一、基础环境

1、vscode

官网：https://code.visualstudio.com/docs/?dv=win

汉化：chinese

c/c++插件：c/c++

c/c++运行环境：c/c++ Compile Run extension

其他插件：C/C++ Snippets; EPITECH C/C++ Headers; File Templates ; GBKtoUTF8; Include Autocomplete;

windows: 下载安装编译器,

https://code.visualstudio.com/docs/cpp/config-mingw

通过msys2, https://www.msys2.org/

（linux 下直接根据提示apt install gcc/ g++）

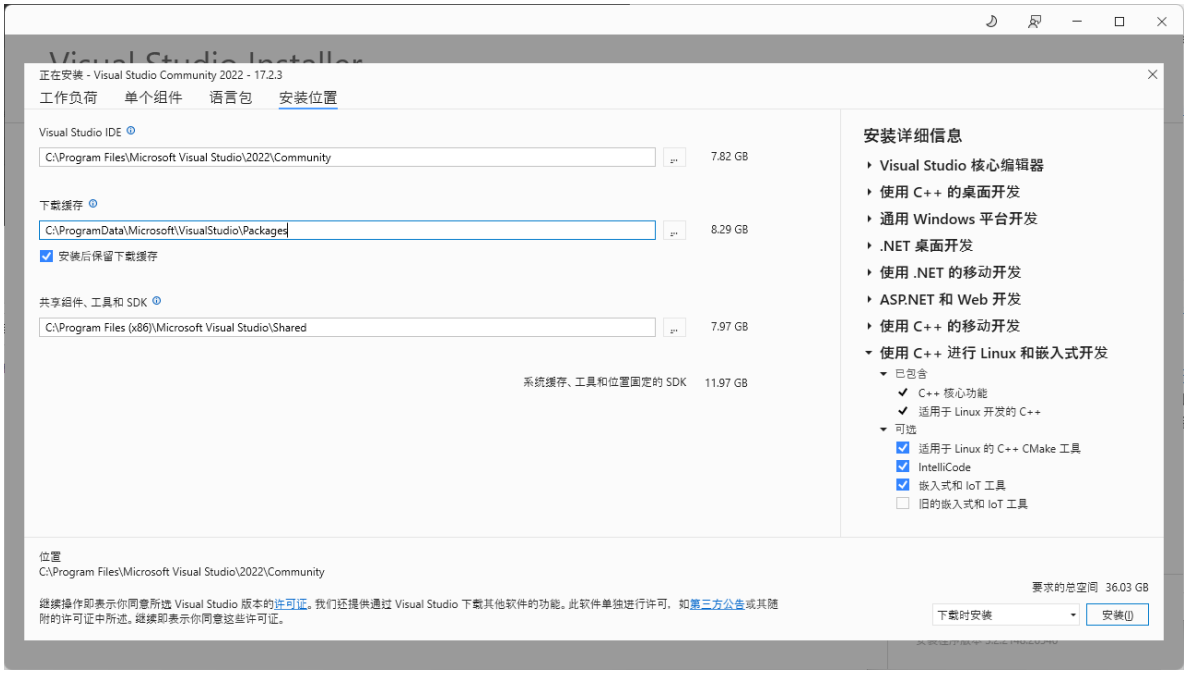
2、vs

官网：https://visualstudio.microsoft.com/zh-hans/downloads

下载最新：vs2022

安装，根据需要选择一些安装， 通过 仅选择所需的组件 来节省安装时间和磁盘空间。 你始终可以根据需要随时以增量方式添加更多组件。（工具/获取工具和功能；文件/项目/找到打开visual studio安装程序/）

安装位置：（为了不卡，放在了SSD）



二、编译

参考： g++ 编译&库链接相关知识： https://zhuanlan.zhihu.com/p/372722440

1、gcc: GCC中的GUN C Compiler（C 编译器 ）

```
gcc test.c -o test
./test
```

2、C生成动态链接库（so）并调用：

参考：C动态链接库的编译： https://blog.csdn.net/weixin_40437821/article/details/110671132

```
$ gcc test_a.c test_b.c test_c.c -fPIC -shared -o libtest.so   // 编译成一个动态
库：libtest.so
$ gcc test_main.c -L. -ltest -o test_main  // 测试调用 动态库的链接
$ ldd test   // 测试是否动态连接，ibtest.so => not found, bashrc添加.so的环境;
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/***/HDD/wood/work/Research/CPP/c_so/src/l
ibtest.so
$ ./test      // 查看如何调用动态库函数
```

3、g++: GCC中的GUN C++ Compiler（C++编译器）

```
g++ testcpp.cpp -o testcpp
./testcpp
# 多个cpp
g++ cpp文件名称 cpp文件名称 -o 可执行文件名称
```

4、C++动态连接库的编译：

```
g++ testcpp_a.cpp testcpp_b.cpp -fPIC -shared -o libtestcpp.so
g++ testcpp_main.cpp -L. -ltestcpp -o testcpp_main
./testcpp_main
```

5、cmake编译：

5-1、windows:

```
windows 安装cmake:
下载msi, 安装。
cd build;
cmake ..;
再vs打开sln文件
```

5-2、linux:

参考：cmake编译c++: https://cmake.org/documentation/ ； https://zhuanlan.zhihu.com/p/373256365

CMake编译生成CPP动态库与静态库：

https://blog.csdn.net/CSDN__Kaker/article/details/122048776

https://blog.csdn.net/cindywry/article/details/86063930

ubuntu 源码安装cmake-3.23.2:

```
https://blog.csdn.net/jingtaoaijinping/article/details/109111957
# 环境配置：
wget https://github.com/Kitware/CMake/releases/download/v3.23.2/cmake-3.23.2.tar.gz
tar -zxvf cmake-3.23.2.tar.gz
cd cmake-3.23.2
./bootstrap
make -j4     // 4线程
make install

# 编译：
cd build;
cmake .. ;
make ;
cd ../bin ;
./***
```

cmake 编译生成可执行文件： example: TestDemo

```
-TestDemo
-- bin   // 可执行文件
-- build  // 编译生成文件
-- include // 头文件：.h
-- lib   // 库文件：.so .a
-- src  // 源文件 .c .cpp
-- CMakeLists.txt
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)
# set the project name 工程名
project(TestDemo)
# 设置编译模式
set(CMAKE_BUILD_TYPE Release)
# 设置可执行文件与链接库保存的路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
# 设置头文件目录
include_directories(
${PROJECT_SOURCE_DIR}/include
)
# add the executable 添加编译文件
add_executable(TestDemo src/OO_class.cpp)
```

cmake 编译生成动态库：example : cmakedemo

CMakeLists:

```
cmake_minimum_required(VERSION 3.10)

# set the project name 工程名
project(TestDemo)

# 设置编译模式
set(CMAKE_BUILD_TYPE Release)

# 设置可执行文件与链接库保存的路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

# 设置头文件目录
include_directories(
${PROJECT_SOURCE_DIR}/include
)

# 生成动态链接库 SHARED
add_library(dymc_lib SHARED ${PROJECT_SOURCE_DIR}/include/so_test.h
                            ${PROJECT_SOURCE_DIR}/src/test_a.c
                            ${PROJECT_SOURCE_DIR}/src/test_b.c
                            ${PROJECT_SOURCE_DIR}/src/test_c.c)

# 添加下面两行调用 链接库生成可执行文件
# ADD_EXECUTABLE(hello_dymc src/test_main.c)
# TARGET_LINK_LIBRARIES(hello_dymc dymc_lib)
```

cmake 调用动态链接库：example : cmakedemo_test

CMakeLists:

```
cmake_minimum_required(VERSION 3.10)

# set the project name 工程名
project(TestDemo)

# 设置编译模式
set(CMAKE_BUILD_TYPE Release)

# 设置可执行文件与链接库保存的路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

# 设置头文件目录
include_directories(
${PROJECT_SOURCE_DIR}/inc
)

# 链接库路径
link_directories(${LIBRARY_OUTPUT_PATH})
# 链接库，dymc_lib 为库名
link_libraries(dymc_lib)

# 生成目标文件
```

```
add_executable(TestDemo src/test_main.c)

# 链接库文件
TARGET_LINK_LIBRARIES(TestDemo dymc_lib)
```

