C++ 教程 & Pytorch Lightning

菠萝芝士焗饭

2023 年 5 月 14 日

目录

1 前言

一些笔记，来自视频 https://www.bilibili.com/video/BV1uy4y167h2/?spm_id_from=333.788&vd_source=a059a118f33728f79abd79e02f8f72d4 只是一些简单记录，代码类的还是得上手实践，就算完完整整的全部记下来，最后也很容易忘记。就像学数学一样不做题不巩固，就不知道这些知识点要怎么用。所以简单过一遍，然后上项目，项目遇到了什么不懂的再回来补充。

2 C++ 基础教程

2.1 C++ 如何工作

首先看一段非常简单的程序：

```
#include <iostream>

// 如果找不到这个函数在其他文件的位置，就会出现链接报错
void Log(const char* message);
// 任何一个C++的程序都需要main函数，是程序的入口
// 注意到mian函数返回了一个整数，当你什么也不写的时候默认返回0
// 但这只对main函数适用，对于其他函数是一定要有对应的返回值
int main()
{
    // <<是一个重载运算符
    // 将hello world推送到cout，然后打印到控制台上，endl是输出回车
```

```
    std::cout << "Hello, World!" << std::endl;
    // get函数是等待我们输入，也是一个暂停函数
    std::cin.get();
}
```

第一句 include 是预处理，因为他在编译之前就处理完了，所有的预处理语句都会用 # 开始。include 表示把后面的文件复制到当前 cpp 文件上。这个文件也叫头文件，因为他写在开头。
main 是这个整个程序的入口，默认返回 0，如果你不写她就默认返回 0，但对于其他函数是需要返回和类型相同的。«表示一个重载运算符，把 hello world 写到输出流，在控制台显示，endl 表示回车。get 表示等待用户输入，可以作为暂停的一种方式。
cpp 文件的处理可以分成几步：

- 预处理阶段，也就是执行头文件，在这里的主要操作就是把这些头文件复制到当前文件中
- 编译阶段，这个阶段会把 cpp 代码编译成机器执行的代码，这些代码会变成.o 的文件形式
- 然后链接，将这些 o 文件链接成一个 exe 可执行文件

通常，我们不会把一个函数写在一个代码里面，比如上面，把 log 函数单独拿了出来。需要在 test.cpp 文件上面单独声明这个函数，程序会自己去找这个函数在哪里。我把 Log 函数写在了 Log.cpp 文件上，那么在第二步编译阶段的时候，就会出现两个 o 文件，这个时候链接程序就会找到这个 Log 函数放进 test 程序里面。
首先编译：g++ -c test.cpp log.cpp ，会生成.o 文件，然后链接，g++ test.o log.o -o main，就是 test.cpp 中找到这个 log 函数的过程包含在其中，最后生成 main.exe，执行即可。

2.1.1 编译

这个过程的具体表现就是生成.o 文件，主要做了两件事：

- 预处理代码
- 创建抽象语法树

首先是预处理的代码，预处理这部分就是复制，把头文件复制到当前文件中去。看一个例子

```
#include <iostream>

int Log(const char* message)
{
    std::cout << "Logging ..." << message << std::endl;
```

少了两个关键部分，return 和}，我们可以在头文件补上。新建一个 my.h 文件，里面补上缺失的内容：

```
    return 1;
}
```

在代码后面补上

```
#include "my.h"
```

就可以运行了。然后就是生成 obj 文件，这个文件可以用 visual studio 可视化，但是我用的命令行直接 g++ 编译看不到，里面都是一些汇编代码，也就是说编译器会把这些代码变成汇编指令，让机器执行。

2.1.2 链接

链接的主要的目的是寻找每个符号和函数是在哪里，链接本身还是比较好理解的，这一节主要解决的是一些链接的问题。

2.1.3 函数重载

按照视频的内容，返回值不同，那么函数就不同，代码应该是能识别出来的。但实际上，在我的测试代码里面，仅仅只有返回值不同的函数是无法进行函数重载的，这个标准应该是新的标准，我以前学的时候返回值不同还是可以的。所以**仅仅返回值不同的函数是不能重载的**。声明 void Log(const char* message)，如果你的引用代码里面是有 int Log(const char* message)，她是会链接到的。

2.1.4 Ambiguity

这个问题是出现在存在多个可以链接到的函数，程序混乱了导致的。比如在 log.cpp 我们存在两个除了返回值其他都一样的函数，那么模型是不知道链接哪一个函数的。当然，这个错误只会出现在我们链接的时候，编译的时候是不会出现的。

以下有几种特殊的 ambiguity 情形。首先我在头文件 my.h 定义了一个 test 函数，然后在 test.cpp 和 log.cpp 引用这个函数，需要引用我们就需要声明，我们直接用

```
#include "my.h"
```

加在这两个 cpp 文件里面，这个时候编译就会出现

multiple definition of 'test()'

的问题，**这个问题就要回到预处理语句的定义上，include 是把预处理头文件复制到当前文件中，所以相当于在两个 cpp 文件定义了两个相同的函数，这样就出现了定义的问题。**如何解决？有两个方法：

- 把 test 函数隐藏起来，对其他文件不可见。使用 static 关键字。static int test() 就没问题了，虽然他们会被复制到各自的 cpp 文件中，但是 static 关键字使得这些函数是只能自己看到，对其他文件来说不存在。
  - 另一种方法是用 Inline 关键字，这样程序并不会把 test 函数当成是一个函数，她会把引用这个函数直接替换成这个函数的内容。比如 test.cpp 引用了 test()，而 test()return 1;，那么她会把 test.cpp 的 test() 换成 return 1; 这样没有了函数调用自然就没有了链接。
- 另一种方法是通过头文件引入。把方法写到另一个 cpp 文件中，然后把这个方法引入到.h 头文件。其实就是用函数声明，只不过把这个函数声明分到了另一个文件里面。

所以.h 的文件里面很多都是用 static 关键字，就是为了反正多重定义。

2.2 C++ 变量

在程序运行的过程中，我们需要为存储在变量里面的数据命名，这个写数据就存储在变量里面。这一节比较简单

- 变量之间的区别就是大小，他们的大小不同导致他们在计算机里面的区别
- 程序中默认了一些约定俗成的规则，比如 char 一般用于存储字符。但实际上字符的存储也是数字，和 ascii 码表对应

常见的几类 char, short, int, long, longlong, bool 等。他们占用的内存都是字节，因为内存只能以字节为单位查询。虽然 bool 只需要一个比特就能完成查询，但是模型查询的最小单位是字节，所以只能分配字节给 bool 变量。

2.3 函数

函数是用来执行特定任务的代码块。使用函数的目的是希望代码的可读性更高，希望代码块的重用性更高。

- 每次调用函数的时候，编译器会生成 call 指令，也就是说编译器会为每一个函数创建一个 stack 结构，把这些变量都推送到 stack 中。
- 所以在运行函数的时候程序会不断在内存中跳跃执行

所以不要创建过多的函数。另外，如果函数在定义的时候明确要求返回值，那么这个函数就一定要返回。

2.4 头文件

主要了解几个问题

- 为什么需要头文件
- 头文件是干什么的
- 什么情况下需要

头文件通常是用来声明某些函数，这样可以在程序中使用。当我们使用一个定义在其他文件的变量的时候，我们需要声明这个函数，而头文件就是用来存储这些函数声明的地方。当然你也可以直接在你需要用到这个函数的地方进行声明，但如果你有 100 个函数要引用，你就要写 100 次，而且如果你需要在其他的 cpp 文件里面也引用这些函数，那么你又要写 100 次。如果使用头文件只需要把这 100 个函数声明写进去，再引用这个头文件就行，只需要写一次。
再头文件开始的时候还加上了一段代码：

```
#pragma once
```

意思就是头文件里面的内容只会被转换成一个编译单元，这是为了防止你再头文件里面定义了某个函数或者类，然后重复引入同一个头文件导致的定义重复。
另一种复制重复定义的方法是

```
#ifndef _LOG_H
#define _LOG_H

#endif
```

ifndef 表示如果后面这个变量没有被定义，那么就编译后面的代码。如果定义了就不编译。
简单来说 **头文件是用于声明代码，而 cpp 文件是实现这些声明。**
最后还有两个补充的点：

