所以对于pointer传入类型来说要单独写一个特例的class去解决。这就是部分特例化。（kind 层）

（不能单独特化函数）

\*\*Explict spectialisation:

对于partial spectialisation来说，是传入类型需要特别处理，例如pointer，class.但是具体pointer是什么类型,确不知道。int还是double。 而这个时候如果int和double处理还不一样

就需要显示的把具体类型写出来和解决。

（只应在类上进行显式专门化）

\*\*什么时候特例化：

1.您需要保留现有语义，否则将无法正常工作。

2.你想写一个类型特征

3.您可以针对特定类型进行优化

什么时候不用特例化：

1.Don't specialise functions：函数不能部分专用完全特例化

函数最好用重载完成，因为

\*\*Type Traits:类型特征

Traits：表征类型的类（或class模板）

traits一般是利用编译器的能力来获取一些信息。采取的实现方案是模板与模板特化。

template <typename T>

struct numeric\_limits {

static T min();

};

template <>

struct numeric\_limits<int> {

static int min() { return -\_\_INT\_MAX\_\_ - 1; }

}

template <>

struct numeric\_limits<float> {

static int min() { return -\_\_FLT\_MAX\_\_ - 1; }

}

#include <iostream>

template <typename T>

struct is\_pointer {

static const bool val = false;

};

template<typename T>

struct is\_pointer<T\*> {

static const bool val = true;

};

int main() {

std::cout << is\_pointer<int\*>::val << "\n";

std::cout << is\_pointer<int>::val << "\n";

}

T largest = std::numeric\_limits<T>::min();

相当于对当传入类型可以去找对应min函数，而如果用函数重载或者纯templete则对一些case coveer不到。

这里面主要用模版特例化或者显示特例让min找到用T找到对应的函数或者类进行操作。类似两层操作

还有就是主要用在查找value也就是类型的判断，之后用做判断

std::cout << is\_pointer<int\*>::val << "\n";

is\_pointer是用部分特例化进行type traits。这样任何类型，都可以判断下是不是pointer。而不是一个一个写

\*\*Variadic模板：

template <typename T>

void print(const T& msg) {

std::cout << msg << " ";

}

template <typename A, typename... B> //...代表很多其他类型 （可变参数模版）

void print(A head, B... tail) {

print(head); 用递归方法进行输出。 这就相当于如果有一个则调用上面，多个调用下面

print(tail...);

}

\*\*Member Templates

Stack<int> is1;

Stack<double> ds1{is1}; 这样是不行的 因为模版类型不一样，没有转换操作重写或者转换，copy默认会报错

template <typename T>

explicit Stack() {}

template <typename T2>

Stack(Stack<T2>&);

void push(T t) { stack\_.push\_back(t); }

实现copy constructor

template <typename T>

template <typename T2>

Stack<T>::Stack(Stack<T2>& s) {

while (!s.empty()) {

stack\_.push\_back(static\_cast<T>(s.pop()));

}

}

\*\*Template Templa te Template Parameters

template <typename T, template<typename...> class Cont> 后面相当一个模版作为参数

class Stack {

public:

void push(T t) { stack\_.push\_back(t); }

void pop() { stack\_.pop\_back(); }

T& top() { return stack\_.back(); }

bool empty() const { return stack\_.empty(); }

private:

Cont<T> stack\_;

};

Stack<int, std::vector<int>> s1 变成 Stack<int, std::vector> s1;

\*\*模板参数推导：

是从传入的值推导出模版或者函数的类型，例如：

template<typename T>

T min(T a,T b){

return a<b ? a:b

}

min<int>(1,2) 可以直接min(1 ,2)

\*\*implicit 推导：

非类型参数：隐式转换的行为与普通类型转换的行为类似。类型参数：三种可能的隐式转换

就是传进来的和实际要的不是一模一样，但是可以转换或者推导。

\*\*Explicit 推导：

如果我们需要更多地控制正常的演绎过程，我们可以明确指定传入的类型

template <typename T>

T min(T a, T b) {

return a < b ? a : b;

}

int main() {

int i; double d;

min(i, static\_cast<int>(d)); // int min(int, int)

min<int>(i, d); // int min(int, int)

min(static\_cast<double>(i), d); // double min(double, double)

min<double>(i, d); // double min(double, double)