三、基础知识

1、API:

1、简单入门： https://www.runoob.com/cplusplus/cpp-exceptions-handling.html

中文网： http://c.biancheng.net/cplus/ （部分内容收费）

2、API：https://cplusplus.com/doc/

3、进阶： https://github.com/CnTransGroup/EffectiveModernCppChinese

4、cppreference.com

2、C 基础

1、extern ex: 声明变量名而不定义, 不需要建立存储空间。需要在一个源文件中引用另外一个源文件中定义的变量，我们只需在引用的文件中将变量加上 extern 关键字的声明即可。

```
extern int i; //声明，不是定义
```

2、常量定义：#define 预处理器；const 关键字

前缀：0x 十六进制 0 八进制 后缀：后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）

```
#define WINDTH 5
const int HEIGHT = 10;
```

3、存储类: auto register static extern 定义C程序中变量/函数的范围（可见性）和生命周期。

```
// auto: auto 存储类是所有局部变量默认的存储类。{auto int month} auto 只能用在函数内，即 auto 只能修饰局部变量。
// register: 用于定义存储在寄存器中而不是 RAM 中的局部变量。
// static: 指示编译器在程序的生命周期内保持局部变量的存在，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。 static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。
// static example: 每次调用函数，函数中的值不会被重置，所以对于不需要覆盖的场景可用，不需要写到函数外部去。
static int count=10; /* 全局变量 - static 是默认的 */
void func_static(void) {
    static int thingy = 5;
    thingy ++;
    printf("thingy 为 %d, count 为 %d\n", thingy, count);
}
// extern: 提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。
```

4、运算符：

```
// 逻辑运算符：&& || !
// 位运算符： & | ^ ~ << >>
// 杂项运算符：sizeof() 返回变量的大小；ptr = &a 返回变量的地址；* 指向一个变量，*ptr 就是变量a；?: 条件表达式
```

5、初始化变量：局部变量被定义时，系统不会对其初始化；全局变量定义时，系统会自动对其初始化。

```
// 正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果。
//static: https://www.runoob.com/w3cnote/cpp-static-usage.html
```

6、函数传递参数方式：传值调用；引用调用

```
// 传值调用：函数内的参数改变不会影响实际参数。
// 引用调用：指针传递方式，形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作。
void swap(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
    return;
}
```

7、数组

```
// 传递数组给函数：void myFunction(int *param){}; void myFunction(int param[10]){};
void myFunction(int param[]){};
// 从函数返回数组：C 语言不允许返回一个完整的数组作为函数的参数。但是，您可以通过指定不带索引的数组名来返回一个指向数组的指针。返回一维数组：int * myFunction()
// C 不支持在函数外返回局部变量的地址，除非定义局部变量为 static 变量。
int* getRandom() {
    static int r[10];
    int i;
    srand((unsigned)time(NULL)); // 设置种子
    for (i = 0; i < 10; ++i) {
        r[i] = rand();
    }
    return r;
}
// 指向数组的指针：数组名是一个指向数组中第一个元素的常量指针。
```

3、C++ 基础

3-1、linux环境安装、windows vs2019/2022 安装

3-2、C++工作原理：编译与链接

3-3、C++变量

1、基本数据类型：int(4字节byte, 32bit), unsigned int(4), char(1)，short(2), long(4), long long(4), float(4), double(8), bool(1)

2、sizeof(int)

3-4、C++ 函数

1、模块化，防止代码重复

2、声明 .h, 定义 .cpp

3、不要抽出太多函数，程序变慢：每次调用函数时，编译器生成一个call指令。为了调用一个函数，需创建一个堆栈结构，把参数和返回地址等推进堆栈。需要调到二进制执行文件不同部分执行函数指令，内存中跳跃执行函数，跳跃和执行都需要时间。

3-5、C++头文件

1、头文件保护：防止头文件被多次调用导致重定义错误

```
// 方法1：
#pragma once

// 方法2：
#ifndef _LOG_H  // 如果 _LOG_H 宏没有被定义
#defind _LOG_H   // 那么定义它，防止重复包含
// 这里放置头文件的内容
#endif
```

2、引用头文件

” 相对路径 “； <路径>

3、标准库：.h 是C (stdlib.h), 无.h是C++(iostream)

3-6、vs调试

1、断点、读取内存

2、条件和操作断点

3-7、条件与分支（if）

3-8、VS中设置

3-9、循环（for/while）

3-10、C++控制流语句

1、continue: 跳到for循环的下一次迭代

2、break: 完全跳出for循环

3、return: 可以在代码任何地方，退出这个函数

3-11、C++指针

1、指针是一个整数，一种存储内存地址的数字。一个指针只是一个地址，一个保存内存地址的整数。

2、指针前的类型无关紧要（void* , int* , double* 都可以）。但是类型对该内存的操作很有用，如果想对它进行读写，类型可以帮助知道设置几个字节的内存。

但最终，类型是完全没有意义的。

3、**值是内存的地址； * 值是内存中存放变量的地址;

指针本身是变量，这些变量也存储在内存中，因此可以得到双指针或三指针，意思是指向指针的指针。

```
//如下例子
char** ptr = &buffer; //ptr为buffer的内存地址
```

4、例子:

```
char* buffer = new char[8]; //分配了8个字节的内存，并返回一个指向那块内存开始的指针。（创建了一个名为 buffer 的指针，它指向一个可以容纳 8 个 char 元素的动态分配数组（也就是堆上分配的内存））
memset(buffer, 0, 8); // 使用指定的数据填充内存块（使用 memset 函数将数组中的前 8 个字节（因为我们传递了 8 作为第三个参数）设置为 0，也就是初始化为零）
delete[] buffer; //释放了之前动态分配的内存，防止内存泄漏
```

5、指针，只是存储内存地址的整数。

3-12、C++引用

1、引用能做的指针都能做，但是能用引用的就用引用，这样代码会更简洁。

2、引用就是一个别名，不是真正的变量，所以并不占用内存。

2、引用使用的示例:

```
// 按值传递，函数内部对参数的修改不会影响到函数外部的变量。
void Increment(int value){
    value++;
}
int main(){
    int a = 5;
    Increment(a); // a==5
    std::cin.get();
}
// 传内存地址，修改内存地址中的值
void Increment(int* value){
     (*value) ++;
}
int main(){
    int a = 5;
    Increment(&a); // a==6
    std::cin.get();
}
// 引用传递，相当于给a取了一个别名
void Increment(int& value){
    value++;
}
int main(){
    int a = 5;
    Increment(a); // a==6
```

```
    std::cin.get();
}
```

3-13、C++类

1、类只是对数据和功能组合在一起的一种方法。

3-14、C++类和结构体对比

1、技术上唯一区别：默认情况下，类是私有的。如果不指定修改任何可见性。 结构体中，默认值是public的。

2、struct存在原因：希望与c保持向后兼容性。 另一种方法修改：将c语言中的struct 替换class, 然后将其变为public。

3、习惯：struct 只是用做数据的结构。 大量功能、继承等这些就使用类。

3-15、C++中的static

1、类和结构体外部的静态static： 该变量/函数只会在这个翻译单元内部链接。（就像在类中声明一个私有变量，其他所有的翻译单元都不能看到该变量。）

2、当声明静态函数或静态变量时，只会在它被声明的C++文件中看到。

3、在头文件中声明一个静态变量，该头文件包含在两个不同的c++文件中，就是在两个翻译单元中都声明了相同的静态变量。（因为包含头文件时，复制头文件所有内容并将其粘贴到C++文件中。）

4、为何要使用static: 不需要变量是全局变量，就要尽可能多的使用静态变量。因为一旦在全局作用域下声明东西的时候，不设定static, 连接器会跨编译单元进行链接，相当于创建了一个全局变量，会导致一些非常糟糕的bug。