- <> 和"" 的区别，<> 表示从系统目前搜索，然后再搜索环境变量列出的目录，**但是她不会查找当前目录，差不多就是只会查找系统目录。**"" 是会查找当前目前目录下的头文件，然后再去搜索系统目录，所以"" 是会搜索几乎所有可能存在头文件的地方。所有 <> 一般用来引入系统文件，“” 一般用来引入自己写的文件。
- 在写 C++ 的时候引入系统都文件 iostream，这个文件后面是没有扩展名的，这是为了和 C 语言区分开。

2.5 条件和分支

在写代码的时候，我们需要根据结果或者根据某些条件判断我们应该执行哪些语句。但实际上在代码中添加分支会减慢代码的速度。这部分比较简单，主要讨论一些 char * 和数组的内容。
首先有几个关键 Insight：

- 数组是多个元素的组合，他们的地址的连续的
- 指针的类型为整形 int，永远为四个字节，定义的时候 float * q 表示 q 指向一个 float 类型的数据
- 数组的本质就是首个元素的地址

在本章的示例代码中，定义字符串是通过：

```
const char * s = "Neural Radiance Field";
```

以前学的时候不需要 const 是没问题，可能是标准不同了，右边的 NeRF 字符串是一个常量，类型是 const char *，左边和右边的类型对不上，所以报错了。输出的时候直接针对这个地址输出即可:

```
std::cout << s << std::endl;
```

如果定义成

```
char arr[] = "Neural Radiance Field";
```

也是一样，arr 就是字符串的首地址，输出地址对比可以看到是一样的：

```
char * p = arr;
 std::cout << static_cast<void *>(p) << static_cast<void *>(&arr[0]) << std::endl;
```

2.6 For&while

这两个都是用来表示循环，for 的格式

```
for(int i = 0; i < 5; i++)
{

}
```

定义变量，变量条件，变量自增。如果想到达成无限循环，可以在条件部分补上 true，或者直接不写。如果想要达成等差数列，或者等比数列，你可以直接在变量自增部分改成 i+=2, i+=3, i*=3 等。
while 也是类似：

```
while(i < 5)
{
   i ++;
}
```

每次都会判断是否小于 5，while 之后的括号一定要写上 i 自增或者其他改变 i 的语句，否则就会一直循环。但实际上，同样，要达成无限循环也只需要在条件写 true，或者你对 i 这个变量不处理就行；等差数列和等比数列也只需要对 i 这个变量的自增处理一下。所以 while 和 for 其实是可以互换的。
除此之外还有一个 do while 的循环，do while 和 while 其实区别不大，主要在于 do while 是一定要走一次的，while 可以一次都不走。

2.7 控制流

主要介绍三个控制流语句：

- continue
- break

- return

continue 用于跳过当前循环直接进入到下一个循环。Break 推出整个循环。return 直接结束。

2.8 指针

对于程序来说，最重要的部分就是内存，当写下程序运行的时候，程序会被写入到内存中，控制语句，循环语句本质上也是内存之间的不断跳跃。**每一个变量第一会存储在内存中，为了找到这些变量，我们需要为每一个内存编号，这个编号会由另外一个变量存储，这个存储编号的变量就是指针。**
定义一个指针

```
void* ptr = nullptr;
```

指针无论前面定义什么类型，都是一个整形的指针，前面定义的类型只是表明了她需要指向什么样类型的数据。nullptr 表示目前这是一个空指针。现在定义一个变量，用这个指针指向它：

```
int var = 8;
ptr = &var;
```

可以通过指针访问指向的内存数据，& 可以获得当前变量的地址。
如果我们想要一些内存存储，但是又不知道内容的时候，就采用分配的方法。

```
char* buffer = new char[8];
memset(buffer, 0, 8);
```

new char 是分配一个空间为 8 字节的 char 类型内存空间，返回指向这个空间的指针。memset 是填充这个内存，这里表示用 8 个 0 填充。除此之外，还有指针的指针等。

2.9 引用

引用通常就是地址的伪装，引用和指针很大的区别在于指针可以先定义，然后设置他们称为一个空指针；但是引用不可能创建一个空的引用，必须要引用已经存在的变量。
假设一个已知的变量 a，可以通过

```
int& ref = a;
ref = 90;
```

创建一个 a 的引用 ref，ref 就是 a 的别名，对于 ref 的改动其实就是对 a 的改动，之所以说引用没有内存是因为她和 a 共用地址。其实通常并不是会创建一个新的变量指向 a，通常引用是在函数中使用，希望能直接在函数中修改变量的值。
定义一个自增函数

```
void Increment(int value)
{
    value ++;
}
```

直接把 a 传入进去，是不会改变 a 的值。解决这个问题有两个方法，可以利用指针传递，参数改成输入指针：

```
void Increment(int* value)
{
    (*value) ++;
}
```

需要加括号，不加括号相当于是取出下一个地址的内容，然后不做处理。这样相当于获得内存的地址，直接对地址进行修改。另一种方法就是再方法的参数加上 & 引用，传入一个引用。

2.10 类

这部分开始进入面向对象编程。假设一个场景，我们需要写游戏，编写一个玩家的类信息。当然我们可以直接把玩家的所有信息都写进 main 函数里面，位置 xy，速度 velocity 等等。如果定义另外一个人，又需要重新定义一次位置，速度等等。简单定义一个类：

```
class Player
{
public:
    // 公有变量，意味着可以在任何地方访问
    int x, y;
    int speed;

    void Move(int xa, int ya)
    {
        x += xa * speed;
        y += ya * speed;
```

```
    }

};
```

C++ 的类如果默认设置就是私有的，除了自己谁也访问不了，所有通常会写一些公有的访问函数。

2.11 类和结构体的对比

主要是对比结构体和类的区别，说实话学了类之后很少写结构体了，类的功能比结构体强大很大，完全可以替代结构体的功能。
事实上这两者的区别不大，从技术上讲，两者的区别在于结构体的内容都是公有的，但是类是私有的。当然了结构体也可以定义 private 使得变量函数私有。

2.12 static

static 这个关键字在之前就已经见过，用于限制方法的可见范围。通常写在头文件的都是 static。如果 static 写在类的内部，那么说明这个变量是共享的。
在类外的定义：

```
static int static_variable = 5;
```

这个变量只会在当前的编译单元见到。之前这个做法是用来解决链接冲突的问题。A，B 两个 cpp 文件都引用了头文件.h 的一个 log 函数，预处理代码会先把头文件的内容都拷贝到 A，B 文件中，所以这两个文件实际上是定义了两个相同函数，链接的时候编译器就会疑惑怎么会有两个相同的函数。为了解决这个问题，我们把头文件的函数写成 static，这样辅助到 cpp 文件的函数也是 static，他对其他 cpp 文件不可见，这样就只有一个函数可以链接。至于为什么需要 static，这和类中为什么需要 private 变量一样。

2.13 类和结构体内部的 static

上一节的 static 是定义在方法，变量前面，也就是类别外面的。这节的 static 是定义在类内部。
static 成员只能在类内声明不能初始化，要初始化 static 成员必须在类外初始化。比如创建 Player 类别，在类内定义两个变量 x, y。static 变量初始化一定要在所有代码之前，初始化是在编译器初始化。也就是放在所有程序的外面。
有两点需要注意：

- 静态变量的初始化要在所有代码之前，所有一般在头文件写好然后引用到 cpp。
- 类内静态函数只能访问静态变量。和 python，java 类似，每一个类内函数都会传入一个类的实例，C++ 里面的 this 指针，默认传入，python 传入 self 对象。静态方法没有实例也就是没有 this 指针。

2.14 局部静态

在定义变量的时候，我们需要考虑变量的两个属性，生命周期和作用域。通常限制变量生命周期是变量定义的位置，限制作用域通常由特定关键字，比如 static，inline 等，或者定义的位置决定。
假设函数 add，我们希望作用于一个全局变量上，可以如此定义：

```
int sum = 0;
void sum_f()
{
    sum ++;
}
```

但如果我们不想要让其他函数对次变量修改编辑，我们就可以把这个变量编程静态局部变量：

```
void sum_f()
{
    static int sum = 0;
    sum ++;
    LOG(sum);
}
```