}

~~WEEK8.2 Advance Types

\*\* decltype:就是c++ typeof 推断参数类型的，因为auto在函数参数里不能用。

Rule 1:

If expression e is any of:

variable in local scope

variable in namespace scope

static member variable

function parameters

then result is variable/parameters type T

Rule 2: if e is an lvalue, result is T& 左值

Rule 3: if e is an xvalue, result is T&& 右值

Rule 4: if e is a prvalue, result is T

eg:

int i;

int j& = i;

decltype(i) x; // int; - variable

decltype(j) y; // int& - lvalue

decltype(5); // int - prvalue

\*\*Determining return types：??? 是auto

decltype(\*beg) This will not work, as beg is not declared until after the

reference to beg

template <typename It>

??? find(It beg, It end, int index) {

for (auto it = beg, int i = 0; beg != end; ++it; ++i) {

if (i == index) {

return \*it;

}

}

return end;

}

template <typename It>

auto find(It beg, It end, int index) -> decltype(\*beg) {

for (auto it = beg, int i = 0; beg != end; ++it; ++i) {

if (i == index) {

return \*it;

}

}

return end;

}

Iterator用于模板化集合并返回对特定索引处的项的引用

\*\*类型转换：许多add，remove和make函数作为类型特征的一部分存在，提供转换类型的能力

#include <iostream>

#include <type\_traits>

template<typename T1, typename T2>

void print\_is\_same() {

std::cout << std::is\_same<T1, T2>() << std::endl;

}

int main() {

std::cout << std::boolalpha;

print\_is\_same<int, int>();

// true

print\_is\_same<int, int &>(); // false

print\_is\_same<int, int &&>(); // false

print\_is\_same<int, std::remove\_reference<int>::type>();

// true

print\_is\_same<int, std::remove\_reference<int &>::type>(); // true

print\_is\_same<int, std::remove\_reference<int &&>::type>(); // true

print\_is\_same<const int, std::remove\_reference<const int &&>::type>(); // true

}

int main() {

typedef std::add\_rvalue\_reference<int>::type A;

typedef std::add\_rvalue\_reference<int&>::type B; A& && 变为A&， A&& & 变为 A&

typedef std::add\_rvalue\_reference<int&&>::type C;

typedef std::add\_rvalue\_reference<int\*>::type D; int\* &&

std::cout << std::boolalpha

std::cout << "typedefs of int&&:" << "\n";

std::cout << "A: " << std::is\_same<int&&, A>>::value << "\n"; //true

std::cout << "B: " << std::is\_same<int&&, B>>::value << "\n"; //false

std::cout << "C: " << std::is\_same<int&&, C>>::value << "\n"; //trye

std::cout << "D: " << std::is\_same<int&&, D>>::value << "\n"; //false

}

\*\* Binding

类似于传入的值类型和fun写的不一样，但是可以绑定（转换）

template T&& 绑定everything

template <typename T>

void foo(T&& a);

T& 绑定左值

T&& 绑定右值

const T&绑定everything

#include <iostream>

void print(const std::string& a) {

std::cout << a << "\n";

}

const std::string goo() {

return "C++";

}

int main() {

std::string j = "C++";

const std::string& k = "C++";

print("C++"); // rvalue

print(goo()); // rvalue 没有地址

print(j); // lvalue

print(k); // const lvalue

}

#include <iostream>

template <typename T>

void print(T&& a) {

std::cout << a << "\n";

}

const std::string goo() {

return 5;

}

int main() {

int j = 1;

const int &k = 1;

print(1); // rvalue, foo(int&&)

print(goo()); // rvalue foo(const int&&)

print(j); // lvalue foo(int&)

print(k); // const lvalue foo(const int&)

}

\*\*Forwarding functions

如果函数封装时候，传入的是const reference（因为不想改变），那么需要变量的函数有可能报错，

因为它有可能需要rvalues。

fn(value is lvalue ? value : std::move(value));

\*\*std::forword 的使用：(不copy，直接return对应的值)

Returns reference to value for lvalues

Returns std::move(value) for rvalues

// This is approximately std::forward.

template <typename T>

T& forward(T& value) {

return static\_cast<T&>(value);

}

template <typename T>

T&& forward(T&& value) {

return static\_cast<T&&>(value);

}

template <typename T>

auto wrapper(T&& value) {

return fn(std::forward<T>(value));

}

wrapper(std::move(x));

\*\*std::forward and variadic templates：

通常你需要调用你一无所知的函数：

它可能有任何数量的参数

每个参数可能是一个不同的未知类型

每个参数可能是一个左值或右值

template <typename... Args>

auto wrapper(Args&&... args) {

// Note that the ... is outside the forward call, and not right next to args.