5、**要让函数和变量标记为静态的，除非真的需要它们跨翻译单元链接。**

3-16、C++ 类和结构体中的static

1、当类中的变量静态时，在类的所有实例中只有一个实例。不同实例的该变量都指向同一个地方，一个实例修改，所有实例该变量都发生变化。

2、类中使用静态意义：需要在该类所有实例之间共享数据。

3、静态方法不能访问非静态变量。

3-17、C++中的局部静态（Local Static）

1、在函数中声明一个静态变量，生存期可以保持到整个程序结束。相当于在外部声明一个变量。 一般不要这么用，我的习惯是在外部声明。

```
// 1、外部声明
int i = 0;
int function(){
    i++;
}
int main(){
    function();
    function();
}
```

```
// 等价于局部static
int function(){
    static int i=0;
    i++
}
int main(){
    function();
    function();
}
```

2、类中使用：静态Get方法类。

3-18、C++枚举

1、给特定的值命名的一种方式，这样就不需要在各个地方，处理各种整数。

3-19、C++构造函数

1、构造函数是一种特殊类型的方法，每次构造一个对象都会调用的方法。如果不实例化对象，将不会使用。如果只使用一个类的静态方法，将不会使用（不创建对象，直接 类名::静态方法）。

```
// 构造函数
class Entity{
public:
    float X, Y;
    Entity(float x, float y){
        X=x;
        Y=y;
    }
};
```

2、如果不指定构造函数，依旧会有默认构造函数，但该构造函数实际上什么都不会做，不会为变量进行初始化（打印出来的值是该内存原来留下来的值）。

```
// 默认构造函数 等价如下
class Entity{
public:
    float X, Y;
    Entity(){

    }
};
```

3、C++必须手动初始化所有基本类型，不像java等比如会自动初始化int为0。

4、当使用new关键字创建一个对象实例时，也会调用构造函数。

5、删除构造函数方法（比如只有静态方法调用的类）：1）private 隐藏构造函数；2）类() = delete;

3-20、C++ 析构函数/虚析构函数

1、构造函数是创建一个新的实例对象时运行，常设置变量或者做任何需要的初始化，析构函数是在销毁对象时运行，卸载变量，清理使用过的内存。

2、析构函数同样适用于栈和堆分配的对象：使用new分配一个对象，调用delete时候，析构函数会被调用。

3、不调用析构函数，会造成内存泄漏（内存泄漏是指程序在运行时无法释放已经分配的内存，这可能会导致程序在持续运行时消耗越来越多的内存，最终耗尽系统资源。）。

4、虚析构函数：多态。

3-21、栈（stack）上或堆（heap）上分配类的对象

栈分配：

1. **语法**：

```
ClassName objectName; // 在栈上创建一个对象
```

2. **特点**：
   - 对象的生命周期与作用域相同。当离开对象的作用域时，对象会自动销毁。
   - 不需要手动释放内存。
   - 速度相对较快，因为在栈上分配内存是非常高效的。
3. **示例**：

```
class MyClass {
public:
    MyClass() {
        std::cout << "Constructor called" << std::endl;
    }
    ~MyClass() {
        std::cout << "Destructor called" << std::endl;
    }
};

int main() {
    MyClass obj; // 在栈上创建一个对象
    // ...
} // 当 main 函数结束时，obj 会自动销毁
```

堆分配：

1. **语法**：

```
ClassName* pointerName = new ClassName(); // 在堆上创建一个对象
```

2. **特点**：
   - 对象的生命周期由程序员控制，需要手动释放内存，否则会造成内存泄漏。
   - 对象在堆上分配，需要使用 `delete` 关键字显式地释放内存。
   - 对象可以在作用域之外继续存在，只要你保留了指向它的指针。
3. **示例**：

```
class MyClass {
public:
    MyClass() {
        std::cout << "Constructor called" << std::endl;
```

```
    }
    ~MyClass() {
        std::cout << "Destructor called" << std::endl;
    }
};

int main() {
    MyClass* ptr = new MyClass(); // 在堆上创建一个对象
    // ...
    delete ptr; // 显式释放内存，调用析构函数
} // 手动释放内存
```

如何选择：

- 如果对象的生命周期可以很容易地确定，并且在特定作用域内，通常可以使用栈分配，因为它更加简单和安全。
- 如果你需要在程序的多个部分共享对象，或者对象的生命周期在作用域之外，你可能需要使用堆分配，但要记得负责释放内存，以避免内存泄漏。

在现代C++中，推荐使用智能指针（如 `std::unique_ptr` 或 `std::shared_ptr`）来管理堆分配的对象，这样可以避免手动释放内存的烦恼。

3-22、C++继承

1、创建一个子类时，它将包含父类中的一切。

3-23、C++虚函数

1、虚函数(virtual)允许在子类中重写方法。想要覆写一个函数，必须将基类中的基函数标记为虚函数。

> 虚函数引入了一个叫做Dynamic Dispatch(动态联编)的东西，通过v表（虚函数表）来实现编译。v表就是一个表，包含基类中所有虚函数的映射，在运行时，将他们映射到正确的覆写（override）函数。

2、C++11 引入 将覆写函数标记为 override. 好处：可读性更强；会检查是否有对应的基函数或者拼写错误等bug。

3、虚函数不是免费（无额外开销）的：有两种于虚函数相关的运行时成本。1）需要额外的内存来存储v表，以便分配到正确的函数。包括基类中要有一个成员指针指向v表。2）每次调用虚函数时，需要遍历这个表，来确定要映射到哪个函数。 除了在一些cpu特别差的嵌入式设备上，其他影响很小不需要注意。

3-24、C++ 接口（纯虚函数）

1、本质上于其他语言（如java 或 c#) 中的抽象方法或接口相同。纯虚函数允许在基类中定义一个没有实现的函数，强制子类去实现该函数。

2、接口（interface）：创建一个类，只由未实现的方法组成，然后强制子类去实际实现它们，非常常见。

```
class Entity{
public:
    virtual std::string GetName() = 0; // = 0 纯虚函数
};
```

3、有纯虚函数，就不在具有实例化基类能力。纯虚函数必须被实现，才能创建这个类的实例。

3-25、C++可见性

1、指类的某些成员或方法实际上由多可见；对程序运行、性能等无影响。纯粹语言中存在的东西，为了写出更好的代码或组织代码。

2、C++中3个基础的可见性修饰符：private、protected 和 public

private: 只有（并不）该类可以访问这些变量。但是C++中有个叫 friend 的关键字，可以让类或函数成为该类的友元，也可以访问。

protected: 这个类和层次结构中的所有子类，可以访问这些符号。但是比如在main() 中依然不能访问。

public: 都可以访问它。

3、为何不让所有东西都是public的：帮助其他开发者和自己。阅读代码和扩展代码时候对代码容易维护、理解。

3-26、C++ 数组

1、数组分配方式

```
// 栈中分配，生命周期在{}内
int a[5];
// 数组大小：不要直接访问数组元素个数，可以sizeof(a)/sizeof(int)得到count，但是这个方法必须在栈中分配数组。不过这种方法也很糟糕，最好自己维护。
int count = sizeof(a) / sizeof(int);
// 自己维护
static const int exampleSize=5;
int example[exampleSize];

// 堆中分配，生命周期在被销毁前一直在，所有需要delete example[]手动销毁
int* example = new int[5];

// C++11 std::array 会自己做边界检查，使用std数组比原始数组会安全得多。
#include <array>
std::array<int, 5> another;
int count = another.size();
```