此时 sum 变量就变成局部可见了，仅仅只有 sum 函数可见了。用处还是蛮多的，可以用来 debug，测试该方法跑了几回；也可以用这种方法来实现一个类的功能，因为他可以隐藏变量，相当于定义一个全局变量。
总结一下 static 的用法：

- static 在类外定义，在方法或变量前面，该方法和变量仅仅对于该文件可见
- static 在类内定义，该方法相当于类的单例变量，实例通过. 访问，类通过:: 访问
- static 在方法内定义，相当于把全局变量的可见范围缩减到该方法内部。

2.15 枚举

枚举实际上就是一个数值组合，实际上是一种清洁代码的方法。比如假设日志的级别，我们会给 0，1，2 这些数字赋予一些人为的约定，比如 0 是警告等，这个时候如果用枚举就可以简化我们的书写。

```
  enum levels{
   A, B, C, D, E, F, G, H, I, J, K
};
```

这样就创建一个枚举类,这个类里面定义的 A,B,... 都是 static 变量,完全可以通过 levels::A 访问，或者直接就 A 也能访问，这样能让代码更直观。可读性更强。当然也可以直接创建一个实例，但是这个实例只能填 ABC... 那几个给好的数字。

2.16 构造函数

java 也有这东西。构造函数是用来初始化变量的，在定义一个类的时候，我们有时候需要一些特定的操作去初始化函数，比如 python 中的 init 函数。这个时候如果把他们全部写在变量一起，代码看起来就会很混乱，所以构造函数就可以增加模型的可读性。
以类名为函数名，不需要准备返回值类型，就可以定义一个构造函数。

```
  Player(int x, int y)
   {

   }
```

在实例化一个类的时候就会自动调用。

2.17 析构函数

这个函数是在删除实例的时候调用，也就是在该类释放内存的时候调用。

2.18 继承

类之间的继承是 C++ 的最强大特性之一。我们可以通过继承定义一个互相关联的类的层次结构，这些不同的子类都包含了一个共同的祖先。先定义一个父类：

```
class Player
{
```

```
public:
    // 公有变量，意味着可以在任何地方访问
    static int x, y;
    int speed;

    Player(int x, int y)
    {

    }

    void Move(int xa, int ya)
    {
        x += xa * speed;
        y += ya * speed;
    }

    static void s_print()
    {
        LOG(x);
        LOG(y);
    }

};
```

同时定义一个子类继承他：

```
class Good : public Player{
  public:
    using Player::Player;
  };
```

Good 这个子类会拥有这个父类 Public 的所有变量和方法。using 这条语句是 C++11 的标准，可以用这种方法把父类的构造函数继承下来。所以在 Good 类中不用初始化了。注意：

- 如果一个 function 要求参数是 Entity 类，那么可以输入 Entity 及其子类。但是如果定义了子类，就不能传父类，因为父类有可能没有子类方法。比如定义了 Player 类作为参数类型，那么可以传 Good，但是用 Good 类作为参数类型就不能传 Player。这种其实就多态，自适应各种不同类型。

- 继承父类之后重载函数实际上是重新定义函数，而不是重载某一个父类函数。比如父类有 aabb(int i, int j) 和 aabb() 两个函数，子类继承了下来，子类希望重新覆盖 aabb 函数，于是子类自己定义了一个 aabb 函数，那么父类的 aabb(int i, int j) 也会被隐藏起来。

2.19 虚函数

视频中列举了一种情况，现在存在两个类，他们是继承关系：

```
class Entity{
public:
    void printName(){
      LOG("Entity");
   }
};

class player : public Entity{
   public:
     void printName(){
       LOG("Player");
     }
};
```

编译运行以下语句：void testName(Entity* entity) entity->printName();
int main() Entity* entity = new Entity(); Entity* ply = new player(); testName(entity); testName(ply); 会输出两个 Entity，我们原本是创建了一个 plyer 类但是却被识别成 Entity 的类，使用了父类的函数。**因为通常在声明函数的时候，我们的方法通常是在类的内部起作用，这个也很容易理解，因为这两个指针实际上是经过了强转的，模型是不知道他们的来历的，只能知道他们的类型是什么，除此之外就一模一样了。所以这个时候我们就要想个办法让模型能意识到，当前的指针的来自 player 的。**
虚函数就是解决这个问题，使用虚函数模型会创建一个 v 表，这个表会记录当前虚函数的映射，也就是被哪个函数复写了。每次运行都需要查表，当然这就出现了额外的性能损失。

2.20 接口

接口函数比虚函数更极端，他会强制子类去实现这个虚函数。他不需要实现任何东西，只需要给函数参数就行。

但是继承的子类需要强制实现，类似于给子类定义了一个标准，按照这个标准去声明。

2.21 可见性

这部分比较重要。这里的可见性一般是指类成员的可见性。
之前在提到类的时候，类成员定义的时候不加修饰默认是私有。

- private：私有变量只能当前类，或者友元类能访问。
- protected：相比于 private，他的可见性更强，因为他的子类是能访问到的。

所以为什么要设置可见性？
首先代码里面纯用 public 是一个比较糟糕的做法，因为对于开发者和程序来说，你需要保证你代码的安全性，比如在管理数据库的时候你需要写好 API 提供给你的程序员调用，如果全部开放很容易就出现删库跑路或者数据泄露的问题。

2.22 数组

数组和指针息息相关，数组就是指针的基础，可以通过对于指针的操作对数组直接操作。
创建数组有两种形式：

- 直接调用 a[n] 创建数组。这种方式是在栈上创建，到达第一个花括号就会自动释放。
- 调用 new 创建，那么会创建在堆上，不会自动释放，除非关闭了程序。

这两者最大的区别就是生存周期，其次还有一定性能的区别，new 创建的数组实际上是用指针去接受了这个数组的第一个元素的地址，然后再根据这个地址寻找其他变量。这是一种间接寻址，而在栈上创建是直接寻址。
视频后面讲了一堆，感觉没有什么记录的必要，主要需要记录的是栈数组初始化的时候：

```
class Entity{
   public:
      const int size = 5;
      int example[size];
};
```

这种初始化是错误的，需要在 size 前面加上 static，因为编译器不能确定 example 初始化前，你这个 size 和 array 哪一个先创建，很可能是同时创建的。如果你直接在 main 函数，或者在其他函数这样创建是没有问题的，

```
int main()
{
    const int size = 5;
    int example[size];
}
```

最后提了一嘴 STL。

2.23 字符串

另一种数组，只不过用来表达字符串而已。

```
   const char* name = "new array";
    std::cout << name << std::endl;
```

定义了 const 之后就不能对模型进行改变，定义了 name 并且赋值之后，这个字符串的最后会补上一个 0，表示字符串的终结，这也是为什么直接输出 name 能完整的输出字符串。如果用数组定义，那就需要显示的补上 0，否则就会出现乱码：

```
    const char name1[10] = {'n', 'e', 'w', ' ', 'a', 'r', 'r', 'a', 'y', 'a'};
    std::cout << name1 << std::endl;
```

这样就会出现乱码。
另外 STL 也有 string，可以直接用，不够 C++ 对编译器优化了很多问题已经不存在了。

2.24 const

const 就像一种承诺，定义的时候就承诺定义的变量我们就不会改变，但是既然是承诺那也能违背，所以其实是可以通过某种方法去绕过这个变量的。当我们定义一个变量 a = 5，可以随意改变这个变量的值，但如果前面加了 const，那就不能修改这个变量的值了。
首先是 const 和指针搭配使用：

- const int和 int const是一个意思，可以改变指针的指向，但不能改变指针指向的内容，也就是能使得指针 b = &a 某个变量，但是不能b = a。
- int const 能改变指针的内容，但是不能改变指针的指向。
- 放在方法的后面，**这种做法只能在类中使用，int getX() constreturn x，意味着你不能修改类的变量**。通常在需要 print，查看类信息的时候使用。

极端点的情况：

```
const int* const getX() const
  {
    return &x;
  }
```

这个函数有三个 const，当然是在类里面定义的函数，返回的指针不能修改指向，不能修改内容，并且这个函数也不能修改类变量。
突然发现设计一个好的语言真的很烦，随便一个改动就要考虑各种各样的情况。有一个需要调用这个 getX 类内函数的普通函数

```
void print_Get(Entity& entity)
{
  entity.getX();
}
```