// This is because we want to call

// fn(forward(arg1), forward(arg2), ...)

// and not

// fn(forward(arg1, arg2, ...)

return fn(std::forward<Args>(args)...);

}

\*\*std::forword 的使用：

std :: forward的唯一真正用途是当你想要包装一个函数时。

这可能是因为：你想在之前或之后做一些其他事情（例如std :: make\_unique / std :: make\_shared需要将它包装在unique / shared\_ptr变量中）

你想做一些稍微不同的事情（例如.std :: vector ::\templace使用未初始化的内存构造）

你想添加一个额外的参数（例如，总是调用一个函数，最后一个参数为1）。这通常不是很有用，因为它可以用std :: bind或lambda函数实现。

～～WEEK 8.3 Template Metaprogramming 模板元编程

\*\*Metaprogramming 元编程

元编程：编写计算机程序，能够将其他程序代码视为数据。

在C ++中，我们可以通过以下任一方式进行元编程：使用constexpr模板元编程

C ++模板元编程：编译通过模板的递归实例化创建的时间计算值

#include <iostream>

template<int n> struct Factorial {

static const long val = Factorial<n-1>::val \* n;

};

template<> struct Factorial<0> {

static const long val = 1; // must be a compile-time constant

};

int main() {

std::cout << Factorial<6>::val << std::endl;

}

～～WEEK 9 Runtime Polymorphism 多态 静态动态绑定，继承，虚函数等等

继承 inheritance：

\*\*protect 可以被子类访问， private可以被朋友和自己内部访问

class Derived : public BaseClass 一般public

\*\*子类大小包含父类，在construct时候会构建父类。

class BaseClass {

public:

BaseClass(int member): int\_member\_{member} {}

private:

int int\_member\_;

std::string string\_member\_;

}

class SubClass: public BaseClass {

public:

SubClass(int member, int\* ptr): BaseClass{member}, ptr\_member\_{ptr} {}

// Won't compile.

SubClass(int member, int\* ptr): int\_member\_{member}, ptr\_member\_{ptr} {}

private:

std::vector<int> vector\_member\_;

int\* ptr\_member\_;

}

当没有复写虚函数时候，在子类的实例调用function时候会静态调用也就是 看实例是什么类型就是

什么类型.

\*\*虚函数

Virtual functions deterimn at runtime。是哪个子类的函数就找哪个

deconstructor 也要是虚函数；

\*\*虚函数表：

class A {

public:

virtual void f(int) {}

virtual int g() {}

void a() {}

virtual ~A() {}

};

class B : public A {

public:

void f(int) override {}

virtual int h() {}

};

class C : public B {

public:

virtual void f(int, int) {}

virtual void x() {}

static void b() {}

};

A B C

A::f(int) B::f(int) B::f(int)

A::g() A::g() A::g()

~A() ~B() ~C(） //这是因为A,B,C 复写就放前面

B::h() B::h()

C::f(int, int)

C::x()

\*\*final:

当一个子类有final时候，之后的子类就不可以继承了。

指定编译器“这对于任何子类都不是虚拟的

”如果编译器有一个类型为SubClass＆的变量，它现在不再需要在vtable中查找它。

这意味着静态绑定如果你有一个SubClass＆，但是对于BaseClass＆的动态绑定

\*\*Abstract Base Classes (ABCs)抽象类：至少有一个纯虚函数

可能想要处理基类，但基类本身是无意义的

函数接受“可点击”可能需要一些默认行为和数据，

但需要其他所有文件都有一个名称，

但是通过网络或磁盘完成读取如果一个类至少有一个“抽象”（纯虚拟的）

C ++）方法，类是抽象的，它可以有构造函数和析构函数这些函数提供了构造

和破坏任何派生类的ABC子对象的语义

\*\*纯虚函数：

当你有一个子类可能要覆盖的默认实现时，

虚函数很适合有时没有默认可用纯虚函数指定一个类必须覆盖的函数才能不抽象

\*\*重要Creating polymorphic objects：

std::vector<std::unique\_ptr<BaseClass>> base; //std::vector<BaseClass> base;

base.push\_back(std::make\_unique<BaseClass>()); //替换base.push\_back(BaseClass{});

base.push\_back(std::make\_unique<Subclass1>());

base.push\_back(std::make\_unique<Subclass2>());