3-27、C++字符串

1、字符串

```
// 习惯const，不想去改变这些的值， 可变就不用const； 字符串是不可变的，不能扩展字符串使它变大，因为这是一个固定分配的内存块；字符串从指针的内存地址开始，直到碰到0， 空终止字符；长度，strlen(name);
const char* name = "Wood";
// 等价于
const char name2 = {'W', 'o', 'o', 'd', '\0'}
// 如果需要扩展，需要执行一个全新的分配并删除旧的字符串

// C++ std::string; name.size();
std::string name = "Wood";
// 追加：不能 std::string name = "Wood" + " Hello!"; 需要先封装成一个对象
```

```
std::string name = std::string("Wood") + " Hello!";
// or
std::string name = "Wood";
name += " Hello!";
// find
bool contains = name.find("no") != std::string::npos;
// 函数使用传递string：把类（对象）传递给一个函数时，实际上是在复制这个类（对象），所以操作（如追加）不会影响到传递的原始字符串。 相当于只读情况，由于字符串复制较慢，还会分配空间，所以为了优化速度，只读情况下确保使用常量引用传递它。
// 只读较慢 void PrintString(std::string string)
// 可修改：加上常量const和引用&; 引用意味着不会被复制，const 意味承诺不会在这里修改它
void PrintString(const std::string& string){}
```

2、字符串自变量

```
// 标准用法
const char* name = "Wood"; // 等价于 const char* name = u8"Wood";
// 如果要修改
char name[8] = "Wood";
// 扩展
const char* name = u8"Wood"; // 一个字节的字符
const wchar_t* name2 = L"Wood"; //
const char16_t* name3 = u"Wood"; // 2个字节16bit的字符
const char32_t* name4 = U"Wood"; // 4个字节32bit的字符
// C++14 中一些让事情变得简单的库
using namespace std::string_literals;
std::string name0 = "luckey"s + "Wood"; // s是一个操作符函数，返回标准字符串（对象）

// 段落长文 方式1 加上R,忽略转义字符
const char* ex = R"(Line1
Line2
Line3
Line4)";
//2
const char* ex2 = "Line1\n"
    "Line2\n"
    "Line3\n";
```

字符串自变量的内存： 字符串自变量永远保存在内存的只读区域内。

3-28、C++ 的const

1、让代码更干净，承诺某些东西不变。（但是可以绕过，是否遵守诺言取决于你自己。）

2、用法

```
// 常量
const int MAX_AGE = 90;
// 指针
const int* a = new int;  // 不能修改该指针指向的内容(数据 *a)，可把实际的指针本身重新赋值（a）；等价于 int const* a = new int;
int* const a = new int; // 可以改变指针指向的内容（*a），不能把实际的指针本身重新赋值(a)，指向别的东西
const int* const a = new int;  //不能改变指针指向的内容（*a），也不能改变指针本身
// 方法名之后，与变量无关。 只能在类中有效，意味着该方法不会修改任何实际的类，可以看到不能修改类成员变量。只读不写。set 就不能用const
```

```
class Entity
{
private:
    int m_X, m_Y;
public:
    int GetX() const{
        return m_X;
    }
};
// 如果m_X是一个指针，想让它保持不变，写法如下。3个const,意味着 指针的内容也不能修改， 返回了一个不能被修改的指针， 函数承诺不修改实际的Entity类。
class Entity
{
private:
    int* m_X, m_Y; // m_X是一个指针，m_Y是int整型。想让同一行是指针，int* m_X, *m_Y;
public:
    const int* const GetX() const{
        return m_X;
    }
};

// 类中const的方法， 在有常量引用情况下就可以调用，如：
void PrintEntity(const Entity& e){
    std::cout<<e.GetX()<<std::endl;
}
```

mutable: 某些原因，又确实需要改变一些变量。

```
class Entity
{
private:
    int m_X, m_Y;
    mutable int var; // mutable 允许函数是常量方法，但是可以修改变量。
public:
    int GetX() const{
        var = 2;
        return m_X;
    }
};
```

3-29、C++ 的mutable 关键字

1、与const一起使用：标记类成员为mutable, 类中的const 方法可以修改这个成员。

2、用在lambda表达式中：像一个一次性的小函数，写出来并赋值给一个变量。实践中不会发生，用不到。

```
int x =8;
auto f = [=]() mutable{ //= 值传递， &引用传递, 不用 mutable 则需要定义一个 int y; y = x; y++;
    x++;
    std::cout<< x<< std::endl;
}
f();
// x = 8
```

3-30、C++的成员初始化列表

1、构造函数初始化类成员：成员初始化列表时，要与成员变量声明时的顺序一致。

2、为何使用成员初始化列表：风格干净；功能上对性能的浪费；

3-31、C++的三元操作符

1、三元运算符：一个问号，一个冒号。if语句的语法糖。

```
s_Speed = s_Level > 5?10:5;
// 等价于
if (s_Level > 5)
    s_Speed = 10;
else
    s_Speed = 5;
```

3-32、创建并初始化C++对象

1、应用程序，会将内存分为两个部分：栈和堆。

2、栈对象：有一个自动的生存期，由声明的地方的作用域决定。大多数情况下都这样做。

3、堆对象：手动释放。如果想放在函数生存期之外，对象很大（栈空间相对较小），需要在堆上进行分配。

3-33、C++ new关键字

1、new主要目的：在堆上分配内存。new + 数据类型（类/基本类型/数组), 决定了必要的分配大小，以字节为单位。如new int, 将需要4个字节的内存，需要找到一共包含4个字节内存的连续块，找到后，返回一个指向这个内存的指针，这样就可以开始使用数据，存储、读写访问。

2、找包含连续的空闲内存：并不是顺序扫描。 有一个空闲列表，会维护那些有空闲字节的地址。

3、再次强调：指针只是一个内存地址，指针之所以需要类型，是因为需要类型来操控它。

3-34、C++ 隐式转换与explicit关键字

1、C++隐式构造函数和隐式转换。尽量避免使用。

```
#include<iostream>
#include<string>

class Entity
{
private:
    std::string m_Name;
    int m_Age;
public:
    Entity(const std::string& name):m_Name(name), m_Age(-1){}
    Entity(int age):m_Name("UnKnown"), m_Age(age){}
};

int main(){
    //常用方式（栈）
    // Entity a("Wood");
    // Entity b(25);
```

```
    // 隐式转换（隐式构造函数）
    Entity a = "Wood";
    Entity b = 22;

    std::cin.get();
}
```

2、explicit关键字：禁用隐式implicit的功能。放在构造函数的前面，意味着没有隐式的转换。

3-35、C++运算符及其重载

1、运算符：是使用的一种符号，通常代替一个函数来执行一些事情。运算符重载：定义和更改使用的性质。

3-36、C++的this关键字

1、访问成员函数（一个属于某个类的函数）；在方法内部，引用this, this是一个指向当前对象实例的指针，该方法属于这个对象实例。

3-37、C++的智能指针

1、new在堆上分配内存，需要delete释放内存。智能指针实现这一过程自动化，当你调用new的时候，不用调用delete，甚至不需要调用new。

2、智能指针：一个原始指针的包装。创建一个智能指针，它会调用new并为你分配内存，然后基于你使用的智能指针，这些内存会在某一时刻自动释放。

3、智能指针 unique_ptr, 是作用域指针, 只是一个栈分配对象，当栈分配对象死亡时，将调用delete在你的指针上，并释放内存。

4、不能复制。如果你想复制或者分享，使得这个指针可以被传递到一个函数中或者一个类中，不能复制。 std::unique_ptr 是一个不能被拷贝的类。 因为它会导致两个 std::unique_ptr 实例共同拥有相同的资源，这是不被允许的。如果要分享，可以使用shared_ptr。