这样调用是没问题的，getX 的 const 是限制类内函数和外面的环境没有关系。但是如果我在参数加上 const，那么 getX 就不能把后面的 const 留下了，也就是说这样是错误的：

```
class Entity{
   public:
       int x = 4;

       static const int size = 5;
       int example[size];

       const int* const getX()
       {
          return &x;
       }
};

void print_Get(const Entity& entity)
{
  entity.getX();
}
```

因为输入 print_Get 函数的时候你已经要求了 entity 不变，但是如果 getX 不加 const，是

可以改变的，那么这个 const 就失去了意义。所以这个 getX 就需要加 const。所以经常会看到两个版本的 getX 就是在这个原因。
但如果你确实是想做一些标记，希望能在 const 的情况下继续修改，那么就需要 mutable 关键字，这样你的变量就可以修改了。总之 const 就类似一些强行施加的规定，保证代码的安全性。

2.25 成员初始化列表

主要是对类成员初始化所用。最常见的一种初始化方式就是构造函数了，定义不同的构造函数可以使用不同的方式进行初始化。
另一种初始化就是成员列表初始化，在构造函数后面加上冒号按照顺序写好：

```
class Player{
    public:
        int x, y;
        int velocity;

        // 列表初始化需要按照声明变量的顺序来写，顺序不能乱
        Player(): x(0), y(0), velocity(0)
        {

        }

        Player(int x, int y, int velocity): x(x), y(y), velocity(velocity)
        {

        }
};
```

所以为什么我们需要使用这个，成员列表初始化和一般的初始化方法其实结果都是一样的，完全可以互相替代，但是如果使用成员列表初始化，就可以对代码进行简化，当你有很多变量的时候，一般的初始化会使得构造函数很难看出是在做什么，
但如果你用成员列表初始化，构造函数就会很清晰。初次之外，使用成员初始化还能增加程序的性能。假设类内构造函数初始化如下图所示：

```
    int x, y;
    int velocity;
    Entity* entity;
```

```
// 列表初始化需要按照声明变量的顺序来写，顺序不能乱
// Player(): x(0), y(0), velocity(0), entity()
// {

// }
Player()
{
   entity = new Entity(8);
}
```

那么 Entity 会创建两次，第一次是定义的时候，第二次是构造函数声明的时候，如果用构造成员列表，那么只会构建一次：

```
Player(): x(0), y(0), velocity(0), entity(Entity(7))
{

}
```

最后的结论就是尽量使用成员列表初始化。

2.26 三元操作符

三元操作符实际上是一种简化代码的形式，if 语句的一种语法糖。正常 if 的写法：

```
if (level < LEVEL){
    level = LEVEL;
 }
```

用语法糖可以一行写完：

```
level = level < LEVEL ? LEVEL : level;
```

有两个优点：

- 代码更简洁
- 速度更快，因为他不会创建一个临时的字符串作为比较结果

2.27 创建以及初始化类

首先是栈和堆的区别，内存主要也是分成两部分，堆和栈，栈的生命周期是和他在哪里声明相关，如果在函数中声明，那么函数结束的时候就会把这些变量都弹出来删除。堆不一样，堆内的对象不会自动释放，而是会等待你做出释放的命令，否则他会一直停留到结束。对于类来说，默认初始化就是在栈上创建，new 关键字创建就是在堆上创建。

2.28 new 关键字

如果希望用到 C++，一般都要关注于内存，性能，并行计算和优化问题。Java，python 这类语言带有内存清理的机制，C++ 没有，需要自己控制内存。
使用 new 初始化类，是一种非常常见的操作了。new 实际上是一个操作符，她不仅仅分配了空间，她还调用了类的构造函数。**new 实际上是调用了 c 语言的 malloc 分配函数**

```
Entity* e = new Entity();
Entity* e = (Entity*)malloc(sizeof(Entity));
```

上面两行的区别在于第二行只是分配了空间，而第一行不仅分配了空间，还调用了构造函数。**在调用 new 操作符的时候，需要手动清理内存空间，因为在堆上的变量是不会自动释放的。所以如果使用了 new，最后要接上 delete。**

2.29 隐式转换和 explicit 关键字

首先作者举了一个类的例子，对于有构造函数

```
Entity(int a)
{
   cout << "create " << a << endl;
}
```

的类，我们可以直接对她进行赋值，Entity a = 23;。这里会自动做一次隐式变换，**隐式变换只能做一次，如果需要两次变换的会报错。**比如 Entity a = "67" 就有问题，她需要两次变换，首先变成 string，然后变成 int，再变成 entity。这种代码的书写方式可读性不高，所以如果想禁止这种方式可以使用 explicit，只需要再构造函数之前加上 explcit。
隐式变换在书写代码的过程中处处都有，比如对于一个函数参数为 double 类型的函数，如果调用时候输入 1，那么她会把 1 变成 double 类型，这就是一种隐式变换。

2.30 运算符重载

- 首先什么是运算符：运算符是一种符号，可以用来代替函数进行执行，常见的数字运算符加减乘除，new，delete 或者问号都是运算符。

看一个简单的重载例子，定义一个类

```
class Vector_two{
    public:
        int x, y;
        Vector_two(int a, int b):x(a), y(b) {}
};

Vector_two speed(1, 3);
Vector_two position(2, 4);
```

如果想要把这两个类对应的元素相加，可以写一个类内的函数调用，另一种比较方便的写法是重载 + 运算符。

```
   Vector_two operator+(const Vector_two& b) const
    {
        return Vector_two(x + b.x, y + b.y);
    }

    Vector_two operator*(const Vector_two& b) const
    {
        return Vector_two(x * b.x, y * b.y);
    }
```

重载加号和乘号的运算符。
这是在类内的重载，我们也可以重载 «

```
ostream& operator<<(ostream& stream, const Vector_two& other)
{
    stream << other.x << ", " << other.y;
    return stream;
}
```

2.31 对象生存期

- 如何理解栈上的创建的对象？
- 如何写出高效的代码？

栈可以认为是一种数据结构，这种数据结构的特点就是，她会在作用域结束之后删除，如果你在一个函数里面不用 new 方法创建了一个类，然后返回指向这个类的指针，那么在函数结束之后这个类就会被释放，返回的指针指向了一个空区域。

2.32 智能指针

这算是一个新东西了，在使用关键字 new 创建一个新的变量的时候，我们需要配套使用 delete 删除这个内存，这样相对来说比较麻烦，而智能指针的对这一过程简化。引入 memory 头文件，

```
int main()
  {
    {
      // std::unique_ptr<Entity> entity(new Entity());
      // 这种方式最安全
      std::unique_ptr<Entity> entity = std::make_unique<Entity>();
    }
  }
```

这样就创建一个 entity 对象，跳出这个作用域那么智能指针会自动删除。但是智能指针是不能复制的，也就是说当智能指针指向了这个内存后不能有其他指针指向了，因为如果智能指针删除了当前变量，那么另一个指针就没有效果了。
shared_ptr 是另一种指针，她的释放内存方式和 python 的差不多，都是计算内存指向的指针数目，如果是 0 那么就释放。shared_ptr 解决了 unique_ptr 不能复制的问题。weak_ptr 是 shared_ptr 的一种，她不会增加 shared_ptr 的引用次数。

2.33 C++ 复制与拷贝函数

在 python 中复制还分深度拷贝还浅度拷贝，其实这里说的就是这两个概念。创建两个变量:

```
  int a = 3;
   int b = a;
```

这种就相当于深度复制，a 和 b 相当于两个不同的变量，只不过数值相同而已。如果我用指针复制：

```
int* a = new int();
int* b = a;
```

那 a 和 b 指针相当于共享内存了。
看一个简单的例子：

```
class String
{
 private:
    char* m_Buffer;
    unsigned int m_Size;
 public:
    String(const char* p)
    {
      this->m_Size = strlen(p);
      this->m_Buffer = new char[this->m_Size];
      memcpy(m_Buffer, p, this->m_Size);
    }

    // 声明为友元重载，可以访问私有变量
    friend std::ostream& operator<<(std::ostream& os, const String&);
};

std::ostream& operator<<(std::ostream& os, const String& string)
{
 os << string.m_Buffer;
 return os;
}
```