这样解决了动态静态绑定，slice problem

\*\*破坏多态对象:

每当你编写一个要继承的类时，

总是让你的析构函数虚拟记住：当你声明一个析构函数时，移动构造函数和赋值不会被synthesized

class BaseClass {

BaseClass(BaseClass&&) = default;

BaseClass& operator=(BaseClass&&) = default;

virtual ~BaseClass() = default;

}

\*\*Static and dynamic types:

由于对象切片，既不是引用又是指针的对象始终具有相同的静态和动态类型

\*\*动态静态类型绑定：

#include <iostream>

class B {

public:

virtual void f() {

std::cout << "B ";

}

};

class D : public B {

public:

void f() override {

std::cout << "D ";

}

};

int main() {

B b; //静态 B

D d; //静态 D

b = d;

B& bref = b; //静态 B&

D& dref = d; //静态 &

B& dbref = d; //静态 B&

b.f(); // 动态B==静态

d.f(); // 动态D==静态

bref.f(); //动态B 和静态B& 区别 （也就是如果f是虚函数）

dref.f(); //动态D

dbref.f(); //动态D

}

The static type is the declared type of a variable.

The dynamic type is the type a variable actually is at runtime.

动态绑定是指针或引用关联的类型所以在runtime

最后就是B D B D D输出

\*\*up cast:

从派生类到基类的转换称为向上转换此转换始终是安全的所有狗都是动物

Dog dog;

// Up-cast with references.

Animal& animal{dog};

// Up-cast with pointers.

Animal\* animal{&dog};

// What's this (hint: not an up-cast)?

Animal animal{dog};

\*\*Down-casting:

从基类转换为派生类称为向下转换此转换不安全并非所有动物都是狗

Dog dog;

Cat cat;

Animal& animal\_dog{dog};

Animal& animal\_cat{cat};

// Attempt to down-cast with references.

// Neither of these compile.

// Why not?

Dog& dog\_ref{animal\_dog};

Dog& dog\_ref{animal\_cat};

\*\*如何Down\_cast:

编译器不知道Animal是否恰好是Dog如果您知道它是，您可以使用static\_cast

否则，您可以使用dynamic\_cast

如果指针类型不匹配则返回空指针

如果它没有引用引用类型不匹配，则抛出异常。

Dog dog;

Cat cat;

Animal& animal\_dog{dog};

Animal& animal\_cat{cat};

// Attempt to down-cast with references.

Dog& dog\_ref{static\_cast<Dog&>(animal\_dog)};

Dog& dog\_ref{dynamic\_cast<Dog&>(animal\_dog)};

// Undefined behaviour (incorrect static cast).

Dog& dog\_ref{static\_cast<Dog&>(animal\_cat)};

// Throws exception

Dog& dog\_ref{dynamic\_cast<Dog&>(animal\_cat)};

Dog dog;

Cat cat;

Animal& animal\_dog{dog};

Animal& animal\_cat{cat};

// Attempt to down-cast with pointers.

Dog\* dog\_ref{static\_cast<Dog\*>(animal\_dog)};

Dog\* dog\_ref{dynamic\_cast<Dog\*>(animal\_dog)};

// Undefined behaviour (incorrect static cast).