5、shared_ptr的工作方式是引用计数，跟踪你的指针有多少个引用。一但引用计数达到0，就被删除。

6、弱指针 weak_ptr: 不会增加引用计数。用处：如果不想用Entity类的所有权，例如排序一个Entity列表，不关心他们是否有效，只需要存储它们的一个引用。

```
#include<iostream>
#include<string>
#include<memory>

class Entity{
public:
    Entity(){
        std::cout<<"Create Entity!"<<std::endl;
    }

    ~Entity(){
        std::cout<<"Destroyed Entity!"<<std::endl;
    }
    void Print(){}
};

int main(){
```

```
    {
        // std::unique_ptr<Entity> entity(new Entity()); //ok, 一般不这样写
        std::unique_ptr<Entity> entity = std::make_unique<Entity>(); // ok, 异常安全
        // std::unique_ptr<Entity> e0 = entity; // 错误， std::unique_ptr 是一个不能被拷贝的类。 因为它会导致两个 std::unique_ptr 实例共同拥有相同的资源，这是不被允许的。

        // shared_ptr
        std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
        std::shared_ptr<Entity> e0 = sharedEntity;

        // 弱指针
        std::weak_ptr<Entity> weakEntity = sharedEntity; // 相比shared_ptr，不会增加引用计数。

        entity->Print();
    }

    std::cin.get();
}
// g++ unique_ptr_ex.cpp -o unique_ptr_ex
// ./unique_ptr_ex
```

7、如果想自动化内存管理，尽量使用智能指针 unique_ptr, 开销更低。shared_ptr因为还要维护一个引用计数，所有有额外开销。但是如果需要在对象之间共享，就使用shared_ptr。

3-38、C++的复制和拷贝构造函数

1、拷贝：指要求复制数据，复制内存。有时候需要避免，因为浪费性能。

2、深拷贝：根据定义，复制整个对象。记住，使用const 引用来传递对象，如PrintString 函数。

避免拷贝：如果你传递一个对象作为参数，而不使用引用，会导致对象的拷贝。对于大型对象或者需要频繁传递的对象来说，这可能会带来性能开销。使用引用可以避免这种拷贝，提高程序的效率。

保护原始对象：使用 `const` 修饰引用意味着在函数内部不能修改原始对象的状态。这提供了一定程度的安全，因为你可以确保在函数内部不会意外地修改传递的对象。

允许操作临时对象：如果你有一个临时对象（例如在函数调用中创建的对象），你可以通过使用 `const` 引用来传递它，因为临时对象无法作为非常量引用的参数传递。

更通用的函数：使用 `const` 引用允许你编写更通用的函数，因为它们可以接受常量和非常量对象作为参数。

避免不必要的拷贝和内存开销：当你传递大型对象时，使用 `const` 引用可以避免不必要的拷贝操作，从而减少了内存开销。

```
#include<iostream>
#include<string>
#include<cstring>

class String{
```

```
private:
    char* m_Buffer;
    unsigned int m_Size;
public:
    String(const char* string){
        m_Size = strlen(string);
        m_Buffer = new char[m_Size + 1];
        memcpy(m_Buffer, string, m_Size + 1);
        //  memcpy(m_Buffer, string, m_Size);
        //  m_Buffer[m_Size] = 0;
    }
    // 添加拷贝构造函数
    String(const String& other): m_Size(other.m_Size){
        std::cout<<"copy-------"<<std::endl;
        // m_Size = other.m_Size;
        m_Buffer = new char[m_Size + 1];
        memcpy(m_Buffer, other.m_Buffer, m_Size + 1);
    }
    ~String(){ // 不加会内存泄漏
        delete[] m_Buffer;
    }
    char& operator[](unsigned int index){
        return m_Buffer[index];
    }
    friend std::ostream& operator<<(std::ostream& stream, const String& string);
};

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

// 不写 void PrintString(String string)，这样会多次调用拷贝构造函数，每次调用都会发生复制，浪费性能。直接传引用，只发生String second = string 这一个复制。
void PrintString(const String& string){
    std::cout<<string<<std::endl;
}

int main(){
    String string = "Wood";
    // 赋值 -- 崩溃：浅拷贝，默认的拷贝构造函数。代码中，当你执行 String second = string; 时，实际上只是将 m_Buffer 的指针从 string 复制到了 second，而不是创建一个新的内存副本。
    // string 和 second 都被销毁时，它们会尝试释放相同的内存，导致了问题。
    // 需要实现一个自定义的拷贝构造函数，以确保在创建新对象时会分配新的内存并复制原始字符串。
    // String second = string;
    // std::cout << string << std::endl;
    // std::cout << second << std::endl;

    // 要做的：分配一个新的char数组，来存储复制的字符串。深拷贝，实现拷贝构造函数。
    String second = string;
    second[2] = 'a';
    PrintString(string);
    PrintString(second);
    // std::cout << string << std::endl;
    // std::cout << second << std::endl;


    std::cin.get();
```

```
}

// g++ string_ex.cpp -o string_ex
// ./string_ex
```

3-39、C++的箭头操作符

1、箭头操作符（->）用于通过指针访问对象的成员，箭头操作符（`->`）通常用于访问一个类（或结构体）的成员，当你有一个指向类（或结构体）对象的指针时，可以使用箭头操作符来访问对象的成员。

```
MyClass* ptr = new MyClass(); // 创建一个MyClass的对象，并将其地址赋给ptr

ptr->memberFunction(); // 调用MyClass的成员函数
ptr->memberVariable = 10; // 设置MyClass的成员变量的值
```

2、点操作符（.）用于通过对象本身访问成员，你有一个指向类对象的引用，你可以使用点操作符（`.`）来访问成员。

```
MyClass obj;
obj.memberFunction(); // 调用MyClass的成员函数
obj.memberVariable = 20; // 设置MyClass的成员变量的值
```

3、运算符重载；获取内存中某个值的偏移量。

3-40、C++的动态数组/静态数组/多维数组

1、动态数组通常是通过使用指针和`new`运算符来创建的。动态数组的大小可以在运行时确定，而不是在编译时确定。必须显式地调用`delete[]`来释放内存。

```
// 动态数组：在运行时决定数组的大小，数组的大小是在运行时通过new运算符动态分配的，因此可以根据需要来分配不同大小的数组。
int* dynamicArray = new int[5]; // 创建一个包含5个整数的动态数组

// 静态数组
int myArray[5]; //数组的大小在编译时确定
```

```
// 创建动态数组时不指定大小，你可以使用动态内存分配函数 new[] 来分配一个初始大小为0的数组，然后在运行时根据需要动态调整大小
int* dynamicArray = nullptr; // 初始化指针为nullptr，表示没有分配内存
int size; // 在运行时获取数组的大小
std::cout << "Enter the size of the dynamic array: ";
std::cin >> size;
dynamicArray = new int[size]; // 动态分配大小为size的数组
// 使用动态数组
for (int i = 0; i < size; ++i) {
    dynamicArray[i] = i * 10;
}
// 访问动态数组的元素
for (int i = 0; i < size; ++i) {
    std::cout << dynamicArray[i] << " ";
}
delete[] dynamicArray; // 释放动态数组的内存
```

2、std::vector: C++标准库中提供的一个动态数组（ArrayList）容器。尽量使用对象，不使用指针，这样再连续的内存栈中（在C++中，尽量使用对象而不是指针，因为对象会在栈上分配内存，而指针可能会在堆上分配内存。栈上分配的内存是连续的，可以提高访问效率。）。指针是一种，就像栈分配和堆分配一样，指针是最后的选择。 将vector传递给函数或类或其他时，确保通过引用传递，不修改就const引用，`void fuction(const std::vector<Vertex>& vertices)`。这样确保没有把整个数组复制到这个函数中。

```
#include<iostream>
#include<vector>

int main(){
    std::vector<int> myVector; // 创建一个存储整数的空向量

    myVector.push_back(10); // 将10添加到向量末尾
    myVector.push_back(20); // 将20添加到向量末尾
    myVector.push_back(30);

    int value = myVector[0]; // 访问第一个元素，值为10
    int size = myVector.size(); // 获取向量的大小，此时为2
    for (int i = 0; i < myVector.size(); ++i) {
        std::cout << myVector[i] << " ";
    }
    std::cout<<std::endl;
    // myVector.pop_back(); // 删除向量末尾的元素
    myVector.erase(myVector.begin() + 1); // 移除某一个元素
    for (auto it = myVector.begin(); it != myVector.end(); ++it) {
        std::cout << *it << " ";
    }
    myVector.clear(); // 清空向量，使其变为空
    std::cin.get();
}

// g++ vector_ex.cpp -o vector_ex
// ./vector_ex
```