在这个类中，我们声明了一个字符串，一个友元重载，因为我们需要访问私有变量。在运行输出的时候由于没有终止符，会出现一些错乱的输出，我们只需要在 m_Size 改成 m_Size + 1 就行，memcpy 多余的字符会用 0 填充。
但是如果我们创建另外一个变量，这个变量 string 赋值，那就会出现问题：

```
String string = "new code";
String second = string;
```

```
std::cout << string << std::endl;
```

因为这个时候复制的变量是指针，其中一个指针释放了内存另一个又要释放，完了找不到内存就崩溃了。这种只是浅拷贝，要解决这个问题，就需要构造一个拷贝构造函数。

```
String(const String& other): m_Size(other.m_Size)
{
    this->m_Size = strlen(other.m_Buffer);
    this->m_Buffer = new char[this->m_Size + 1];
    memcpy(m_Buffer, other.m_Buffer, this->m_Size + 1);
}
```

相当于重新分配了一个内存给这个变量，这个时候

```
String string = "new code";
 String second = string;
```

就相当于是两个不同的变量了。
另外一个点就是，形参也是会重新复制一份，所以如果不需要复制，那就用引用就行，这样就不会复制，而是直接把指针传过去。**尽可能使用 const 去传递参数，没必要到处去复制，可以在程序内部决定要不要复制。**

2.34 箭头运算符

我感觉是不是顺序有问题，这个内容应该就在前面就讲。
总的来说就是指针用箭头。这节不打算记录了，efficients C++ 里面不太建议这样写。
最后作者还补了一个语法糖，简单看了一下：

```
class Vector3
{
    public:
        float x, y, z;
};
int offset = (long long)(&((Vector3*)nullptr)->z);
```

offset 就是偏移量。首先 nullpter 是空指针，强转成 Vector* 就是初始化了一个指针。然后-> 得到类内元素，& 取得当前元素的地址，再强转成 long long。作者是转成 int，但是我这出问题，long long 是可以隐式转 int 的。nullpter 就是存储在 0 地址中，所以 xyz 是从 0 开始分配。

emmm 这种代码我也不会写，就放着吧。

2.35 动态数组

也就是 vector。一般的数组在规定了她的大小之后就不能再增加元素个数了。但是 vector 是可以不断增加元素个数的。for 循环可以遍历：

```
for(Vertex& x : vertices)
{

}
```

注意还是尽量使用引用，避免出现复制。
在 vector 的使用中，有两个部分是可以进行优化的：

- vector.push_back 添加元素的时候，是现在 main 函数内创建，然后再移动进 vector 里面，在移动的过程中就相当于是复制了。这里复制了一次。
- vector 是动态数组，每一次增加都需要扩充容量，每一次扩充容量就相当于把原来的数组的复制一遍。

解决方法都很直接

```
std::vector<Vertex> vectices;
vectices.reserve(3);
```

直接将容量变成 3，这样就不用动态扩充了。
第一个问题用 emplace_back，她并不会创建变量然后推进 vector 中，而是会把创建的参数丢进 vector 里面，然后再 vector 里面创建。所以尽量还是用 emplace_back 创建吧。

2.36 处理多返回值

处理方式有很多种

- 可以用数组或者 string 来接收这些返回值然后返回，包括各种数据结构了。
- 也可以定义一个 static 函数，然后引用接收参数

2.37 templates

模板，就是设计一套规则，让机器帮你写代码。比如需要输出不同类型的变量，对于一个 print 函数可能要写很多次，对于不同的变量都需要进行重载。

```
template<typename T> void print(T value)
{
    std::cout << value << std::endl;
}
```

类型并没有指定，等待人为指定或者自动的隐式变换指定。一开始第一次编译碰见这个函数的时候并不会定义出来，知道编译到调用这个函数的位置才会创建，也就是只有调用这个函数的时候才会正式创建这个函数。

```
   print(5);
    print("cahjks");
```

当然也可以自己指定。

```
   print<int>(5);
    print<String>("cahjks");
```

如果在类上使用模板呢？

```
template<int N>
class Array
{
    private:
        int m_Array[N];
    public:
        int GetSize() const
        {
            return N;
        }
};
```

在声明的时候并不会报错，编译器会根据使用情况进行编译。类定义的变量类型可变可以通过模板定义。前面加两个模板就行：

```
   template<typename T, int N>
```

2.38 栈和堆的比较

当我们请求内存分配的时候，栈和堆的内存分配方式是不一样的，
栈的分配速度很快，栈的分配只需要移动栈指针即可，分配四个字节的内存，只需要将栈指针移动四个单位即可。所以栈的内存都紧挨着的。
对于堆的分配一般是用 new 关键字，new 关键字会调用 malloc 分配内存。当启动这个程序的时候，os 会分配一些内存给你，程序会去维护一个空闲列表的数据结构，里面记录了哪些内存是空闲的，哪些不是空闲的。当使用 malloc 分配内存的时候，他会寻找这个空闲列表，找到一些空闲的块，分配给程序。**当空闲列表中内存不够的时候，就需要向 os 请求，这个过程是很麻烦的。**
这两种分配方法最大的区别就是在于分配方法的速度，在栈上分配的速度非常块，只需要一条 cpu 的指令，就可以快速分配。堆的分配会慢一些。所以**尽量在栈上分配，除非不能或者需要更长的生命周期。**

2.39 宏

一开始的时候就提到过宏，比如 define，if 都是，是在预处理阶段处理的语句。

```
#define WAIT std::cin.get()
```

注意到这里没有加;，那么你在调用的时候就需要写 WAIT;，因为他是替换这行。如果 define WAIT std::cin.get() 写了; 号，那么 WAIT 就不用写了。就像前面 include 把 { 复制进来一样。
define 就是把 WAIT 定义成后面那句话，在预处理的时候，他会把 WAIT 语句替换成后面那句话，和 inline 做一样的事情。**这种方式比较蠢，不推荐这样做，因为这样会导致可读性不强。**
在一些特定的情况，宏还是有用的，比如日志的输出，在 debug 阶段是需要输出各种日志，但是在运行阶段我们就不希望输出了，这个时候可以如下定义：

```
#ifdef PR_DEBUG
#define LOG(x) std::cout << x << std::endl;
#else
#define LOG(x)
#endif
```

如果是 debug 模型，把 PR_DEBUG 定义出来，这样就会输出日志，否则就不输出。**宏必须是要在同一行，所以如果一行写不下，可以用**

跳到下一行写。

2.40 auto 关键字

auto 让模型自动推断变量的类型。emmmm 说实话我一般不会经常用 auto，除非是接收函数返回的时候，因为这会严重影响可读性。

2.41 静态数组 array

和动态数组 vector 相对应的，他一出生就决定了他的大小，不能改变 size 了。

```
std::array<int, 5> data_array;
```

这样定义的静态数组有有个问题：不知道他的个数，没办法传递参数到函数里面，因为传递函数需要把这个静态数组的完整类型写出来。这个时候可以使用模板

```
template<typename T> void print_array(const T& value)
{
   for(int i = 0; i < value.size(); i++)
   {
      LOG(value[i]);
   }
}
```

我原本想着用 auto 让模型自行推断，但是不行。相比直接创建的数组，std::array 数据结构有边界检查。**应该尽量用 std::array 替代 C 语言的数组：**

- 首先他提供了一系列的安全检查
- 其次并没有声明性能损失，还能记录数组大小。

2.42 函数指针

获得函数指针之后可以通过指针调用这个函数

```
auto function_pointer = hello_world;
function_pointer();
```

function_pointer 的类型是 void(*function_pointer)，function_pointer 只是一个名字，可以随意取。

定义一个函数指针有两种方法

- 一种是直接定义，void (*fun_name) (int); 直接定义一个 fun_name 的函数指针，fun_name = hello_world; 将这个函数 hello_world 赋值给函数指针。
- 第二种是先把函数指针类型定义出来 typedef void (*fun_name) (int); 再根据这个类型把变量定义出来 fun_name fun = hello_world；第二种方式对后续使用更友好。
- 第三种就是直接 auto 定义了。

函数指针可以让函数进行参数的传递

```
  void print_array(const T& array, void (*prt_array)(int))
{
   for(int i = 0; i < array.size(); i++)
   {
      prt_array(array[i]);
   }
}
```

直接把函数名称传递进去就行。

2.43 命名空间

为什么不用 using namespace std？这里说的是作者为什么不用，我经常用其实。
作者不爱用这玩意的原因：

- 显示的写出这些命名空间，可以知道是使用哪一个库的内容。比如使用 std 就能知道是再 C++ 标准库使用的。
- 可能出现命名空间的冲突，如果两个命名空间里面都有同名称的函数，很容易就冲突了

为什么要使用命名空间？这是下一节的内容。

- 简单来说，就是为了解决在大工程中容易出现命名重复的问题。
- 命名空间是在限定作用域下进行。

2.44 线程

目前我们所完成的代码都是单线程的，一次只能做一条指令。但实际上多线程在日常中很常见，比如在制作游戏的时候，我们通常需要等待用户输入指令，但是如果主线程等待用户输入指令，那么整个游戏就会停下来，所以我们需要其他线程来接受用户指令，主线程运行。

main 就是一个主线程，定义一个函数

```
void DoWork()
{
    while(!is_finishing)
    {
        std::cout << "working" << std::endl;
    }
}
```

在主函数中，我们定义一个线程去运行这个函数

```
    std::thread worker(DoWork);
```

这样我们就定义一个线程去运行这个函数，但是只是定义还没开始

```
    std::cin.get();
     is_finishing = true;
```

表示输入任一按键推出子线程。

```
    worker.join();
```

开始运行线程，并且主线程会等待子线程运行完毕。还有另一个 detach，不过这个是主线程不等待子线程运行完毕。

2.45 多维数组

多维数组是数组的数组，把一个数组里面的元素换成数组即可。其实就是创建了很多很多个数组，然后把指向这些数组的指针收集再一起，就变成了二维指针。多维数组就相当于多层指针的意思。

```
    int** a2d = new int*[50];
```

a2d[0] 是一个指针，指向的是个数组。上面那行代码我们只是初始化了一系列指针，并没有实质分配空间，需要用 for 循环挨个挨个分配：

```
for(int i = 0; i < 50; i++)
 {
    a2d[i] = new int[50];
 }
```

删除的时候不能直接用 delete 删除，如果直接 delete[][] a2d，你只是删除了这一堆指针，而指针指向的内存并没有删除，删除也是需要循环删除，否则会造成内存泄漏。
二维数组如果这样初始化是会存在内存不连续的问题，因为每一个指针之间的内存不是连续的，导致运行效率的问题。up 提到了个 cache miss，意思就是每一次系统会把当前访问内存的一些邻居内存的内容也收进 cache 里面，因为系统认为相邻的内存的访问时空频率会相近。如果内存不连续，那么这样的假设不成立，会造成频繁的 cache miss。
二维数组不是连续的，那么一维数组是连续的吧？用一维度数组去模拟二维数组就行了。这个比较简单，常见操作了。

2.46 排序

用 STL 了基本上就是。最简单的用法

```
std::vector<int> values = {3, 5, 4, 2, 1};
 std::sort(values.begin(), values.end(), std::greater<int>());
 for(int value : values)
 {
    std::cout << value << std::endl;
 }
```

第二行排序，按照降序排列。当然也可以自定义，这比较容易。后面再看很多代码的时候大部分都是自己实现的，因为不同排序在不同情况下效率不同。

2.47 Union

定义多个变量占用同一个内存，大小为最大的那个变量。定义一个结构体

```
struct test
{
union
{
```

```
    struct
    {
      float a, b, c, d;
    };

    struct
    {
      Vector v1, v2;
    };


  };

};
```

如果修改 c 和 d 的值，会发现 vector v2 的 xy 的值也会随之修改。这个 Union 想要达成的目的是想设置一个变量能够存储多个不同类型的变量，不用定义多个类型。

2.48 虚析构函数

这个问题和之前的虚函数一样，都是解决编译器无法识别内存实际存储变量类型的问题。

```
  class Base
{
   public:
       Base() {std::cout << "Base" << std::endl;}
       ~Base() {std::cout << "Delete Base" << std::endl;}
};

class Derived : public Base
{
   public:
       Derived() {std::cout << "Derived" << std::endl;}
       ~Derived() {std::cout << "Delete Derived" << std::endl;}
};
```

定义一个类，另一个继承。在初始化的时候，显示调用的构造函数，模型自然是知道这个类就是 Derived，但是删除的时候模型并不知道这个内存里面的内容是哪一个类，只能通过类型判断，这个时候就需要使用虚函数了。**如果析构函数不对，会出现内存泄漏。**

只需要在析构函数上上 virtual 即可，他会创建一个新表记录。

2.49 类型转换

主要还是显示类型转换，定义 double 变量，如果强行赋值给 int 变量，编译器会默认使用隐式转换，因为这样会有精度丢失，当然可以指定显示转换 static_cast 等。这些 cast 会自带编译器的一些安全检查，会有一定的速度损失。
作者后面给了一堆例子，没看太懂，暂不记录。cast 的类别有四类

- static_cast: 针对的是精度算是较小的转换，比如浮点和整数，不能进行指针的转换
- reinterpret_cast: 用于不同类型指针的强转，常用于继承类之间的转换
- const_cast: 去除 const 属性
- dynamic_cast: 不带安全检查，仅仅只在运行的时候进行检查，如果失败返回 null。这个可以用来判断一个类是否是其子类，当然也有 instance 可以用。

2.50 Safe

C++ 编程中的安全是指降低崩溃，内存泄漏，非法访问等问题。**随着 C++11 的到来，更应该转向智能指针而不是原始指针。这是作者说的。**

- 内存泄漏问题：在一个堆上分配内存，如果不人为进行删除，那么这个内存将一直存在。如果存在循环的话这个内存就会一直分配。
- 内存清理的所有权问题：由谁来清理管理内存。

这集 up 估计是被评论喷了回一波，感觉没什么内容在里面，过。

2.51 预编译头文件

主要解决的问题就是编译速度过慢。每一次在头文件引入 vector，string 这些头文件的时候，都需要重新对 vector，string 这些头文件代入的代码进行重新编译，速度会慢很多。预编译头文件就是引入他们的二进制格式的文件，这样会加快编译。

2.52 dynamic_cast

类型转换四种之一。C 语言的类型转换直接在变量前加上类型即可，C++ 的类型转换有四种 static_cast，dynamic_cast，reinterpret_cast，const_cast。

dynamic_cast 只适合多态类之间的转换，可以通过这种方式判断一个当前的实体是否是可以转换的，或者确认当前实体是否是某个类的子类。
编译器是如何知道不能转换的呢？因为在运行的时候编译器存储了类运行时的信息，RTTI。这是会增加开销，

2.53 结构化绑定

作者提到了常用的接受多个返回值的方法。用 tuple 接收：

```
  std::tuple<std::string, int> CreatePerson()
{
   return {"LY", 24};
}
```

这个 tuple 接收两个返回值，一个 string，一个 int。一种是用 get 取出值：

```
  auto person = CreatePerson();
   std::string& name = std::get<0>(person);
   int age = std::get<1>(person);
```

另一种是 tie，相比来说 tie 更整洁

```
  std::string name;
   int age;
   std::tie(name, age) = CreatePerson();
```

如果使用结构化绑定，不需要先定义两个接收变量的类型，直接使用即可：

```
  auto[name, age] = CreatePerson();
```

不过这个特性是 17 后生效。现在 22 年了大家基本都是大于 17 版本肯定生效的。
后面两章介绍 17 新特性的，先跳过。

2.54 如何能让 C++ 字符串更快？

前面有提到过，尽量在栈内分配，避免在堆上分配。string 就是在堆上分配的，但是我阅读了一下 string 的源码，如果是在 16 字节内，他就是在换存池里面分配，也就是栈，如果是大于 16 字节那就是在堆上分配。事实上这也是很自然的，毕竟栈内存在编译后就确定了，如果是动态变量需要很大的内存空间，栈是满足不了的。

```
std::string name = "hajksdhjkkqwey";
 std::string first = name.substr(0, 3);
 std::string second = name.substr(4, 9);
```