Dog\* dog\_ref{static\_cast<Dog\*>(animal\_cat)};

// returns null pointer

Dog\* dog\_ref{dynamic\_cast<Dog\*>(animal\_cat)};

\*\*Static and dynamic binding :

静态绑定：决定在编译时调用哪个函数（基于静态类型）

动态绑定：确定在运行时调用哪个函数（基于动态类型)

c++:

Statically typed (types are calculated at compile time)

Static binding for non-virtual functions

Dynamic binding for virtual functions

\*\*Types of functions：

virtual void fn() = 0; pure virtual，Inherit interface only

virtual void fn() {}；virtual ，Inherit interface with optional implementation

void fn() {} ； nonvirtual，Inherit interface and mandatory implementation（强制实现）

\*\*Covariants：

相当于子类返回的值应该在父类返回的值等于或之下，也就是子类继承父类，函数返回类型也有

继承关系的话就返回父类之下的。

就是看返回类型能推导出父类不。

（等待）我要返回时landanimal，如果你返回animal不是land的话不行，而你返回的dog一定是land

class Base {

virtual LandAnimal& GetFavoriteAnimal();

};

class Derived: public Base {

// Fails to compile: Not all animals are land animals.

Animal& GetFavoriteAnimal() override;

// Compiles: All land animals are land animals.

LandAnimal& GetFavoriteAnimal() override;

// Compiles: All dogs are land animals.

Dog& GetFavoriteAnimal() override;

}

\*\*Contravariants：

相反的，传入函数。

（主动）我准备要传入的是landaniamal，如果你要传入是animal，那我一定会满足你，但是你要dog，

那么我有可能传入cat。这样就不满足

vclass Base {

virtual void UseAnimal(LandAnimal&);

};

class Derived: public Base {

// Compiles: All land animals are valid input (animals).

void UseAnimal(Animal&) override;

// Compiles: All land animals are valid input (land animals).

void UseAnimal(LandAnimal&) override;

// Fails to compile: Not All land animals are valid input (dogs).

void UseAnimal(Dog&) override;

};

\*\*Default arguments and virtuals：

出于效率原因，默认参数在编译时确定，

因此默认参数需要使用函数的静态类型

当覆盖虚函数时避免使用默认参数

\*\*Construction of derived classes：

在构造派生类之前始终构造基类。基类永远不依赖于派生类

派生可能依赖于基类

class Animal {...}

class LandAnimal: public Animal {...}

class Dog: public LandAnimals {...}

Dog d;

// Dog() calls LandAnimal()

// LandAnimal() calls Animal()

// Animal members constructed using initialiser list

// Animal constructor body runs

// LandAnimal members constructed using initialiser list

// LandAnimal constructor body runs

// Dog members constructed using initialiser list

// Dog constructor body runs

\*\*Destruction of derived classes：

Easy to remember order: Always opposite to construction order

相反顺序

class Animal {...}

class LandAnimal: public Animal {...}

class Dog: public LandAnimals {...}

Dog d;

// ~Dog() destructor body runs

// Dog members destructed in reverse order of declaration

// ~LandAnimal() destructor body runs

// LandAnimal members destructed in reverse order of declaration

// ~Animal() destructor body runs

// Animal members destructed in reverse order of declaration.

\*\*Virtuals in constructors：

If a class is not fully constructed, cannot perform dynamic binding

class Animal {...};

class LandAnimal: public Animal {

LandAnimal() {

Run();

}

virtual void Run() {

std::cout << "Land animal running\n";

}

};

class Dog: public LandAnimals {

void Run() override {

std::cout << "Dog running\n";

}

};

// When the LandAnimal constructor is being called,

// the Dog part of the object has not been created yet.

// C++ chooses to not allow dynamic binding in constructors

// because Dog::Run() might depend upon Dog's members.

Dog d;

\*\*Virtuals in destructors：

If a class is partially destructed, cannot perform dynamic binding

Unrelated to the destructor itself being virtual

原因一样，动态调用不了

～～WEEK10 ： review

\*\*What is the object slicing problem?

When you try and assign a derived class into a variable of type base class,

the base class only has sufficient space for the base class subobject.

Hence, it copies over the base class subobject and completely ignores the derived class' data

~~~~typoe traits:

namespace traits{

template<typename T>

struct is\_a\_pointer{

static constexpr bool value = false;

}

template<typename T>

struct is\_a\_pointer<T\*>{

static constexpr bool value = true;

}

}

template <typename T>

printPointer(T t) {

if constexpr (traits::is\_a\_pointer<T>::value) {

std::cout << \*t << "\n";

} else {

std::cout << t << "\n";

}

}

=========================================

template<typename T>

struct is\_real\_number{

static bool

}

template <typename T>

if (is\_real\_number<T>::value) {

std::cout << "Is real number" << "\n";

}