3、std::vector 优化：push_back, 当旧分配的内存用完，旧会发生复制到新的分配，会将代码拖慢。优化复制：

拷贝1：将vertex（构造的动态数组） 从main函数（main栈中）放到实际的vector分配的内存中。==》优化：适当的位置构造vertex在vector分配的内存中。使用emplace_back，因为push_back 传递的是构建的vertex对象，emplace_back只是传递了构造函数的参数列表，在实际的vector内存中，使用参数构造一个vertex对象。

拷贝2：vector默认大小是1，push过程中调整大小会发生复制。 ==》优化：避免大小调整的复制操作。制造足够大小的内存。`std::vector<Vertex> vertices; vertices.reserve(3);` 调整容量。

4、静态数组：不增长的数组，std::array。存储在栈中。有边界检查，可记录数组大小。

```
#include <iostream>
#include <array>

int main() {
    // 声明并初始化一个 std::array
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    // 访问元素
```

```
    std::cout << "Element at index 2: " << myArray[2] << std::endl;

    // 修改元素
    myArray[2] = 10;

    // 遍历数组
    std::cout << "Updated array: ";
    for (const auto& element : myArray) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // 获取数组大小
    std::cout << "Array size: " << myArray.size() << std::endl;

    // 使用迭代器遍历数组
    std::cout << "Array elements using iterators: ";
    for (auto it = myArray.begin(); it != myArray.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}

// g++ array_ex.cpp -o array_ex
// ./array_ex
```

5、多维数组：二维数组==》数组的数组；三维数组==》数组的数组的数组；

如何解决内存碎片化问题：让数组在一个连续的内存缓冲区。一维数组内存都在一行，多维数组每遍历完一维，就必须跳转到内存中的另一个位置来读写数据，就会导致cache miss(缓存不命中)， 浪费时间从ram中获取数据。可以使用一维数组，像二维数组进行访问： array[x + y * size].

RAM（随机存取存储器）和Cache（高速缓存）是计算机中用于存储数据的两个不同层次的存储设备，它们在速度、容量、成本等方面有着显著的区别。以下是它们的主要区别：

1、速度：

Cache： Cache 是计算机内部速度最快的存储设备之一，它位于处理器核心附近，通过高速缓存总线直接与处理器通信。Cache 的访问速度比 RAM 快得多。

RAM： RAM 速度相对较快，但通常比 Cache 慢。它位于处理器和硬盘之间，用于存储正在执行的程序和数据。

2、容量：

Cache： Cache 的容量通常较小，因为其制造成本昂贵。通常分为多个层次（L1、L2、L3 Cache），每个层次的容量逐渐增加，但总体仍然相对有限。

RAM： RAM 的容量通常比 Cache 大得多，以 GB 为单位。RAM 存储着当前正在使用的程序和数据。

3、成本：

Cache： Cache 的制造成本非常高昂，尤其是更接近处理器核心的 L1 Cache。因为它需要使用更昂贵的材料和制造技术。

RAM： RAM 的制造成本相对较低，使得它能够提供更大的存储容量。

4、层次结构：

Cache： Cache 通常分为多个层次（L1、L2、L3 Cache），每个层次的作用和容量不同。L1 Cache最接近处理器核心，L2 Cache 通常位于处理器和主内存之间，而 L3 Cache 可能在处理器芯片或芯片组级别。

RAM： RAM 通常分为两个主要类型，即静态RAM（SRAM）和动态RAM（DRAM）。SRAM 更快但成本更高，通常用于较小的 Cache。DRAM 成本相对较低，用于主内存。

6、可编程性：

```
    Cache： Cache 是硬件级别的存储，通常不由程序员直接控制。
    RAM： RAM 是可编程的，程序员可以通过指针或其他内存管理工具直接访问和控制 RAM。
```

6、 自己写 C++数据结构：

- Array数组
- Vector数组

3-41、C++ 中使用库

1、静态链接：意味这这个库会被放到你的可执行文件中。静态链接在技术上更快，因为编译器或链接器实际上可以执行链接时优化之类的。通常静态链接是最好的选择。静态链接是在编译时发生的。

2、使用动态库：动态链接是在运行时发生的。

3、创建和使用库：visualstudio中。

3-42、C++ 处理多返回值 / 结构化绑定

1、元组 tuple，pair，

```
1、函数void，参数引用传递。 这样没有复制拷贝操作，但是需要传递一个有效的变量。
2、函数void，指针。可以传递null。
3、返回一个数组，如返回两个字符串：static std::string* fun(...){...  return new std::string[]{vs, fs};} 。使用了new，导致了堆分配的发生，如何避免：static std::array<std::string, 2> fun(...){... std::array<std::string, 2> result; result[0] = vs; result[1] = fs; return result;}。
4、std::vector<std::string> 。Array在栈上创建，vector底层存储在堆上，所以技术上std::array更快。
5、返回不同类型的变量：
tuple：基本上是一个类，包含x个变量，不关心类型。 std::tuple<std::string, std::string, int>; return std::make_pair(vs, fs);
    调用和取数据：auto source = fun(...); std::string vs = std::get<0>(source);
... 用数字处理变量，代码可读性不是很好。
pair: std::pair<std::string, std::string>;  source.first; source.second; 依旧可读性很差，所以使用struct
6、struct结构体：
    struct Shader{
        std::string VertexSource;
        std::string FragmentSource;
    };
    source.VertexSource; source.FragmentSource; 可读性很高。
```

2、C++的结构化绑定（c++17新特性）：

```
// 避免使用结构体，且保持可读性的方法
std::tuple<std::string, int> CreatePerson(){
    return {"Wood", 25};
}
int main(){
    auto[name, age] = CreatePerson();
}
// g++ -std=c++17 **.cpp -o **
```

3-43、C++的模板

1、模板定义：模板允许定义一个可以根据你的用途进行编译的模板。（基于给编译器的规则让编译器给你写代码）

2、 模板只有在它被调用时，才会被创建。

3-44、C++的堆与栈内存的比较

1、应用程序启动后，操作系统将程序加载到内存，并分配一大堆物理RAM。 栈和堆是ram中实际存在的两个区域。这两个内存区域的实际位置都在我们的内存中。在栈上分配 更快，尽量栈分配。在栈上无法分配再选择在堆上分配，比如需要这个生命周期比函数的作用域更长，或者需要更多的数据，比如几十M。

2、栈：通常是一个预定义大小的内存区域，通常约为2兆字节左右。 内存分配：栈中分配内存的时候，发生的是栈指针移动字节，内存相互叠加存储，所以栈分配很快，所做的就是移动栈指针，然后返回栈指针的地址。

3、堆：一个预定义了默认值的区域，但是它可以生长，并随着应用程序的进行而改变。 内存分配：需要new关键字分配。 new关键字实际上调用了一个叫做malloc的函数（memory allocate的缩写），这样做通常会调用底层操作系统或平台的特定函数，将在堆上分配内存。 当启动应用时候，会得到一定数量的物理ram分配给你，程序会维护一个空闲列表（free list）跟踪哪些内存块是空闲的。

3-45、C++的宏

1、在编译之前被替换。调试、日志W等 #if ... #endif 可以用，尽量不要使用后导致代码阅读性差。

3-46、C++的auto关键字

1、类型非常长和复杂的时候可以使用。简短的如int, string等不要使用，会导致代码可读性差。

3-47、C++的函数指针

1、函数指针：是将一个函数赋值给一个变量的方法。

```
#include<iostream>
#include<vector>

void HelloWorld(){
    std::cout << "Hello World!" << std::endl;
}

// void HelloWorld(int a){
//     std::cout << "Hello World! value: " << a << std::endl;
// }

void PrintValue(int value){
    std::cout << "Value: " << value << std::endl;
}

void ForEach(const std::vector<int>& values, void(*func)(int)){
    for(int value : values)
        func(value);
}

int main(){
    // 方式1： c语言中的原始函数指针
```

```
    void(*func)() = HelloWorld;
    func();

    // 方式2：
    typedef void(*HelloWorldFunction)();
    HelloWorldFunction func2 = HelloWorld;
    func2();

    // 方式3：
    auto func3 = HelloWorld;
    func3();

    // typedef void(*HelloWorldFunction)(int);
    // HelloWorldFunction func2 = HelloWorld;
    // func2(8);

    std::vector<int> values = {1, 5, 4, 2, 3};
    ForEach(values, PrintValue);

    // 用完即弃的lambda 函数, []叫捕获方式，也就是如何传入传出参数
    ForEach(values, [](int value) {std::cout << "Value: " << value << std::endl;
});

    std::cin.get();
}
// g++ function_ptr_ex.cpp -o function_ptr_ex
// ./function_ptr_ex
```