这三步每一步都会分配内存，一共在堆上分配了三个内存。我们并不希望他分配内存，只需要能 access 到这个 name 就可以。
string_view 可以达成，string_view 只是记录了当前 string 的指针和偏移量，也就是说他只是 string 的一个指针类型，相比 string 需要分配更大的空间，string_view 明显效率更高。

```
std::string_view first(name.c_str(), 3);
 std::string_view second(name.c_str(), 4);
```

c_str 返回一个指向当前字符串的指针，比如 char* c = a.c_str()，本身就是用来兼容 c 的。这样就只需要分配一次了。如果要把剩下的这个去除，只需要把 string 修改即可:

```
const char* name = "hajksdhjkkqwey";
 // std::string first = name.substr(0, 3);
 // std::string second = name.substr(4, 9);

 std::string_view first(name, 3);
 std::string_view second(name, 4);
```

这样一次分配都没有。

2.55 单例模式

这应该是常见的一种设计模式。当我们想让当前数据共享于所有的类或者结构体的时候，并且我们希望保证数据的一致性，单例模式就非常有用。比如渲染器就是一种全局通用的单例，不会因为不同物体就采用不同的渲染器。当然渲染也分很多种，有体渲染，也有面渲染，但他们都只需要创建一个就行。
这个比较简单，基本上面试笔试都要写。主要就是满足对于每一次调用都只能共用一个实例就行。创建

```
class Singleton
{
   public:
      Singleton(const Singleton&) = delete;
```

```
        static Singleton& Get()
        {
            return s_Instance;
        }
    private:
        Singleton() {}
        static Singleton s_Instance;
};
```

构造函数放在 private 是不允许显示调用，public 中还需要静止复制，在 private 初始化然后 public 写个窗口供调用就行。

2.56 小字符优化

小字符优化也叫 SSO，字符占用字节如果小于 16 字节，那么会在栈的缓冲区分配内存，如果超过了，就会堆上返回。这里的源码不一样，要找 if CXX14 的源码看，他看的是旧版源码，新的用的 construct 分配。
在 xstring 源码可以看到

```
  if (_Count < _BUF_SIZE) {
    _My_data._Mysize = _Count;
    _My_data._Myres = _BUF_SIZE - 1;
    if constexpr (_Strat == _Construct_strategy::_From_char) {
      _Traits::assign(_My_data._Bx._Buf, _Count, _Arg);
    } else if constexpr (_Strat == _Construct_strategy::_From_ptr) {
      _Traits::move(_My_data._Bx._Buf, _Arg, _Count);
    } else { // _Strat == _Construct_strategy::_From_string
```

小于特定的 buf_size 的时候不会分配堆内存。

2.57 跟踪内存分配

主要还是为了性能服务，在写安全协议的时候也用到，不过是借助了其他语言工具。
就是自己写一个能记录内存分配的类或者工具，感觉还是用给定好的包或者库好用。

2.58 左值和右值

最简单的一个例子

```
int i = 10;
```

左值一般在左边，右值一般在右边。但这种规则并不总是适用，比如 int a = b 这就不适合，两个都是左值。

- 左值一般来说是可寻址，有一定内存指向的变量。右值一般是临时变量。
- 赋值给左值是可以的，给右值不行，而左值是要非 const 可修改的。
- 左值引用，只能是给左值。
- const 左值引用可以给右值，const int& a = 10。实际上是创建了一个临时变量。**所以 const 左值引用是兼容左值和右值的。这也是为什么 string 的调用通常使用 const 的原因，因为兼容左值和右值。**

```
std::string a = "sjkdf";
 std::string b = "aksla";
 std::string c = a + b;
 a + b;
```

c 是左值，a+b 是右值。因为在 c = a + b 的过程中调用了 string 的构造函数，而 a + b 没有，就是一个右值。

2.59 参数计算顺序

跳过了，更考试题一样，这节的内容谁写进工程代码里面我砍谁。

2.60 移动语义

为什么需要移动语义？

- 通常我们只是希望创建出一个临时变量给构造函数，如果在主进程创建一个变量然后传到构造函数，然后构造函数会复制这个变量。

一个简单的例子

```
class String {
public:
 String() = default;
 String(const char* string) {
```

```
        printf("Created!\n");
        m_Size = strlen(string);
        m_Data = new char[m_Size];
        memcpy(m_Data, string, m_Size);
    }

    String(const String& other) {
        printf("Copied!\n");
        m_Size = other.m_Size;
        m_Data = new char[m_Size];
        memcpy(m_Data, other.m_Data, m_Size);
    }

    ~String() {
        delete[] m_Data;
    }

    void Print() {
        for (uint32_t i = 0; i < m_Size; ++i)
            printf("%c", m_Data[i]);

        printf("\n");
    }
private:
    char* m_Data;
    uint32_t m_Size;
};

class Entity {
public:
    Entity(const String& name)
        : m_Name(name) {}
    void PrintName() {
        m_Name.Print();
    }
private:
    String m_Name;
};
```

创建变量 entity 的时候，传入 entity(string("a"))，首先创建 string，然后在构造函数又复制一次。我们可以强制他传右值，把 string 的构造函数去掉。增加

```
String(String&& other) {
    printf("Moved!\n");
    m_Size = other.m_Size;
    m_Data = other.m_Data;
    other.m_Data = nullptr;
    other.m_Size = 0;
}
```

entity 构造函数也改成只接收右值

```
Entity(String&& name)
    : m_Name(name) {}
```

但实际上还是没能解决问题，因为右值传进来的时候就已经退化了，在 entity 的构造函数传递 name 的时候就变成了左值，这个时候用 std::move 即可。**move 实际上是移动资源，有时候需要额外操作防止内存泄漏**。所以通常需要首先删除当前变量资源，然后赋值，然后删除赋值变量的资源。

3 C++ 小游戏

https://www.bilibili.com/video/BV14z4y1r7wX/?spm_id_from=333.999.0.0&vd_source=a059a118f33728f79abd79e02f8f72d4 的教程，自己的理解以及一些改进。

3.1 俄罗斯方块

创建方块，这里一共 7 个，作者用的 wstring，wstring 每一个字符占用 2 个字节，我看他做的都是字母，感觉 string 应该也行。

```
std::wstring tetromino[7];
```

然后挨个挨个赋值就行。

3.1.1 坐标

坐标这个问题比较重要，他涉及到碰撞检测，移动，显示等的问题。正常的坐标在左上角开始，xy 两个方向 0 开始增加：

```
/*
0 1 2 3 4 5
0
1
2
3
4
5
*/
```

作者用一维数组表示，每一个小格子用一个 index 表示，从左上角开始 0 一直编号下去

```
/*
0 1 2 3 4 5
6 7 8 9 10 11
*/
```

对于在 (x, y) 的点，用一维数组表示就是 index = y * w + x。当他顺时针旋转 90 度 index = 12 + y - (x * h)，当顺时针旋转 180 度 index = 15 - (y * w) - x，当顺时针旋转 270 度 index = (y * w) + (h - x - 1)

3.2 前期工作准备

没有写过 win，这些 API 看的头疼，只能一个个查了。首先区域分为三个区域

- 旋转区域：这个区域是方块按照玩家指示旋转的，上面提的顺时针旋转这些就是。

  这个区域作者没有写出来，只是约定俗成了就是 4x4 我还以为是整个屏幕坐标的，所以那四条公式我都写成 w 和 h。但实际上就是再 4x4 的格子进行旋转。

- 玩家区域：也就是游戏区域，一整个窗口不会都是游戏。

  pField 就是玩家区域，注意这个 pField 只是会记录固定的方块，也就是说你这个方块如果是在下落过程中，是不会记录的。只有落下了才会记录。因为 pField 实际上是就是游戏环境，而下落的方块不是游戏环境，是玩家。就好像迷宫一样，人是玩家，迷宫就是这个 pField。

- 窗口区域：显示给用户看的整个区域。

  screen，需要把 pField 移动到 screen 才能显示。

玩家控制区域的建立

```
pField = new unsigned char[nFieldWidth * nFieldWidth];
 for (int x = 0; x < nFieldWidth; x++)
    for (int y = 0; y < nFieldHeight; y++)
       // 边界设置为9否则都设置为0
       pField[y * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y ==
           nFieldHeight - 1) ? 9 : 0;
```

0 的地方是可以进行移动，9 的地方是边界。窗口的创建需要了解几个 API：

- CreateConsoleScreenBuffer：文档地址 `https://learn.microsoft.com/zh-cn/windows/console/createconsolescreenbuffer`。
  - 作用的创建控制台缓冲区。第一个参数是访问权限，代码里面是 read 和 write 都可以。第二个参数表示缓冲区是否共享，意思就是这个缓冲区能不能被其他线程 access，肯定是不能，所以设置为 0。第三个参数表示句柄能不能让子线程继承，NULL 就是不能了。第四个参数是类型，这个类型没得选。最后一个参数是否包含这个数据，NULL 表示保护。
  - 返回句柄，表示创建了一个能被主进程读写，不能被其他进程共享，不被继承并且数据被保护类型指定的缓冲区。
- SetConsoleActiveScreenBuffer：将指定缓冲区设置为屏幕控制缓冲区。emmm 人家文档好像不建议用这个，写完再看看。
- WriteConsoleOutputCharacter：将字符写入到缓冲区。
  - 第一个参数表示写入的缓冲区，要求 access 必须是可写的。第二个参数要写入的字符。第三个参数是写入的字符数，第四个参数写入的字符坐标，默认 00 就是在左上角，如果想要在中间那可以在这调整，第五个参数表示一个指针，表示实际写入的字符数，作者用 DWORD 接收，这是 MFC 的数据类型，32 位无符号数。

所以他是先创建了一个缓冲区，然后把这个缓冲区设置成屏幕的缓冲区，让他显示在屏幕上，每一次游戏更新，把更新的数据写进缓冲区就行。作者的缓冲区用 wchar 表示，看代码实例里面大多数都是用一个共用缓冲区的联合体表示。
在这里运行的时候出现一个问题，在对 wchar_t 进行内存分配的时候，居然出现分配错误，一开始以为是 os 内存不够，但是后面看了一下是完全够的。后面看了一下是屏幕缓冲区的问题，调大点就没事了。

然后在测试的时候发现，屏幕显示有问题，这个窗口大小太大了。MoveWindow，SetWindow，Setbuffer 这些都没有效果，返回值显示都成功了，但就是没效果。后面上外网看了一下，他这个是因为 visual studio 和 clion 都强制设置了窗口大小，对没错，为了解决这个问题我还下了 clion。后面我在 dev c++ 上用 MoveWindow，SetWindow，Setbuffer 方法是可以的。说明这个问题是和编译器相关。反正只需要手动调整这个窗口大小就行了。

3.3 碰撞检测

用 0 表示空区域。方块的旋转是在一个局部空间内进行，先在局部空间内进行旋转，然后再转移到全局坐标上去。

```
if (nPosX + px >= 0 && nPosX + px < nFieldWidth)
  if (nPosY + py >= 0 && nPosY + py < nFieldHeight)
  {
    if (tetromino[nTetromino][pi] == L'X' && pField[fi] != 0)
    {
      return false;
    }
  }
```

碰撞检测主要就是判断当前点移动之后，会不会碰到其他的点。如果是空白区域那就是 0，否则就是不是。

3.4 控制移动

检测按键

```
  for (int i = 0; i < 4; i++)
  {
    // 检查最高位是不是1
    // 虚拟键码里面上，左，右分别对应38，37，39，十六进制对应27 25 28，还有一个Z用来变换
    bKey[i] = (0x8000 & GetAsyncKeyState(static_cast<unsigned
        char>("\x27\x25\x28Z"[i]))) != 0;
  }
```

GetAsyncKeyState 主要是检测按键，x27 表示 27 是一个十六进制的。用虚拟键码表示按键。0x8000 做与或运算，和最高位对比，如果最高位是 1 说明就按下了，接下来就是移动了。

```
nCurrentX += (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX -
    1, nCurrentY)) ? 1 : 0;
nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX +
    1, nCurrentY)) ? 1 : 0;
nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX,
    nCurrentY + 1)) ? 1 : 0;
```

旋转需要注意的是如果不加限制，会一直旋转的，因为 nCurrentRotation 一直都不会是 0，所以需要一个 flag 来确定是否需要旋转。

```
if (bKey[3])
{
  nCurrentRotation += (!bRotateHold && DoesPieceFit(nCurrentPiece,
      nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
  bRotateHold = true;
}
else
  bRotateHold = false;
```

bRotateHold 来确定当前是否是继续旋转。不能直接把 nCurrentRotation 设置成 0，因为每一次旋转都是接着上一次。

3.5 下一块

首先要添加一个自动下落，这个比较容易，bForceDown 表示就行。

```
if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1))
// 到底了
{
  ++nCurrentY;
}
```

如果能下落，就继续下落，如果下面一个位置是不能下落的，那么就不下落了。既然不下落了，那就要把这个物体固定在环境中，就是固定在 pField 中，前面提到过了这个 pField 实际上是游戏环境，方块是玩家，方块到底就变成一个环境了。就好像 CS 人在移动的时候他就是玩家，死了就变成环境的一部分。

```
for (int px = 0; px < 4; px ++)
  for (int py = 0; py < 4; py++)
```

```
{
  if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation, 4, 4)] == L'X')
  {
    pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 1;
  }
}
```

这里要注意的是 tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation, 4, 4)] 表示当前物体的状态，这里为什么要有一个 Rotate 呢？因为这个控制旋转的逻辑不是 subsequent 的，并不是说点击一次旋转，0 到 90 度，再点击一次 90 度到 180 度。而是点击一次 0 到 90 度，再点击一次 0 到 180 度。
这段代码相当于是把当前方块给画出来了。其实这个代码重复了很多遍，可以写成一个函数。设置下一块的话重新设置就好了

```
nCurrentX = nFieldWidth / 2;
nCurrentY = 0;
nCurrentPiece = rand() % 7;
nCurrentRotation = 0;
```

3.6 删除

主要是指删除某一行。这个检测是否一行都有方块并不是什么时候都要检查，肯定是等下落到底了之后再检查。只需要检查四行就行，因为当前这个方块就占 4 行，如果超过这个范围就不关你的事了，是之前或者之后下落的物体要检查的。

```
for (int py = 0; py < 4; py++)
{
  if (nCurrentY + py < nFieldHeight - 1) {
    bool bLine = true;
    for (int px = 1; px < nFieldWidth - 1; px++)
    {
      // 当前这个高度，全部的px都不是0，也就都不是空
      bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;
    }
    if (bLine)
    {
      // Remove Line
      for (int px = 1; px < nFieldWidth - 1; px++)
```

```
            {
                pField[(nCurrentY + py) * nFieldWidth + px] = 8;
            }
            // 要删除的行
            vLines.push_back(nCurrentY + py);
        }
    }
}
```

当前整行都不是空，那么就把这行放进 vLines 里面，等会删掉他。

```
for (auto& v : vLines)
{
    for (int px = 1; px < nFieldWidth - 1; px++)
    {
        for (int py = v; py > 0; py--)
        {
            pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
        }
        pField[px] = 0;
    }
}
vLines.clear();
```

pField[px] = 0; 这一句我一开始不知道是干什么，后面想了一下，我觉得应该是处理特殊情况的。当你所以方块都填满了整个 pField，你这个下移动只会影响 1 到 nFieldHeight-1 行，第 0 行是没有东西可以给他降下来，没有-1 行，所以这里额外需要处理一下。

3.7 渲染

pField 和移动方块是两种不同的物体，一个是环境一个是玩家，自然渲染不一样。

```
    for (int x = 0; x < nFieldWidth; x++)
    for (int y = 0; y < nFieldHeight; y++)
        // 乘上nScreenWidth是因为整个窗口长度就是nScreenWidth，所以需要乘上这一行
        // 边界是#号，其他地方为空格
        screen[(y + 2) * nScreenWidth + (x + 2)] = L" ABCDEFG=#"[pField[y * nFieldWidth
            + x]];

for (int px = 0; px < 4; px ++)
```

```
for (int py = 0; py < 4; py++)
{
   if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation, 4, 4)] == L'X')
   {
      screen[(nCurrentY + py + 2) * nScreenWidth + nCurrentX + px + 2] =
          nCurrentPiece + 65;
   }
}
```

如果想显示什么分数啊什么的，再 pField 写就行了，如果想增加新的方块，那就要额外渲染。