3-48、C++中的lambda

1、lambda, 匿名函数，只要你有一个函数指针，都可以在C++中使用lambda. 不需要通过函数定义，就可以定义一个函数的方法。

2、用法：在会设置函数指针指向函数的任何地方，都可以将它设置为lambda. (参考cppreference.com)

3、std::find_if, 在迭代器中找值，查找满足特定条件的第一个元素。

```
#include<iostream>
#include<vector>
#include<functional>
#include<algorithm>


void ForEach(const std::vector<int>& values, void(*func)(int)){
    for(int value : values)
        func(value);
}

void ForEach2(const std::vector<int>& values, const std::function<void(int)>&
func){
    for(int value : values)
        func(value);
}

int main(){
    std::vector<int> values = {1, 5, 4, 2, 3};
```

```
    // 用完即弃的lambda 函数, []叫捕获方式，也就是如何传入传出参数
    ForEach(values, [](int value) {std::cout << "Value: " << value << std::endl;
});
    auto lambda = [](int value) {std::cout << "Value: " << value << std::endl;
};
    ForEach(values, lambda);

    // 有变量传入
    int a = 5;
    // = 按值传入所有的变量
    auto lambda2 = [=](int value) {std::cout << "Value: " << a << std::endl; };
    ForEach2(values, lambda2);

    // std::find_if
    auto it = std::find_if(values.begin(), values.end(), [](int value){return
value > 3;});
    std::cout << *it << std::endl;
    // 输出找到的元素的位置
    if (it != values.end()) {
        std::cout << "First element greater than 3 found at position: " <<
std::distance(values.begin(), it) << std::endl;
    } else {
        std::cout << "No element greater than 3 found." << std::endl;
    }
    std::cin.get();
}
// g++ lambda_ex.cpp -o lambda_ex
// ./lambda_ex
```

3-49、命名空间namespace

1、主要目的：为了避免命名冲突。

3-50、线程

1、并行与并发：

```
并发：
并发（Concurrent），在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行。
并发不是真正意义上的“同时进行”，只是CPU把一个时间段划分成几个时间片段（时间区间），然后在这几个时间区间之间来回切换，由于CPU处理的速度非常快，只要时间间隔处理得当，即可让用户感觉是多个应用程序同时在进行。如：打游戏和听音乐两件事情在同一个时间段内都是在同一台电脑上完成了从开始到结束的动作。那么，就可以说听音乐和打游戏是并发的。
并行：
并行（Parallel），当系统有一个以上CPU时，当一个CPU执行一个进程时，另一个CPU可以执行另一个进程，两个进程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。
其实决定并行的因素不是CPU的数量，而是CPU的核心数量，比如一个CPU多个核也可以并行。
所以，并发是在一段时间内宏观上多个程序同时运行，并行是在某一时刻，真正有多个程序在运行。
```

2、join 调用目的：在主线程上等待工作进程完成所有的执行后，再继续执行主线程。

```
#include<iostream>
#include<thread>
```

```
static bool s_Finished = false;

void DoWork(){
    using namespace std::literals::chrono_literals;
    std::cout << "Started thread id=" << std::this_thread::get_id() <<
std::endl;
    while(!s_Finished){
        std::cout << "Working...\n";
        std::this_thread::sleep_for(1s);
    }
}

int main(){
    std::thread worker(DoWork);
    std::cin.get();
    s_Finished = true;

    worker.join(); // 在主线程上等待工作进程完成所有的执行后，再继续执行主线程。
    std::cout << "Finished." << std::endl;
    std::cout << "Started thread id=" << std::this_thread::get_id() <<
std::endl; //id 与DoWork不一致，因为在不同线程上
    std::cin.get();
}
// 在 Linux 系统上，需要使用 -pthread 标志来链接线程库。在编译和链接时添加这个标志可以解决这个问题。修改编译命令如下：
// g++ -pthread thread_ex.cpp -o thread_ex
// ./thread_ex
```

3-51、C++的计时

1、chrono api 。

```
#include<iostream>
#include<thread>
#include<chrono>


// struct Timer{
//     std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
//     std::chrono::duration<float> duration;
//     Timer(){
//         start = std::chrono::high_resolution_clock::now();
//     }
//     ~Timer(){
//         end = std::chrono::high_resolution_clock::now();
//         duration = end - start;
//         float ms = duration.count()*1000.0f;
//         std::cout<<"Timer took " << ms << "ms" << std::endl;
//     }
// };


class Timer
{
public:
    Timer() : beg_(clock_::now()) {} // 构造函数：初始化计时器，记录当前时间点。
```

```
    void reset() { beg_ = clock_::now(); } // reset 函数：重置计时器，将起始时间点设置为当前时间点。
    // double elapsed() const { // elapsed 函数：计算自上次重置以来经过的时间，并以秒为单位返回。
    //     return std::chrono::duration_cast<second_>
    //         (clock_::now() - beg_).count(); }
    double elapsed() const { // elapsed 函数：计算自上次重置以来经过的时间，并以毫秒为单位返回。
        return std::chrono::duration_cast<second_>
            (clock_::now() - beg_).count() * 1000.0f; }
    void out(std::string message = ""){ // out 函数：输出经过的时间，并可附带一条消息。
        double t = elapsed();
        std::cout << message << "\nelasped time:" << t << "ms\n" << std::endl;
        reset();
    }
private:
    typedef std::chrono::high_resolution_clock clock_;
    typedef std::chrono::duration<double, std::ratio<1> > second_;
    std::chrono::time_point<clock_> beg_;
};


void Function(){
    Timer timer;
    for (int i=0; i<100; i++){
        // std::cout << "Hello" << std::endl;
        std::cout << "Hello\n";
    }
    // 输出经过的时间
    timer.out("Task completed");
}


int main(){
    // using namespace std::literals::chrono_literals;

    // auto start = std::chrono::high_resolution_clock::now();
    // std::this_thread::sleep_for(1s);
    // auto end = std::chrono::high_resolution_clock::now();
    // std::chrono::duration<float> duration = end - start;
    // std::cout << duration.count() << "s " << std::endl;

    Function(); // 0.09358ms

    std::cin.get();
}
// g++ time_chrono_ex.cpp -o time_chrono_ex
// ./time_chrono_ex
```

3-52、C++ 排序

1、std::sort

3-53、C++类型相关

1、C++是强类型语言，不是所有东西都用auto去声明，C++ 有一个类型系统，创建变量的时候，必须声明是整数/双精度数/布尔数/结构体/类。

2、C++ 中，虽然类型是由编译器强制执行的，但可以直接访问内存。

3、类型绕过。类型双关，不改变地址，更高效率。

3-54、联合体union

1、一个联合体只能有一个成员。给同一个变量取两个不同的名字。

3-55、C++的类型转换

1、C语言风格：隐式与显式类型转换

2、C++语言风格：静态转换static_cast<> / 动态转换 dynamic_cast<>/常量转换 const_cast<>/ 重新解释转换 reinterpret_cast<>

```
#include<iostream>


class Base {
    virtual void foo() {}
};

class Derived : public Base{};
int main(){

    // 静态转换（static_cast）：
    // 用于基本类型之间的转换，以及具有继承关系的类之间的转换。
    // 在编译时进行检查，相对安全。
    int a = 10;
    double b = static_cast<double>(a);

    // 动态转换（dynamic_cast）
    // 用于类层次结构中的多态类型转换，需要有虚函数。
    // 在运行时进行检查，只能用于指针或引用。
    Base* basePtr = new Derived();
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);

    // 常量转换（const_cast）：
    // 用于添加或删除变量的常量性。
    // 主要用于指针或引用
    const int c = 10;
    int* d = const_cast<int*>(&c);

    // 重新解释转换（reinterpret_cast）：
    // 用于不同类型之间的二进制位的转换，通常用于指针之间的转换。
    // 非常危险，慎用。
    // int* ptr = reinterpret_cast<int*>(someVoidPointer);


    std::cin.get();
}
// g++ type_conversion_ex.cpp -o type_conversion_ex
// ./type_conversion_ex
```

3、dynamic_cast（动态转换类型）：为何可以检查，因为存储了运行时类型信息（rtti: runtime type information).

3-56、c++的预编译头文件

1、使用预编译头文件：加速编译时间；stdafx.h； stdafx.cpp

2、C++的预编译头文件（Precompiled Header）是一种优化编译时间的机制，它允许将常用的头文件提前编译并保存为二进制文件，以便在后续编译过程中加速编译。预编译头文件通常使用文件扩展名`.pch`（Precompiled Header）。

3-57、C++17新特性：如何处理OPTIONAL数据 / 单一变量存放多种类型的数据variant/ 存储任意类型的数据 any

1、std::optional， C++17引入的模板类，用于表示一个可能包含值的容器。

2、存放多种类型的数据：union； C++17引入的`std::variant`或`std::any`；

3、union：空间大小为其中最大的元素所占空间大小； std::variant ， 空间大小为所有类型所占空间大小的和，可以理解为创建了一共结构体或类，将不同的数据类型存储为那个类或结构体中的成员。

3-58、多线程

1、C++11的库特性：std::async`是 C++11 标准引入的一个工具，用于支持异步任务的启动。它允许你创建一个异步任务，并返回一个`std::future`对象，用于获取异步操作的结果。

异步任务: 是指可以在后台执行而不阻塞主线程的任务。在编程中，异步任务通常用于执行耗时的操作，以确保主线程能够继续执行其他任务而不被阻塞。

```
#include <iostream>
#include <future>
#include <chrono>

int add(int a, int b) {
    // 模拟一个耗时的操作
    std::this_thread::sleep_for(std::chrono::seconds(20));
    return a + b;
}

int main() {
    // 启动异步任务
    // std::future<int> result = std::async(add, 2, 3);

    // 使用 std::launch::async 强制立即启动异步任务
    std::future<int> result = std::async(std::launch::async, add, 2, 3);

    // 使用 std::launch::deferred 延迟启动异步任务
    // std::future<int> result = std::async(std::launch::deferred, add, 2, 3);

    // 在这里进行其他操作，主线程不会被阻塞
    for (int i = 0; i < 5; ++i) {
        std::cout << "Main thread is doing other work..." << std::endl;
```

```
        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    // 获取异步任务的结果
    int sum = result.get();

    std::cout << "Sum: " << sum << std::endl;

    return 0;
}
// g++ -pthread async_ex.cpp -o async_ex
// ./async_ex
```

2、互斥锁 std::mutex,

3-59、让字符串（string）更快

1、std::string_view, C++ 17引入的一个新类，一种轻量级、非拥有、不可变的字符串视图类。它提供了对字符串的非拥有引用，允许你轻松地在不复制字符串的情况下对其进行操作。

2、小字符串优化(sso): 不进行堆分配，只分配一小块基于栈的缓冲区。 小于string中设置的字符最大数量（15）就是小字符串，不进行堆分配，const char*。

3-60、C++可视化基准测试

1、chrome://tracing/ 进行计时与与线程分析

3-61、C++的单例模式

1、单例：一个类的单一实例。单例模式是一种设计模式，确保一个类只有一个实例，并提供一个全局访问点。在 C++ 中，有几种方式可以实现单例模式，其中最常见的是使用静态成员函数和静态成员变量。

2、面向对象编程：拥有对象和类，可以多次实例化。

3-62、跟踪内存分配

1、RAM: RAM 是计算机中的随机访问存储器（Random Access Memory）的缩写，也被称为内存。RAM 是一种临时存储设备，用于存储正在运行的程序和系统需要的数据，以及当前正在执行的进程的相关信息。

RAM 是一种易失性存储器，这意味着在关闭计算机电源时，其中的数据会被清除。相对于长期存储设备（如硬盘驱动器或固态硬盘），RAM 提供了更快的读写速度，因为它是直接存取的，而不是按照地址顺序访问的。

2、跟踪内存分配和释放：**使用重载的 `new` 和 `delete` 运算符**（这种方法主要用于调试目的，不建议在生产代码中使用）

3-63、持续集成

1、持续集成（Continuous Integration，CI）：多个平台上运行，多个配置，多人开发；持续集成帮助我们自动化整个过程，确保代码在所有平台和所有配置下都可以编译，然后运行一些基本的测试或自动化测试。 提交、构建代码、运行。

2、 持续集成环境：Jenkins，

3-64、C++ 静态分析

1、静态分析：（代码复查）检查代码，检测错误。工具如：**Cppcheck**/ PVS Studio / **Clang Static Analyzer** /

3-65、参数求值顺序

1、参数计算顺序：未定义。C++17添加新规则，不能并行同时计算，必须一个接一个完成。

> 函数参数的求值顺序是未定义的（`Undefined Behavior`）。这意味着编译器可以按照任何顺序对函数参数进行求值，而不受特定的规则约束。因此，不应该依赖于函数参数的求值顺序，因为这可能会导致不一致的结果，并且在不同的编译器或编译选项下可能表现出不同的行为。
>
> 最好的做法是不依赖于参数求值顺序，以确保代码的可移植性和一致性。

3-66、C++ 移动语义

1、左值/右值； 左值引用/右值引用

> 左值通常表示可以标识内存位置的表达式，而右值通常表示临时值或无法标识内存位置的表达式。
>
> 左值引用（`lvalue reference`）：
> 使用 `&` 符号声明。
> 绑定到左值，延长左值的生命周期。
> 主要用于传递和修改左值。
>
> 右值引用（`rvalue reference`）：
> 使用 `&&` 符号声明。
> 绑定到右值，允许修改右值，也可以实现移动语义。
> 主要用于支持移动语义和完美转发。

2、移动语义：移动语义是C++11引入的一个特性，旨在提高资源管理的效率，尤其是在涉及动态内存分配的情况下。移动语义的核心是通过将资源的所有权从一个对象转移到另一个对象，而不是复制资源，从而避免不必要的开销。移动语义主要通过右值引用（rvalue reference）来实现。

移动语义中，通过使用 `std::move` 将一个对象的资源所有权转移到另一个对象。这通常涉及将资源的指针或资源管理对象的指针从一个对象“移动”到另一个对象，而不是复制整个资源。

```
#include <iostream>
#include <vector>

class MyString {
public:
    MyString(const char* data) : data_(nullptr) {
        if (data) {
            size_ = strlen(data);
```

```
            data_ = new char[size_ + 1];
            strcpy(data_, data);
        }
    }

    // 移动构造函数
    MyString(MyString&& other) noexcept : data_(other.data_), size_(other.size_)
{
        other.data_ = nullptr;
        other.size_ = 0;
    }

    // 移动赋值运算符
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }

    ~MyString() {
        delete[] data_;
    }

private:
    char* data_;
    size_t size_;
};

int main() {
    MyString source("Hello, World!");

    // 移动构造
    MyString destination1 = std::move(source);

    // 移动赋值
    MyString destination2;
    destination2 = std::move(source);

    return 0;
}
```

3、std::move 与 移动赋值操作符（当我们把一个变量赋值给一个已有的变量时才会被调用）。

3-67、C++ 构建桌面应用程序

1、QT；

2、 imgui;

四、进阶

Code source: