

快捷键：

CTRL+F7 单独编译某一文件

一、 C++是如何运行的

“<<”重载运算符

```
std::cout << "hello world!" << std::endl;
```

可理解为将字符串一个个推送到 cout，然后打印到终端。

预处理文件和 asm 汇编文件

编译后的文件：属性→C/C++→Preprocessor→Preprocess to a file

汇编文件：属性→C/C++→Output Files→Assembly Output→/FA

#INCLUDE 原理

其原理就是将文件复制粘贴在相应位置，所以在多个 cpp 中定义时，注意此处可能出现的错误。

二、 C++变量

Float 类型

在赋值是要加 f 才可定义为浮点型，否则就是 double 型。

Sizeof

用以查看各个数据类型所占字节

Include 头文件

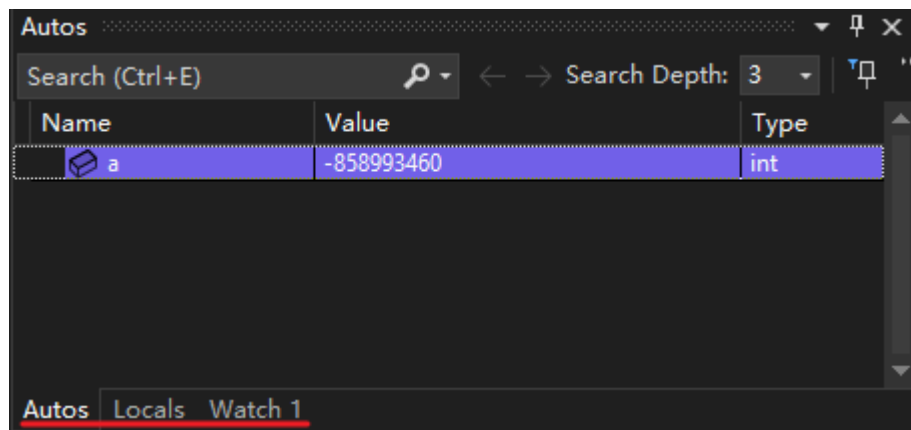
头文件不可重复定义，使用预处理关键则和`#ifndef`都可。

三、DEBUG

F9 打断点或点击左侧即可。

指针悬浮可查看详细信息

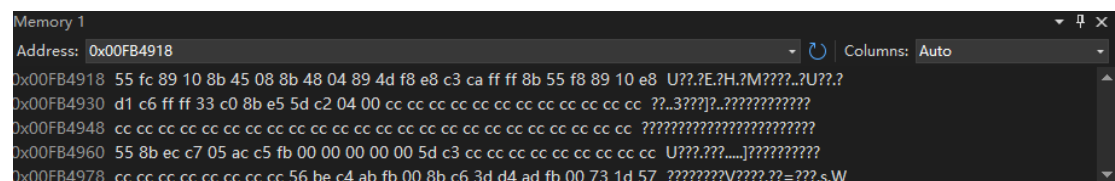
内存视图与内存



内存视图

DEBUG→Windows→Memory

在 Address 中输入&+变量可以找到相应的地址 (如&a: CC 就是未初始化的内存, 在视图中右键点击十六进制显示可看到为 CCCCCC…)



内存

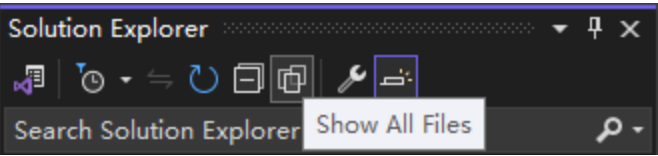
反汇编

调试过程中右键→Go To Disassembly 即可查看汇编代码

四、 VS 设置

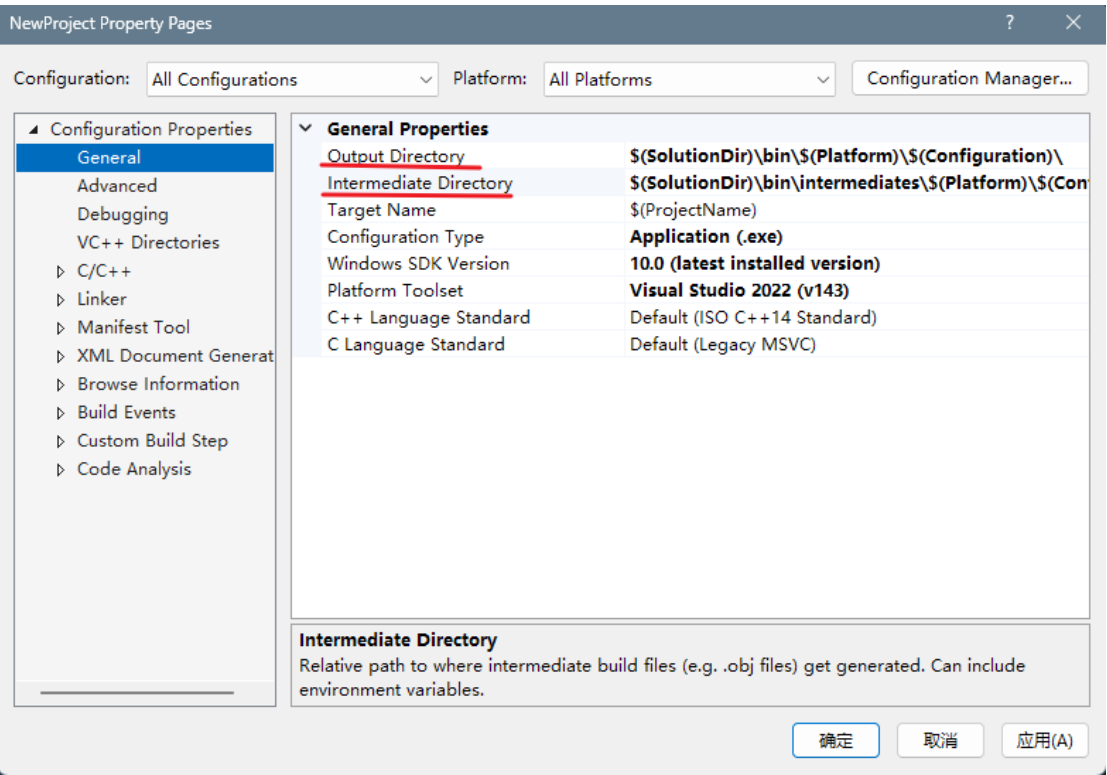
过滤器

此图标可以打开和关闭过滤器（显示和关闭真实目录）



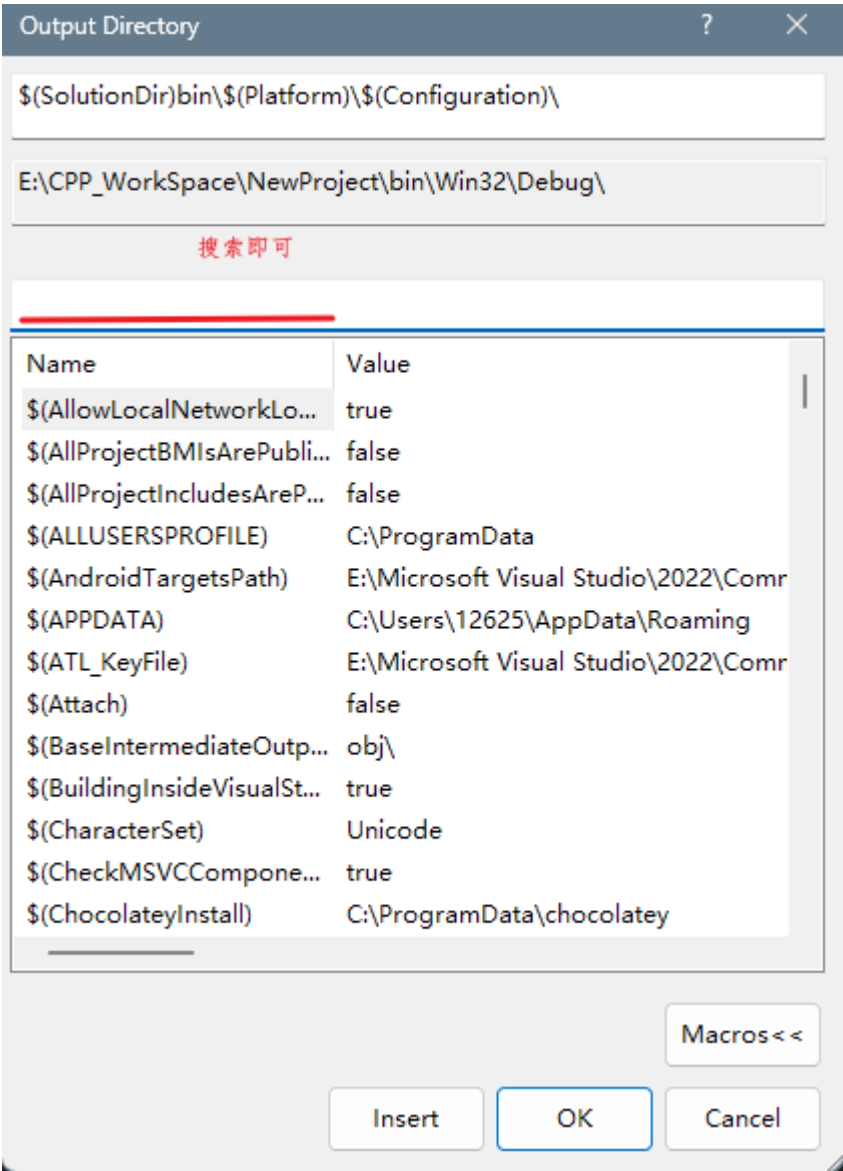
修改文件路径

修改输出文件路径和中间文件路径



查询宏指代

General Properties	
Output Directory	\$(SolutionDir)bin\\$(Platform)\\$(Configuration)\
Intermediate Directory	<Edit...>
Target Name	<Browse...>
Configuration Type	<inherit from parent or project defaults>



五、 指针

- 一个指针只是一个地址，是一个保存内存地址的整数。
- 与前方数据类型无关，其数据类型在修改指针内所存储的值时才会有作用（只有操作的时候数据类型才会起作用）。

- 定义格式： 在数据类型后添加*，是数据类型的一部分。 &取址符 *取值符

```
char *buffer
```

- 指针本身是变量，因此变量也储存在内存中，所以有双重指针和三重。

```
char **ptr = &buffer;
```

- 注意定义的时候指针和变量，和两个指针。

```
private:
    int *m_X, m_Y;
    int *m_X, *m_Y;
```

六、 引用（与指针相似）

- 不等同于指针，指针可以创建一个新的指针变量，然后设置它等于空指针或者类似的东西；引用不是这样，是引用已经存在的变量，引用本身不是新的变量，不占用内存，没真正的存储空间。
- 格式：数据类型+&，为数据类型的一部分。

```
int a = 5;
int& ref = a;
```

- 不管出于什么原因，ref 就是 a，只是 a 的别名，且编译后 ref 实际并不存在。
- 注意逻辑上的问题（并非是 ref 变为 b 的引用），定义引用时就要初始化（赋值）。因此引用无法更改引用对象，只有指针可以做到。

```
int a = 5;
int b = 8;

int& ref = a;
ref = b;

// a = 8, b = 8
```

函数的应用

如果不带引用，实际上只是创建了一个局部变量而并没有使得 a 的值加 1。
因此，如果需要修改其值的话。

```

void Increment(int value)
{
    // 如果参数不带引用，等效于添加
    // int value = 5;
    value++;
}

int main()
{
    int a = 5;
    Increment(a);
}

```

1. 指针（参数传递地址）

```

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

```

int *value

2. 引用（参数直接传 a 即可）

```

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

```

七、 类

- 定义格式：class 名字

```

{
}; //有分号

```
- 由类类型构成的变量成为对象，新的对象变量称为实例。

```
class Player
{
    int x, y;
    int speed;
};
int main()
{
    Player player; ///实例化了一个player对象。
    std::cin.get();
}
```

可见性

- 是一个属于面向对象编程的概念，它指的是类的某些成员或方法实际上有多可见即，谁能看到、调用、使用。
- 对程序实际运行方式完全没有影响的东西，即性能和类似的东西。
- 默认情况下，一个类中的所有东西都是私有的，也即只有类中的函数才能访问这些变量。
- 类可以包含函数，类内的函数成为方法。
- 虽然类非常有用，可以使得代码简洁，但如果不类搞不定的事情用类也一样搞不定，也既是类只是用以美化代码，方便调试的;类不会给你任何新的功能，用类搞定的事情不用类也一样可以搞定。

1. Public

Public 意味着我们可以在任何地方访问这些变量。

2. private

- private 意味着只有这个类和它的友元可以访问这些变量。
- 在 C++中有 friend 关键字，它可以让类或者函数成为另外一个类的友元，可以访问其私有成员。

3. protected

- 比 private 可见比 public 不可见，介于中间。
- 这个类和层次结构中的所有子类可以访问这些符号，但其他情况不允许访问例如在其子类中可以访问但 main 函数中不可访问。

类和结构体

基本上没什么区别，在 C++ 中两者都有可见性，继承等。

- 类中默认是 `private` 而结构体的默认是 `public`。
- 建议：结构体只是数据的结构，仅此而已，不适用继承；而使用类则包含继承，方法等多种功能性。

八、 Static 和 extern

Static 关键字

有两种意思，取决于上下文。

翻译单元：编译过程中的基本单元。它包含一个源文件及其包含的所有头文件，但不包括那些被条件预处理语句（如 `#if`）排除的部分。每个翻译单元都会被编译器独立编译，然后由链接器将它们合并成最终的程序。简而言之，翻译单元就是编译器在单次编译操作中处理的完整源代码集合。

在类或结构体外部使用 static 关键字

类外面的 `static`，意味着你声明为 `static` 的符号，链接将只是在内部，意味着它只能对你定义它的翻译单元（单个文件）可见。

这个变量只会在翻译单元内部链接（类似于作用域）。

```
static int s_variable = 5;
```

在类或结构体内部使用 static 关键字

类内部的静态变量，意味着该变量实际上将与类的所有实例共享内存，这意味着该静态变量在你在类中创建的所有实例中，静态变量只有一个实例（共享内存）。

例如：我创建一个名为 `Entity` 的类，我不断创建 `Entity` 实例，我仍然只会得到那个变量的一个版本；如果某个实例改变了这个静态变量，那么它在所有的实例中都将改变。

注意事项

内部静态变量

在类或结构体内部使用 `static` 关键字时，需要在某处定义静态变量。


```

struct Entity
{
    static int x, y;

    void Print()
    {
        std::cout << x << "," << y << std::endl;
    }
};
int Entity::x;
int Entity::y;

```

因此使用对象进行改变静态变量是没有意义的，我们可以使用以下，就像他们在这个 Entity 作用域内一样。类似于在名为 Entity 的命名空间中创建了两个变量，他们实际上并不属于类。（因此不用实例化类）

```

Entity::x = 3;
Entity::y = 4;

```

内部静态方法

如果我们让 x,y 保持非静态而 print 为静态方法，那么就会出 bug。因为静态方法无法访问非静态变量，原因是静态方法没有类实例。

```

struct Entity
{
    int x, y;

    static void Print()
    {
        std::cout << x << "," << y << std::endl;
    }
};

```

以下我们还原为什么不能引用，上述 Print 静态方法等价于以下外部函数。不知道想要访问哪个 Entity 的 x 和 y，因为没有给它一个 Entity 的引用。

```

static void Print()
{
    std::cout << e.x << ", " << e.y << std::endl;
}

```

与普通方法的区别

静态方法的好处包括：

内存效率：由于静态方法不需要对象实例，它们不需要为每个对象实例分配内存。

全局访问：静态方法可以通过类名直接访问，无需创建对象实例。

封装性：静态方法可以用来封装与对象状态无关的功能，使得这些功能可以作为工具函数使用。

与普通方法（实例方法）的区别：

调用方式：静态方法可以通过类名直接调用，而普通方法需要通过对象实例调用。

访问限制：静态方法不能访问类的非静态成员变量和非静态方法，因为它们需要一个对象的上下文。

```
class Example {
public:
    static void staticMethod() {
        // 静态方法内容
    }

    void instanceMethod() {
        // 普通方法内容
    }
};

// 调用静态方法，无需创建对象实例
Example::staticMethod();    不能在成员函数 "Example::st

// 创建对象实例来调用普通方法
Example obj;
obj.instanceMethod();    此声明没有存储类或类型说明符
```

Local static（局部静态）关键字

变量的生存期：在被删除之前，它会在我们的内存中存在多久。

变量的作用域：我们可以访问变量的范围。

静态局部变量允许我们声明一个变量，其生存期相当于这个程序的生存期；其作用域限制在这个某个区域内（看在哪里定义）。

在类作用域中，类中的任何东西都可以访问这个静态变量；如果阿紫函数作用域中声明一个静态变量，那么它将是那个函数的局部变量。

单实例（在类中实例化）

定义：单例模式确保一个类只有一个实例，并提供一个全局访问点来访问该实例。在类中实例化是一种实现方式，通常称为**饿汉式**单例模式，这种方式会在类加载时立即初始化这个实例。但还有其他实现方式，如**懒汉式**，它会延迟实例化，直到第一次调用 `getInstance()` 方法时才创建实例。

一般方法

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

    void Hello(){}
};

Singleton *Singleton::s_Instance = nullptr;

int main()
{
    Singleton::Get().Hello(); /// 在类空间中GET返回指针，然后再运行Hello。

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

局部静态

```

class Singleton
{
public:
    static Singleton& Get() /// 返回引用
    {
        static Singleton instance; /// 不加静态则函数就会被删除，会报错。在主函数中无法调用。
        return instance;
    }

    void Hello(){}
};

int main()
{
    Singleton::Get().Hello();

    std::cin.get();
}

```

注意事项

如果你不需要变量是全局变量，就需要尽可能多地使用静态变量。因为一旦你在全局作用域下申明东西的时候，如果没有设定 static，那么编译器会跨编译单元进行链接，可能会造成一些 bug，因此除非真的需要它们跨翻译单元链接，否则就让函数和变量标记为静态的。

extern 关键字

与 static 相对应，既是在外部翻译单元寻找变量的定义。

但如果外部变量带有 static 关键字，则 extern 也无法找寻到，即其他所有的翻译单元都不能看到带有 static 关键字的变量。

九、 枚举

- 其本质是一个数值的合集
- 里面的数只为整数

```
enum Example : unsigned char
{
    a=1,b,c;
};

int main()
{
    Example value = 5;    "int" 类型的值不能用于初始化枚举类型，只能为整数因此不可为float等类型
                           只能说枚举类的数或者标识符

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

- 格式：含有限制条件，枚举类型初始化时候只能赋予所枚举出来的值。

十、构造函数

- 构造函数基本上是一种特殊类型的方法，它在每次实例化对象时运行，是一种每次你构造一个对象时都会调用的方法。
- 通常用于设置变量或者做初始化。
- 无返回类型，且它的名称必须与类的名称相同。

```
class Entity
{
public:
    float x, y;

    Entity()
    {
        x = 0.0f;
        y = 0.0f;
    }

    void Print()
    {
        std::cout << x << "," << y << std::endl;
    }
};
```

- 如果不指定构造函数，代码中仍然有一个构造函数，名为默认构造函数，实际上什么都没做。
- 如何不实例化对象，将不会运行。所以如果只是用一个类的静态方法，它将不会运行。

- 当将构造函数设置为 private 时，或者删除构造函数，将不能进行实例化。

```
class Log
{
public:
    Log() = delete;
}

class Log
{
private:
    Log() {}
}
```

函数重载

可以写多个构造函数，前提是它们有不同的参数：即有相同的函数（方法）名，但是有不同的参数的不同函数版本。

即可以选择使用参数来构造 Entity 对象，以下是多个构造函数例子。

```
class Entity
{
public:
    float x, y;

    Entity()
    {
        x = 0.0f;
        y = 0.0f;
    }
    Entity(float x, float y)
    {
        x = x;
        y = y;
    }

    void Print()
    {
        std::cout << x << "," << y << std::endl;
    }
};

int main()
{
    Entity e(5.0f, 10.0f);
    std::cout << e.x << std::endl;
    e.Print();
    std::cin.get();
}
```

十一、析构函数

- 在销毁对象时，析构函数将被调用，格式：类名前加~。
- 常用于写在变量并清理你内存。

- 适用于栈和堆分配的对象。使用 new 分配一个对象当 delete 时，析构函数会被调用；如果是一个栈对象，当作用域结束时，栈对象将被删除，析构函数也会被调用（在函数中实例化就是放在栈中）。

```
class Entity
{
public:
    float x, y;

    Entity()
    {
        x = 0.0f;
        y = 0.0f;
        std::cout << "Created Entity!" << std::endl;
    }
    ~Entity()
    {
        std::cout << "Destroyed Entity!" << std::endl;
    }
}
```

- 也可手动调用析构函数（e.~Entity），但不推荐。

虚析构函数

- 虚析构函数的意思不是覆写析构函数，而是加上一个析构函数。因此，如果我把基类析构函数改为虚函数，它实际上会调用两个析构函数，先调用派生类析构函数，然后层次结构向上，调用基类析构函数。
- 在这种情况下可以看到第三种并没有调用 Derived 的析构函数，会造成内存泄露。

<pre>class Base { public: Base() { std::cout << "Base Constructor\n"; } ~Base() { std::cout << "Base Destructor\n"; } }; class Derived : public Base { public: Derived() { std::cout << "Derived Constructor\n"; } ~Derived() { std::cout << "Derived Destructor\n"; } }; int main() { Base *base = new Base(); delete base; std::cout << "-----\n"; Derived *derived = new Derived(); delete derived; std::cout << "-----\n"; Base *poly = new Derived(); delete poly; std::cin.get(); }</pre>	<pre>Base Constructor Base Destructor ----- Base Constructor Derived Constructor Derived Destructor Base Destructor ----- Base Constructor Derived Constructor Base Destructor</pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- 因此我们需要在基类在析构上设置为虚函数，这意味着这个类有可能被拓展成子类，可能还有一个析构函数也需要被调用。
 - ◆ 在 C++ 中，声明虚函数允许实现多态性。这意味着当基类指针或引用指向派生类对象时，可以调用实际对象类型的方法，而不仅仅是基类中定义的方法。这是通过虚函数表（vtable）实现的，它是一个在运行时用来解析函数调用的机制。
 - ◆ 当基类中的函数被声明为虚函数时，编译器会为该类生成一个虚函数表。每个类都有自己的虚函数表，其中包含了指向虚函数实现的指针。如果派生类重写了这些函数，它的虚函数表会被更新为指向新的函数实现。当通过基类指针或引用调用虚函数时，程序会查看虚函数表来确定应该调用哪个函数实现。

```
class Base
{
public:
    Base() { std::cout << "Base Constructor\n"; }
    virtual ~Base() { std::cout << "Base Destructor\n"; }
};
```

十二、继承

- 任何在 Entity 类中不是私有的东西，实际上都可以被 Player 访问，player 是 Entity 的超集

```
class Entity
{
public:
    float x, y;

    void Move(float xa, float ya)
    {
        x += xa;
        y += ya;
    }
};

class Player : public Entity
{
public:
    const char *Name;
    void PrintName()
    {
        std::cout << Name << std::endl;
    }
}
```

```
class Player : public Entity , Printable
```


十三、虚函数

- 虚函数允许我们在子类中重写方法。（假设我们有两个类 A 和 B，B 是 A 派生出来的，即 B 是 A 的子集。如果我们在 A 类中创建一个方法，标记为 virtual，那么我们可以选择在 B 类中重写那个方法，让它做其他事情。）
- 虚函数的应用并不是无耗的
 - 会增加内存，需要额外的内存来存储 V 表，基类中要有成员指针指向 V 表。
 - 每次调用虚函数时，需要遍历 V 表，来确定要映射到哪个函数。
- 应用例子：
 - 在这个例子中，我们所期望的是第一次打印 entity 第二次打印 LV，但实际上两次都是 entity（因为 Player 包含 Entity 所以理论上 Entity 指针也可以指向 Player 类即第 35 行可改为 Entity* p = ...）原因是 26 行函数的参数是 Entity 类，也即是当我们调用 GetName 时，如果是在 Entity 里面，那么就会在 Entity 里找这个 GetName 函数。
 - 当我们希望第二个传递的 Entity 实际上是 Player，请调用 Player 里面的 GetName 函数，这就是虚函数出现的地方，引入了一种叫做 Dynamic Dispatch（动态联编）的东西，通过 V（虚函数表）表来实现编译。

```
1  class Entity
2  {
3  public:
4      std::string GetName()
5      {
6          return "entity";
7      }
8  };
9
10 class Player : public Entity
11 {
12 private:
13     std::string m_Name;
14 public:
15     // 构造函数 初始化的写法，相当于把name赋值给m_Name
16     Player(const std::string& name): m_Name(name)
17     {
18     }
19
20     std::string GetName()
21     {
22         return m_Name;
23     }
24 };
25
26 void PrintName(Entity* entity)
27 {
28     std::cout << entity->GetName() << std::endl;
29 }
30
31 int main()
32 {
33     Entity *e = new Entity();
34     PrintName(e);
35     Player *p = new Player("LV");
36     PrintName(p);
37
38     std::cin.get();
39 }
```

- 修改后既是

```
class Entity
{
public:
    virtual std::string GetName()
    {
        return "entity";
    }
};
```

Override 关键字

- 实际上我们还需要在子类覆写方法后添加 override。
 - 这不是必须的但可以增加我们的可读性。
 - 其次当与父类拼写不同因为手误等情况还会报错。

```
class Player : public Entity
{
private:
    std::string m_Name;
public:
    // 构造函数 初始化的写法, 相当于把name赋值给m_Name
    Player (const std::string& name): m_Name(name)
    {
    }
    std::string GetName() override
    {
        return m_Name;
    }
};
```

接口 (纯虚函数)

- 纯虚函数允许我们在基类中定义一个没有实现的函数，然后强制子类去实现该函数，不实现就无法实例化。
- 在基类中含有 virtual 关键字且等于 0 本质上使它成为一个纯虚函数，意味着这个基类不能被实例化，如果你想实例化这个子类的话，它必须在一个子类中实现，即纯虚函数必须被实现才能创建这个类的实例。

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

十四、数组

- 数组基本上是元素的集合，按特定的顺序排列的一堆东西，表示一堆变量组成的集合。
- For 循环中避免等于操作，会消耗性能。
- 指针指向的是数组的首元素地址，此处 `ptr+2` 是移动 2×4 个字节，因为 `ptr` 本身是 `int` 型。

```
int main()
{
    int example[5];
    int* ptr = example;
    for (int i = 0; i < 5; i++)
    {
        example[i] = 2;
    }
    *(ptr + 2) = 6;
    std::cin.get();
}
```

- 因此我们可以改为 `char` 型再改回 `int` 型指针。`Char` 型占据一个字节，因此需要+8。

```
*(int*)((char*)ptr + 8) = 6;
```

- 在类中使用变量去定义数组个数时会报错，在 C++ 中，成员数组的大小必须是常量表达式，因为数组的大小在编译时需要确定。在您的代码中，`size` 被定义为一个 `const` 整数，但它是一个非静态成员变量，这意味着它属于类的对象实例，而不是类本身。因此，每个 `Eneity` 对象都会有自己的 `size` 副本，这不满足数组大小的编译时常量要求。

```
class Eneity
{
public:
    const int size = 5;
    int example[size]; 非静态成员引用必须与特定对象相对
};
```

改为静态就可以。`size` 就成为了类的一个静态成员，它在所有 `Eneity` 对象之间共享，并且在编译时是已知的，满足数组大小的要求。

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

Std::array 关键字

- 类型安全：`std::array` 是一个容器类，提供了类型安全的接口。这意味着它会在编译时检查类型错误，而普通数组不会。

- 大小信息：std::array 内部存储了数组的大小信息，可以通过 .size() 方法直接获取，而普通数组需要手动管理大小信息。
- 值语义：std::array 具有值语义，可以像其他标准容器一样被复制和赋值。普通数组在赋值时需要手动复制每个元素或使用 std::copy。
- 迭代器支持：std::array 支持 STL 迭代器，可以使用标准的迭代器操作和算法，而普通数组通常使用指针进行迭代。
- 性能：std::array 是对普通数组的薄包装，所以在性能上与普通数组几乎相同，储存在栈中。

使用实例：

```
class Entity
{
public:
    static const int size = 5;
    int example[size];
    std::array<int, 5> another;

    Entity()
    {
        for (int i = 0; i < another.size(); i++)
        {
            ;
        }
    }
};
```

动态数组(std::vector)

- 当所给予的元素大于所创建的，它会在内存中创建一个比第一个大的新数组，把所有东西复制到这里，然后删除旧的那个。
- 是内存连续的数组。
- 效率不是第一位，一般会自己优化效率。

For 循环的特殊用法

- ◆ 范围 for 循环，允许遍历一个容器中的所有元素，不需要迭代器或者索引。
- ◆ **declaration** 是用于遍历集合中每个元素的变量声明。
- ◆ **expression** 是要遍历的集合。

```
for (declaration : expression)
{
    // 循环体
}
```

- ◆ Vertex v 是循环变量声明，它在每次迭代时都会被赋予 vertices 容器中的下一个 Vertex 对象。
- ◆ vertices 是要遍历的容器。

```

struct Vertex
{
    float x, y, z;
};

std::ostream& operator<<(std::ostream& stream, const Vertex& vertex)
{
    stream << vertex.x << "," << vertex.y << "," << vertex.z;
    return stream;
}

int main()
{
    std::vector<Vertex> vertices;
    vertices.push_back({1, 2, 3});
    vertices.push_back({4, 5, 6});
    for (int i = 0; i < vertices.size(); i++)
    {
        std::cout << vertices[i] << std::endl;
    }
    for(Vertex v :vertices)
    {
        std::cout << v << std::endl;
    }
    for(Vertex& v :vertices) // 通过引用可以不用复制
    {
        std::cout << v << std::endl;
    }
    vertices.clear(); // 将数组大小设为0
    vertices.erase(vertices.begin() + 1); // 删除第二个元素
    std::cin.get();
}

```

Std::vector 优化

- 第一次 push 的时候就已经打印了 copied，Vertex 对象首先在 main 函数的栈上构造，然后通过拷贝构造函数复制到 vertices 中。这是因为 push_back 方法接受一个 Vertex 类型的参数，当你传递一个临时对象给它时，会调用拷贝构造函数来创建 vector 中的对象副本。
- 第二次会弹出两个 copied，通过 VS 可以看到动态数组容量变为了 2，所以它在调整它的大小，重新复制它的所有元素因此有两次 copied。
- 同理第三次会弹出三个一共六次。
- 因此有些计算机如果分配内存碰巧可以装下三个的时候，结果会只有三个 copied，理由是第一条。
- **优化策略：**
 - 我们可以在适当的地方构造 vertex。(reserve)
 - 提前预留好适当内存避免重新分配。(emplace_back)，emplace_back 不是传递我们已经构建的 vertex 对象，我们只是传递了构造函数的参数列表，在我们实际的 vector 内存中，使用以下参数，构造一共 vertex 对象。

迭代器

- 在 C++ 中，迭代器是一种类似于指针的对象，它允许我们遍历容器（如数组、向量、

列表等) 中的元素。迭代器提供了一种方法来访问容器中的连续元素, 而不需要了解容器的内部实现细节。

- 举个例子, 假设我们有一个 `std::vector<int>` 类型的向量, 我们可以使用迭代器来遍历它的所有元素:
 - 在这个例子中, `vec.begin()` 返回一个指向向量第一个元素的迭代器, 而 `vec.end()` 返回一个指向向量最后一个元素之后位置的迭代器。我们通过增加迭代器 `it` 来遍历向量, 使用 `*it` 来访问它所指向的元素的值。这就是迭代器的基本用法。

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    // 使用迭代器遍历向量
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << ' ';
    }
    return 0;
}
```

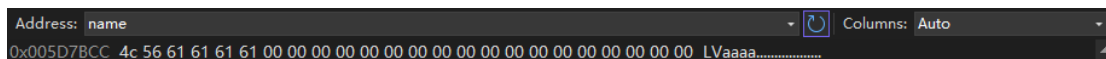
十五、字符串

- 一种能够表示和处理文本的方法
- 字符串实际上是字符数组

定义 (C): `char*` 是用双引号

```
const char *name = "LV";
```

- 在内存中发现后方有 00, 这被称为空终止字符。



- 定义 (C++): C++ 标准库中有名为 `String` 类, 在 C++ 中使用字符串, 应该使用 `std::string`.
 - 基本上是一个字符数组和一些操作这个字符数组的函数
 - 默认接收的参数为 `const`, 因为通常字符串不会被改变。

```
std::string name = "LVaaaa";
```

- 不使用 `new`, 就不使用 `delete`。

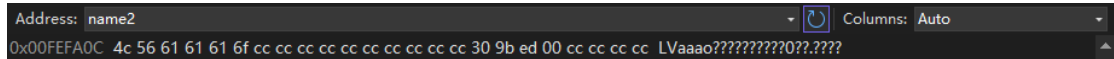
字符

- 一般来说在 C++ 中, `Char` 类型为一个字节。
- 字符通过单引号: 此处因为无终止符号, 因此打印出来会出现乱码。

```
char name2[6] = {'L', 'V', 'a', 'a', 'a'};

std::cout << name2 << std::endl;
```

内存视图如下：



添加终止符号即可。

```
char name2[7] = {'L', 'V', 'a', 'a', 'a', 'o', '\0'};
```

追加字符串

错误写法：逻辑上是想将两个 const char 的数组相加，双引号里面的是 const char 数组，不是真正的字符串，不能把两个指针相加。

```
std::string name = "LVaaaa"+"aaaaaa";
```

解决办法：分成两行，使用+=。是将一个指针，加到了 name，name 是一个字符串，你把它加到了字符串上，+=这个操作符在 string 类被重载了。

- std::string name = "LVaaaa"; name += "aaaaaa"; 这样写可以，因为首先创建了一个 std::string 对象 name，然后使用 += 运算符将 "aaaaaa" 字符串字面量附加到 name 对象上。

```
std::string name = "LVaaaa";
name += "aaaaaa";
```

- std::string name = "LVaaaa"+"aaaaaa"; 这样写不行，因为 "LVaaaa" 和 "aaaaaa" 都是字符数组，试图使用 + 运算符将它们连接起来时，实际上是尝试将两个指针相加，这在 C++ 中是非法的。当你写 "LVaaaa" 时，你得到的是一个指向这个数组第一个元素的指针。同样，"aaaaaa" 也是一个字符数组，其表达式会提供一个指向该数组第一个元素的指针。
- 如果你想在声明时连接两个字符串字面量，你可以先将其中一个转换为 std::string 对象，然后使用 + 运算符，如下所示：

```
std::string name = std::string("LVaaaa") + "aaaaaa";
```

- 这样，第一个字符串字面量 "LVaaaa" 被转换为 std::string 对象，然后使用 std::string 类的 + 运算符来连接第二个字符串字面量 "aaaaaa"。这是因为 std::string 类重载了 + 运算符来允许字符串连接。
- 在 C++14 中，可以引入一个库使得更方便的追加字符串（字符串字面量后加 s 即可），也可以修改任意的字符数据类型。

```
using namespace std::string_literals;
std::u32string name0 = U"Cherno"s + U"Hello";
std::string name0 = u8"Cherno"s + u8"Hello";
```

- 也可以前添加 R 以此来输入字符串什么样打印出来就什么样子。

```
const char *example = R"(line1
line2
line3
line4)";
// 等价于
const char *ex = "line1\n"
"line2\n"
"line3\n ";
std::cout << example << std::endl;
std::cout << ex << std::endl;
```

```
line1
line2
line3
line4
line1
line2
line3
```

字符串字面量

- 基于字符串的东西，是在双引号之间的一串字符。"Hello"就是一个字符串字面量，它实际上在内存中表示为{'H', 'e', 'l', 'l', 'o', '\0'}。字符串字面量通常存储在程序的只读数据段中，因此尝试修改它们的内容会导致未定义行为，通常是运行时错误。

```
"LvJiale";
```

- 而字符串字面量在 C++ 中是存储在只读内存区域的。因此，尝试修改字符串字面量的内容会导致未定义行为，通常会引发运行时错误。

```
char* name = "LvJiale";
name[2] = 'a';
std::cout << name << std::endl;
```

上述定义中可能会报错

- 在 C++ 中，char* name = "LvJiale";可能会报错，因为这种赋值方式试图将一个字符串常量的地址赋给一个非常量指针。字符串常量（如"LvJiale"）通常存储在只读的内存段中，而 char* 类型的指针默认指向可修改的内存。如果你试图通过这个指针修改字符串，可能会导致运行时错误。
 - ◆ 为了解决这个问题，可以使用 const char* 类型的指针，这样就明确表示指针指向的字符串内容是不可修改的
 - ◆ 另一方面，char* name = (char*)"LvJiale";不会报错，因为这里使用了强制类型转换(char*)，显式地告诉编译器将字符串常量的地址转换为一个非常量指针。这样做虽然可以避免编译时错误，但是如果通过这个指针修改字符串内容，仍然可能导致运行时错误，因为实际上它仍然指向只读内存。

修改:

```
char name[] = "LvJiale";  
name[2] = 'a'; // 这是合法的, 因为name是一个可修改的字符  
数组  
std::cout << name << std::endl; // 输出将会是  
"LvaJiale"
```

宽字符等其余字符变量

- **const char* name = u8"LvJiale";**: 这里使用的是 char 类型, 通常用于存储 UTF-8 编码的字符串。在大多数系统中, char 类型占用 1 个字节。UTF-8 是一种可变长度的编码, 可以使用 1 到 4 个字节来表示一个 Unicode 字符, 具体取决于字符的码点。
- **const wchar_t* name2 = L"LvJiale";**: wchar_t 是一个宽字符类型, 它的大小可以是 2 个字节或 4 个字节, 这取决于你的编译器和平台。在 Windows 上, wchar_t 通常是 16 位的, 用于存储 UTF-16 编码的字符。在其他一些平台上, wchar_t 可能是 32 位的, 用于存储 UTF-32 编码的字符(平台不同所占字节不同)。
- **const char16_t* name3 = u"LvJiale";**: char16_t 是 C++11 中引入的类型, 用于明确表示 UTF-16 编码的字符。它总是占用 2 个字节。
- **const char32_t* name4 = U"LvJiale";**: char32_t 也是 C++11 中引入的类型, 用于明确表示 UTF-32 编码的字符。它总是占用 4 个字节。

```
int main()  
{  
    const char* name = u8"LvJiale";  
    const wchar_t* name2 = L"LvJiale";  
    const char16_t* name3 = u"LvJiale";  
    const char32_t* name4 = U"LvJiale";  
}
```

十六、const

- 在改变代码生成方面做不了什么, 承诺这个东西是静态不改变。
- 也可以被强制改变 (强制类型转换符)。

指针 (引用) const 关键字

- 用法一: const 在*前
 - 因此有两种写法是同义的。

```
const int* a = new int;  
int const* a = new int;
```

- 指针是不能改变指针指向的内容而并非指针的指向。

```
int main()
{
    const int MAX_AGE = 90;

    const int *a = new int;
    *a = 2; 表达式必须是可修改的左值
    a = &MAX_AGE;
}
```

- 用法二：const 在*后

- 指针是不能改变指针的指向而可以改变指针指向的内容。

```
int* const a = new int;
*a = 2;
a = (int *)&MAX_AGE; 表达式必须是可修改的左值
std::cout << *a << std::endl;
```

类及方法种的 const 关键字

- 用法一：在参数列表之后

这个方法不会修改任何实际的类，因此可以看到外面不能修改类成员变量，只能读不能写。

```
class Entity
{
private:
    int m_X, m_Y;
public:
    int GetX() const
    {
        m_X = 2; 表达式必须是可修改的左值
        return m_X;
    }
}
```

- 案例：

为什么要写这个，因为有时候函数传参的时候是 const 在*或者&（引用）前面，即不可修改指向的值，而一旦类中参数列表后不加 const，就无法保证值不会被改写因此无法调用这个方法。因此有时候会有有无 const 两个版本方法。

<pre>class Entity { private: int m_X, m_Y; public: int GetX() { return m_X; } }; void PrintEntity(const Entity& e) { std::cout << e.GetX() << std::endl; }</pre>	<pre>class Entity { private: int m_X, m_Y; public: int GetX() const { return m_X; } }; void PrintEntity(const Entity& e) { std::cout << e.GetX() << std::endl; }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

如果真要在 const 方法中更改常量，可用 mutable 关键字。

- 三个 const 例子：

- 意味返回一个指针指向都不能被修改的指针，这个方法也承诺不修改类成员。

```
const int* const GetX() const
```

mutable 关键字

mutable 关键字意味可改变的，即使在 const 方法中也可以。

```
class Entity
{
private:
    int m_X, m_Y;
    mutable int var;

public:
    int GetX()
    {
        var = 2;
        return m_X;
    }
};
```

十七、成员初始化列表

在构造函数（通常）中初始化类成员（变量）的一种方式，多使用，比一般方法省性能。

- 一般方法

```
class Entity
{
private:
    std::string m_Name;
public:
    Entity()
    {
        m_Name = "Unkonwed";
    }
    Entity (const std::string& name)
    {
        m_Name = name;
    }
}
```

- 成员初始化列表

- 成员列表初始化顺序与成员声明要一致，不然就会导致各种各样的依赖问题。

```
private:
    std::string m_Name;
    int m_Score;

public:
    Entity()
    {
        :m_Name("Unkonwed"),m_Score(10)
    }
    Entity (const std::string& name, const int& score)
    {
        :m_Name(name),m_Score(score)
    }
```

十八、创建并初始化对象

- 在 C++ 中，我们要选择对象放在栈上还是堆上；在堆上一定要手动释放内存。
- 性能问题：在堆上分配要比在栈上分配花费更长的时间。
- 如何选择：如果对象太大或要显式地控制对象的生存期，那就堆上创建(new)。。
其余在栈上创建。
- 对于栈对象，有一个自动的生存期，由它声明的地方作用域决定的，只要变量超出作用域，也就是内存被释放了，因为当作用域结束的时候，栈会弹出作用域的东西，栈上的任何东西都会被释放；以下就是放在栈中的。

```
int main()
{
    Entity entity("Lvjiale");

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

- 对于堆对象 (new)，一旦在堆中分配一个对象，实际上你已经在堆上创建了一个对象。它会一直待在那里，直到你做出决定，如手动释放内存。

```
int main()
{
    Entity* entity = new Entity("Lvjiale");
    delete entity;
    std::cin.get();
}
```

十九、new

- 本质上是一个操作符，就像加、减、等于一样，意味着我们可以重载这个操作符，并改变它的行为。

```
int main()
{
    int *b = new int[50]; //200 bytes
    int *c = new int; //4 bytes

    Entity *E = new Entity();
    Entity *e = new Entity[50];
    std::cin.get();
}
```

- 在分配类时，它还调用构造函数。
- 使用 **new** 时必须使用 **delete**，delete 也是一个操作符，其调用了 C 函数中的 free。数组释放的时候要加 []。

```
int main()
{
    int *b = new int[50]; //200 bytes

    Entity *e = new Entity();
    delete e;
    delete[] b;

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

- New 在其底层就是调用 malloc，以下两个语句是基本相同的，区别就是 new 调用了 Entity 的构造函数，因此在 C++ 大部分情况使用 new。

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

Placement new

二十、隐式转换与 explicit 关键字

隐式转换

- 隐式的意思是不会明确地告诉它要做什么，而是通过上下文知道什么意思，C++ 允许编译器对代码执行一次隐式转换。
- 因为在构造函数中有单个相对应的参数，C++ 可以隐式的把 22 变成 Entity 类，构造出一个 Entity。而注释中的不可行，因为“Lvjjiale”是一个 char 类型的数组并非 std::string，如果要构造出 Entity 要进行两次隐式构造而 C++ 支持一次。

```
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()
{
    //Entity a = "Lvjjiale"
    Entity a = (std::string)"Lvjjiale";
    Entity b = 22;

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

- 类似的例子还有

```
void PrintEntity (const Entity& entity)
{
    //printing
}

int main()
{
    PrintEntity(22);
    PrintEntity(std::string("Lvjiale"));
}
```

- 对于强制类型转换符 C 和 C++ 风格写法。
 - ◆ (std::string)"Lvjiale" 是一个类型转换，将 C 风格的字符串字面量转换为 std::string 类型的对象。这种转换通常称为 C 风格转换或旧式转换。
 - ◆ std::string("Lvjiale") 是直接调用 std::string 的构造函数，创建一个新的 std::string 对象，其内容初始化为 "Lvjiale"。
 - ◆ 在实际使用中，直接调用构造函数 std::string("Lvjiale") 是更加现代和首选的方式，因为它更明确地表达了创建 std::string 对象的意图。而 C 风格的转换 (std::string)"Lvjiale" 虽然也可以达到相同的效果，但不推荐使用，因为它可能会导致类型转换的含义不够清晰，特别是在复杂的表达式中。

Explicit 关键字

- Explicit 禁用这个隐式 implicit 的功能。
- Explicit 关键字放在构造函数前面，意味着没有隐式的转换，即如果要使用整数构造这个 Entity 对象，则必须显式调用此构造函数。

```
public:
    Entity(const std::string &name)
        : m_Name(name), m_Age(-1){}
    explicit Entity(int age)
        : m_Name("Unknown"), m_Age(age){}
};

void PrintEntity (const Entity& entity)
{
    //printing
}

int main()
{
    PrintEntity(22);    不存在从 "int" 转换到 "Entity" 的适当转换
    PrintEntity(std::string("Lvjiale"));

    //Entity a = "Lvjiale"
    Entity a = (std::string)"Lvjiale";
    Entity b = 22;    不存在从 "int" 转换到 "Entity" 的适当转换
}
```

二十一、运算符及其重载

- 如果不用运算符及重载，则是 result1。
- 重载类似函数的写法，但注意有 operator 后带上需要重载的运算符，result2 是重载的写法。

```
struct Vector2
{
    float x, y;

    Vector2(float x, float y)
        :x(x),y(y){}
    Vector2 Add(const Vector2& other) const
    {
        return operator+(other);
    }
    Vector2 operator+ (const Vector2& other) const
    {
        // return Add(other); 这样的写法也可以
        return Vector2(x + other.x, y + other.y);
    }
    Vector2 Multiply(const Vector2& other) const
    {
        return Vector2(x * other.x, y * other.y);
    }
    Vector2 operator* (const Vector2& other) const
    {
        // return Multiply(other); 这样的写法也可以
        return Vector2(x * other.x, y * other.y);
    }
};

int main()
{
    Vector2 position(4.0f, 4.0f);
    Vector2 speed(0.5f, 1.5f);
    Vector2 powerup(1.1f, 1.1f);

    Vector2 result1 = position.Add(speed.Multiply(powerup));
    Vector2 result2 = position + speed * powerup;

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

- ==运算符重载。

```
bool operator==(const Vector2& other) const
{
    return x == other.x && y == other.y;
}

if(result1 == result2)
;
```

二十二、this 关键字

- 通过它可以访问方法，在方法内部，可以引用 this，其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。
- 适用场景：
 - 逻辑上是让成员 x 赋值给构造函数的参数 x，但是两个名字相同，无法区分，因此结构行数相当于什么都没做，使用 this 可以做到这一点。

```
class Entity
{
public:
    int x, y;
    Entity(int x ,int y)
    {
        x = x;
    }
};

class Entity
{
public:
    int x, y;
    Entity(int x ,int y)
    {
        //!Entity* e = this; this的本质
        this->x = x;
        //(*this).x = x; 也可以
        this->y = y;
    }
};
```

- 如果我们想要调用函数（类之外），此函数将 entity 作为参数。
- *this 表示对当前对象的引用。由于 this 本身是一个指向当前对象的指针，使用解引用操作符*（星号）可以获取指针所指向的对象的实例。这允许在类的成员函数中返回对象本身，而不是指针。

```
class Entity
{
public:
    int x, y;
    Entity(int x ,int y)
    {
        //!Entity* e = this; this的本质
        this->x = x;
        //( *this ).x = x; 也可以
        this->y = y;

        Entity &e = *this; //这行代码可有可无。

        PrintEntity(*this);
    }
};

void PrintEntity(const Entity& e)
{
    //Print
}
```

二十三、对象生存期（栈作用域生存期）

- 关于对象的生存期，内存以及对象是如何在栈上生存的，生存期对于基于栈的变量意味着什么。
- 栈可以被认为是一种数据结构，可以在上面堆叠一些东西（例如书堆），每次我们在 C++ 中进入一个作用域，是在 push 栈帧，不一定是非得将数据压进（push）一个栈帧。
- 作用域可以是任何东西，比如函数作用域、if 语句作用域、for 或 while 循环、空作用域（下图）、类作用域。

```
int main()
{
    {
        Entity e;
    }

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

- 基于栈的变量，在我们一出作用域就被释放了，被摧毁了；
 - 常见错误：
 - ◆ 这个数组是在我们栈上创建的，一旦离开作用域内存就会被清除，因此

我们要么在堆上创建 (new)，要么将这里的数据赋值给一个在栈作用域之外存在的变量。

```
int* CreateArray()
{
    int array[50];
    return array;
}

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

```
void CreateArray(int *array)
{
    //fill our array
}

int main()
{
    int array[50];
    CreateArray(array);
}
```

■ 应用场景：

- ◆ **作用域指针 (unique_ptr 关键字)：**基本上是一个类，它是一个指针的包装器，在构造时用堆分配指针，然后在析构时删除指针，因此我们可以自动化这个 new 和 delete。

```
class ScopedPtr
{
private:
    Entity *m_Ptr;
public:
    ScopedPtr(Entity* ptr)
        :m_Ptr(ptr)
    {
    }
    ~ScopedPtr()
    {
        delete m_Ptr;
    }
};
```

- ◆ 使用 ScopedPtr 来创建 (两种写法)

其基本原理就是 ScopedPtr 类是在栈上创建，而栈在超出作用域会自动销毁，在销毁时会调用析构函数来 delete；因此，即使我们使用 new 来在堆上创建，在超出作用域时也会被销毁。

```
ScopedPtr e(new Entity());
ScopedPtr e = new Entity(); //隐式转换
```

二十四、智能指针

- New 在堆上分配内存，delete 来删除释放内存，智能指针就是实现这一过程自动化的一种方式，当你掉调用 new 时，不需要调用 delete，在大多数使用的时候甚

至不用调用 new，智能指针本质上是一个原始指针的包装，当使用时，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。

- 在 C++ 中，智能指针本身通常是在栈上创建的，但它们管理的对象是在堆上分配的。例如，当你使用 `std::shared_ptr` 或 `std::weak_ptr` 时，智能指针的实例会在栈上创建，而它所管理的资源，如动态分配的内存，是在堆上创建的。

unique_ptr

- `unique_ptr` 是作用域指针，是超出作用域时，它会被销毁然后调用 `delete`。
- `unique_ptr` 不可被复制，如果复制了一个 `unique_ptr`，那么他们指向的内存会有两个指针，两个 `unique_ptr` 指向同一个内存块。如果一个被销毁，那么它会释放那段内存，第二个就会报错。
- 其构造函数有 `explicit` 关键字，因此不可隐式转换只可显示，推荐使用下面的方法。
 - `std::unique_ptr<Entity> entity(new Entity());` 这种方法直接使用 new 来创建一个 Entity 对象，并将其地址传递给 `unique_ptr`。这种方式存在潜在的问题，比如如果在 new 和 `unique_ptr` 构造函数之间发生异常，那么新分配的对象可能不会被删除，从而导致内存泄漏。
 - `std::unique_ptr<Entity> entity = std::make_unique<Entity>();` 这种方法使用 `std::make_unique` 函数来创建对象。`std::make_unique` 是在 C++14 中引入的，它提供了异常安全保证，因为它将对象的创建和 `unique_ptr` 的初始化封装在一个不可分割的操作中。如果在对象创建过程中发生异常，分配的内存会被自动释放，从而避免内存泄漏。

```
{  
    std::unique_ptr<Entity> entity(new Entity());  
    std::unique_ptr<Entity> entity = std::make_unique<Entity>();  
}
```

shared_ptr

- 共享指针，其实现方式取决于编译器和在编译器中使用的标准库，其工作方式是通过引用计数，可以追踪你的指针有多少个引用，一旦引用计数达到零，它就被删除了。例如，我创建了一个共享指针，创建了另一个 `shared_ptr` 来复制它，那么我的引用计数是 2，当第一个被销毁的时候，我的引用计数减少 1，当最后一个销毁掉，计数回到零，那么共享指针就会被销毁。
 - 当然也可以使用 new 来创建，但不推荐，因为 `shared_ptr` 需要分配另一块内存，叫做控制块，用来存储引用计数，如果首先创建一个 `new Entity`，将其传给 `shared_ptr` 构造函数，它必须分配两次（`new Entity` 分配，然后是 `shared_ptr` 的控制内存块），效率底下。

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

- 例子：
 - ◆ 在只有代码跳出外层空作用域时，e0 才会被销毁，计数到 0，Entity 才

会被销毁。

```
class Entity
{
public:
    Entity()
    {
        std::cout << "created Entity" << std::endl;
    }
    ~Entity()
    {
        std::cout << "Destroyed Entity" << std::endl;
    }
};

int main()
{
    {
        std::shared_ptr<Entity> e0;
        {
            std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
            e0 = sharedEntity;
        }
    }
    std::cin.get();
}
```

◆ 还有一个指针可以与 shared 一起使用，weak_ptr。

weak_ptr

- 弱指针：如同声明其他东西一样声明，可以把它赋值为 sharedEntity，这里所做的和之前复制 sharedEntity 所做的一样，但并不会增加引用计数，
 - 这样代码在跳出内层空作用域时就会被销毁。

```
int main()
{
    {
        std::weak_ptr<Entity> e0;
        {
            std::shared_ptr<Entity> sharedEntity = std::make_shared<Entity>();
            e0 = sharedEntity;
        }
    }
    std::cin.get();
}
```

二十五、复制与拷贝构造函数

- **浅拷贝**只复制对象的表层数据，如果对象包含指向其他数据的指针，浅拷贝不会复制指针指向的数据，而是只复制指针本身。这意味着原始对象和拷贝对象的指针将指向同一块内存地址。如果其中一个对象被销毁，它可能会影响到另一个对象，因为它们共享相同的内存资源。
- **深拷贝**则不同，它会复制对象所有的层级数据。深拷贝不仅复制对象本身，还复制对象所指向的数据。这样，即使一个对象被销毁，也不会影响另一个对象，因为它们各自拥有独立的数据副本。
- 浅拷贝复制的经典错误：以下程序会崩溃
 - 当你创建 second 对象并将 string 对象赋值给它时，m_Buffer 指针被复制，这意味着两个 String 对象都指向相同的内存。当 main 函数结束时，每个对象的析构函数都会尝试删除同一块内存，这会导致**双重删除**错误。
 - 因此用深拷贝就可以解决。

```
class String
{
private:
    char *m_Buffer;
    unsigned int m_Size;
public:
    String(const char* string)
    {
        m_Size = strlen(string); 未定义标识符 "strlen"
        m_Buffer = new char[m_Size+1]; //+1提供终止字符
        memcpy(m_Buffer, string, m_Size+1); //目标, 源头, 大小。 memcpy函数提供终止字符
    }

    ~String()
    {
        delete[] m_Buffer;
    }

    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;
}

int main()
{
    String string = "Lvjiale";
    String second = string;
    std::cout << string << std::endl;
    std::cout << second << std::endl;
    std::cin.get();
}
```

- **深拷贝**，拷贝构造函数：是一个构造函数，当你复制第二个字符串时，它会被引用。即把一个字符串复制给一个对象时（也是一个字符串），当你试图创建一个新的变量并对它分配另外一个变量时，这个变量和你正在创建的变量有相同的类型。
 - 普通构造函数 String(const char* string)接受一个 const char*类型的参数，用

于根据字符串字面量创建一个新的 String 对象。它分配内存并复制给定的字符串到新的内存位置。

- 拷贝构造函数 String(const String &other)接受一个对现有 String 对象的引用作为参数。它的目的是创建一个新的 String 对象，这个新对象是作为参数传递的对象的精确副本。拷贝构造函数也分配内存，但它复制的是另一个 String 对象的数据，而不是一个字符串字面量。
- 尽管两个构造函数看起来很相似，因为它们都进行内存分配和数据复制，但它们的用途完全不同。普通构造函数用于从字符串字面量创建对象，而拷贝构造函数用于从另一个 String 对象创建对象。这就是为什么需要两个不同的构造函数的原因。

```
class String
{
private:
    char *m_Buffer;
    unsigned int m_Size;
public:
    String(const char* string)
    {
        m_Size = strlen(string); 未定义标识符 "strlen"
        m_Buffer = new char[m_Size+1]; //+1提供终止字符
        memcpy(m_Buffer, string, m_Size+1); //目标, 源头, 大小。memcpy函数提供终止字符
    }
    String(const String &other) //深拷贝
    : m_Size(other.m_Size)
    {
        m_Buffer = new char[m_Size + 1]; //新创建副本
        memcpy(m_Buffer, other.m_Buffer, m_Size + 1);
    }
    ~String()
    {
        delete[] m_Buffer;
    }

    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;
}

int main()
{
    String string = "Lvjiale";
    String second = string; //隐式构造
    std::cout << string << std::endl;
    std::cout << second << std::endl;
}
```

二十六、箭头运算符

- 常用于指针的运算，实际相当于逆向引用了 Entity 指针。
 - 在大部分情况中会使用以下。

```

class Entity
{
public:
    void Print() const { std::cout << "Hello!" << std::endl; }
};

int main()
{
    Entity e;
    Entity *ptr = &e;
    ptr->Print();

    std::cin.get();
}

```

- 但也可以重载运算符使得代码变得简易。

```

class Entity
{
public:
    void Print() const { std::cout << "Hello!" << std::endl; }
};

class ScopedPtr
{
private:
    Entity *m_Obj;
public:
    ScopedPtr(Entity* entity)
        : m_Obj(entity)
    {
    }
    ~ScopedPtr()
    {
        delete m_Obj;
    }

    Entity* operator->()
    {
        return m_Obj;
    }
};

int main()
{
    ScopedPtr entity = new Entity();
    entity->Print();

    std::cin.get();
}

```

- 使用箭头操作符，获取内存中某个成员变量的偏移量。
- 这是通过将 nullptr（空指针）转换为 Vector3 类型的指针，然后取其 z 成员的地址来实现的。然后，将该地址转换为 int 类型以获取偏移量。当我们创建一个指向结构体或类的指针，并将其设置为 nullptr（即 0），然后取该指针的成员的地址，我们实际上得到的是成员的偏移量。这是因为结构体或类的起始地址在这种情况下被假定为 0，所以成员的地址就等于它的偏移量。

```

struct Vector3
{
    float x, y, z;
};

int main()
{
    int offset = (int)&((Vector3 *)nullptr)->z;

    std::cin.get();
}

```

```

struct Vertex
{
    float x, y, z;
    Vertex(float x, float y, float z)
        :x(x),y(y),z(z)
    {
    }
    Vertex(const Vertex& vertex)
        :x(vertex.x),y(vertex.y),z(vertex.z)
    {
        std::cout << "Copied" << std::endl;
    }
};

std::ostream& operator<<(std::ostream& stream, const Vertex& vertex)
{
    stream << vertex.x << "," << vertex.y << "," << vertex.z;
    return stream;
}

int main()
{
    std::vector<Vertex> vertices;
    vertices.push_back(Vertex(1, 2, 3));
    vertices.push_back(Vertex(4, 5, 6));
    vertices.push_back(Vertex(7, 8, 9));

    std::cin.get();
}

```

```

int main()
{
    std::vector<Vertex> vertices;
    vertices.reserve(3);
    vertices.emplace_back(1, 2, 3);
    vertices.emplace_back(4, 5, 6);
    vertices.emplace_back(7, 8, 9);

    std::cin.get();
}

```





二十八、库的使用

- 静态链接与动态链接。
- 静态链接意味着这个库会被放到你的可执行文件中，在 exe 文件之中或者其他操作系统下的可执行文件。
- 动态链接是在运行时被链接，所以仍然有一些链接，可以选择在程序运行的时候，装在动态链接库（loadlibrary 函数），会载入动态库，从中拉出函数，然后开始调用函数；也可以在应用程序启动时，加载 dll 文件（动态链接库）。
- 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的优化。
- #include <filename>: 使用尖括号<>通常用于包含系统头文件。编译器会在标准的系统库目录列表中搜索名为 filename 的文件。
- #include "filename": 使用双引号""通常用于包含你自己程序的头文件。编译器首先会在包含当前文件的目录中搜索名为 filename 的文件，如果没有找到，然后会在用于<>的相同系统目录中搜索。

- 头文件同时支持静态和动态（都选择导入头函数）。
- 同时存在一个，在库函数包含中，要么导入静态库，要么导入动态链接指针库。

静态链接

- 以二进制文件的形式进行链接，而不是获取实际依赖库的源代码并自己进行编译。
 - glfw3.lib 是静态链接
 - glfw3.dll 和 glfw3dll.lib 是动态链接，其中库中保存了指向动态链接的指针，包含了所有这些函数的位置，连接器可以直接链接到它们可以更快，但单独的 dll 也可以工作，通过函数名来访问 dll 中的函数。
 - glfw3_mt.lib 是 GLFW 库的一个版本，用于在 Windows 上开发 OpenGL 应用程序。这个特定的库文件是为了与**多线程**（Multi-threaded）运行时库一起使用的。当你在 Visual Studio 等 IDE 中设置项目属性时，如果你选择了**多线程或多线程调试**作为运行时库选项，那么你应该链接到 glfw3_mt.lib。

 glfw3.dll	2024/02/23 23:03	应用程序扩展	207 KB
 glfw3.lib	2024/02/23 23:02	Altium Library	550 KB
 glfw3_mt.lib	2024/02/23 23:03	Altium Library	550 KB
 glfw3dll.lib	2024/02/23 23:03	Altium Library	32 KB

- 包含头文件：
 - 项目右键→Properties→C/C++→general→additional include directories。指定附加的包含目录（可以用宏定义然后相对路径）。
- 连接到库：
 - 项目右键→Properties→Linker→input→Additional Dependencies(在这里可以只写库的名称，其真实路径我们可以另外设置)
 - 项目右键→Properties→Linker→input→general→Additional Library Directories（库函数）

```
#include <GLFW/glfw3.h>

int main()
{
    int a = glfwInit();
    std::cout << a << std::endl;

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

- 我们也可以不包含头文件只包含库文件，可以直接声明这个函数，但其实库是用 C 语言写的，因此我们需要声明时注意。


```
//#include <GLFW/glfw3.h>

extern "C" int glfwInit();

int main()
{
    int a = glfwInit();
    std::cout << a << std::endl;

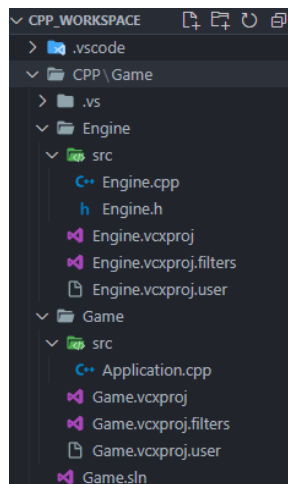
    std::cin.get();
}
```

动态链接

- 发生在运行时，并非可执行文件的一部分，当你启动一个普通的可执行文件时，它会被加载到内存中，然而如果有一共 dll，这意味着实际上链接了另一个库，一个外部的二进制文件，在运行时动态地链接，将一个额外的文件加载到内存中。
- 可选择完全动态的加载库，这样可执行文件与动态库就没有任何关系了。
- **包含头文件：**
 - 项目右键→Properties→C/C++→general→additional include derectories。指定附加的包含目录（可以用宏定义然后相对路径）。
- **连接到库：**
 - 项目右键→Properties→Linker→input→Additional Dependencies(在这里可以只写库的名称，其真实路径我们可以另外设置)
 - 项目右键→Properties→Linker→input→general→Additional Library Directories（dll 的指针库 glfw3dll.lib）
- **放置 dll 库：**
 - 最简单的做法就是把 dll 库放在 exe 同一目录下。

创建与使用库，建立多项目

- 文件结构如下



- 依情况设置其输出为 lib, lld, exe。
- 创建多项目，载入头函数的时候，可以使用相对路径。

```
# include "../..../Engine/src/Engine.h"
```

- 当然也可以添加头文件路径，然后直接载入。
 - 项目右键→Properties→C/C++→general→additional include derectories。指定附加的包含目录（可以用宏定义然后相对路径）。

Additional Include Directories **\$\$(solutiondir)Engine\src**

```
# include "Engine.h"

int main()
{
    engine::PrintMessage();
}
```

- 在对 Engine 生成 lib 的同时，我们不需要对其做上方的静态链接，VS 可以为我们自动化这个，因为这个项目是在实际的解决方案中。
 - ◆ 对生成 exe 的项目右键→add→reference 即可。
 - ◆ 除了没有了上方设置路径等操作，如果我们把 engine 名字改成 core 什么的，依旧可以运行。
 - ◆ 两者形成了依赖，因此构建 game 的时候，会先自动构建 engine。

二十九、如何处理多返回值（存疑）

- 假设有一个函数，这个函数需要返回两个字符串。

三十、模板（泛型）

- 允许你定义一共可以根据你的用途进行编译的模板，让编译器基于一套规则替你写代码。
- 模板并非一个实际的代码，不是一个真的函数，只有当我们实际调用它的时候，基于传递的参数，这些函数才被真的创建。因此在模板中出错且没有被调用的时候，编译器是不会报错的。
 - 例如当我写一个函数的时候，我在这个函数里面使用模板，实际上做的是创建一个蓝本，当我们调用的时候，指定特定的参数，这个参数决定了放入到模板中的实际代码。
 - ◆ 没有模板的写法

```
void Print(int value)
{
    std::cout<< value << std::endl;
}
void Print(std::string value)
{
    std::cout<< value << std::endl;
}
void Print(float value)
{
    std::cout<< value << std::endl;
}
```

◆ 模板写法

- 我们当然可以不加尖括弧，因为编译器会推断出这是什么类型。

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

int main()
{
    Print<int>(5);
    Print<float>(5.5f);
    Print<std::string>("Hello");
    std::cin.get();
}
```

- 当然我们可以不仅仅局限于传递数据类型。

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

int main()
{
    Array<5> array;
    std::cout << array.GetSize() << std::endl;
    std::cin.get();
}
```

- 当然也不一定是一个变量。

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

int main()
{
    Array<int,5> array;
```

三十一、栈与堆内存的比较（存疑）

- 当我们程序开始的时候，堆和栈被分成了一堆不同的内存区域。内存中还有许多东西但我们最关心的两个就是堆和栈，在应用程序启动后，操作系统要做的就是将整个程序加载到内存，并分配一大堆物理 ram，堆和栈就是 ram 中实际存在的两个区域。

栈

- 栈通常是一个预定义大小的内存区域，通常为 2 兆字节左右。

堆

- 堆也是一个预定义了默认值的区域，但是可以生长。

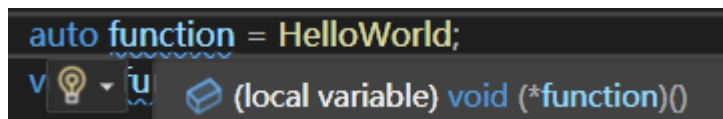
三十二、auto 关键字

- 其作用较为简单。注意事项有以下
- 在 C++ 中，auto 关键字的使用时机通常取决于以下几个因素：
 - 类型明显：当变量的类型很明显，或者从上下文中容易推断出来时，使用 auto 可以使代码更简洁。
 - 迭代器和泛型编程：在使用 STL 容器或泛型编程时，迭代器的类型可能非常复杂，使用 auto 可以避免冗长的类型声明。
 - 类型复杂或易变：当变量类型非常复杂或者可能会改变，而这种改变不应该影响到使用该变量的代码时，auto 可以提高代码的可维护性。
 - lambda 表达式：对于 lambda 表达式，由于其类型不能直接写出，auto 可以用来声明 lambda 表达式的变量。
 - auto 关键字对于函数指针之类的东西非常有用。
- 有一些情况应该避免使用 auto：
 - 类型不明显：如果使用 auto 后，代码阅读者不能清楚地知道变量的类型，这时应该避免使用 auto，以保持代码的可读性。
 - 过度使用：如果 auto 被过度使用，可能会导致代码难以理解和维护，特别是在团队协作的环境中。

三十三、函数指针

原始函数指针（C 语言）

- **函数指针**：是将一个函数赋值给一个变量的方法。
- auto 关键字对于函数指针之类的东西非常有用。
 - 在指向的时候，不用带()，因为指向的是函数的内存地址。



```
auto function = HelloWorld;
```

The image shows a code editor snippet with the line `auto function = HelloWorld;`. A tooltip is visible below the line, showing a lightbulb icon, a dropdown menu with 'u' selected, and the text `(local variable) void (*function)()`.

- 有参和无参。

```

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

int main()
{
    auto function = HelloWorld;
    // 等价
    void(*function)() = HelloWorld;

    HelloWorld();
    function();
}

```

```

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

int main()
{
    auto function = HelloWorld;
    // 等价
    void(*function)(int) = HelloWorld;

    HelloWorld(8);
    function(8);
}

```

- 函数指针做形参

```

void PrintValue(int value)
{
    std::cout << "Value:" << value << std::endl;
}
// 第二个参数为函数指针
void ForEach(const std::vector<int>& values, void(*Function)(int))
{
    for (int value : values)
    {
        PrintValue(value);
    }
}

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

    std::cin.get();
}

```

Lambda (C++语言)

- Lambda 本质上是我们定义一种叫做匿名函数的方式，用这种方式创建的函数，不需要实际创建一个函数，是一个快速运行的一次性函数，更想将其视为一个变量，而不是正式函数那样，在我们实际编译的代码中作为一个符号存在。
- 只要有一个函数指针，那么都可以在 C++ 中使用 lambda，在我们设置函数指针指向函数的任何地方，都可以将它设置为 lambda。

■ 语法

```
[ captures ] <tparams>(optional)(C++20) ( params ) lambda-specifiers { body } (1)
```

```
[ captures ] ( params ) trailing-return-type { body } (2)
```

```
[ captures ] ( params ) { body } (3)
```

```
[ captures ] lambda-specifiers(optional)(C++23) { body }
```

知乎 @冲神

- ◆ **captures 捕获列表**，lambda 可以把上下文变量以值或引用的方式捕获，在 body 中直接使用。

- [] 什么也不捕获，无法 lambda 函数体使用任何
- [=] 按值的方式捕获所有变量
- [&] 按引用的方式捕获所有变量
- [=, &a] 除了变量 a 之外，按值的方式捕获所有局部变量，变量 a 使用引用的方式来捕获。这里可以按引用捕获多个，例如 [=, &a, &b, &c]。这里注意，如果前面加了=，后面加的具体的参数必须以引用的方式来捕获，否则会报错。
- [&, a] 除了变量 a 之外，按引用的方式捕获所有局部变量，变量 a 使用值的方式来捕获。这里后面的参数也可以多个，例如 [&, a, b, c]。这里注意，如果前面加了&，后面加的具体的参数必须以值的方式来捕获。
- [a, &b] 以值的方式捕获 a，引用的方式捕获 b，也可以捕获多个。
- [this] 在成员函数中，也可以直接捕获 this 指针，其实在成员函数中，[=]和[&]也会捕获 this 指针。

- ◆ **tparams 模板参数列表**(c++20 引入)，让 lambda 可以像模板函数一样被调用。

- ◆ **params 参数列表**，有一点需要注意，在 c++14 之后允许使用 auto 左右参数类型。

- ◆ **lambda-specifiers lambda 说明符**，常用的参数就是 mutable 和 exception。其中，表达式(1)中没有 trailing-return-type，是因为包含在这一项里面的。

- 关于 mutable 关键字，当你在 Lambda 表达式中使用[=]捕获列表时，所有捕获的变量都是以值的形式被捕获的，这意味着 Lambda 表达式内部不能修改它们的值。如果你需要修改它们，就必须在 Lambda 表达式后添加 mutable 关键字。但是，即使添加了 mutable，Lambda 表达式内部修改的也只是捕获的副本，而不会影响原始变量 a 的值。如果要改

变，请使用[&]，使用引用捕获，就不需要 mutable 关键字，因为引用捕获默认允许你修改捕获的变量。

```
// 如果需要在body里面改变值，就要加mutable。
auto lambda = [=](int value) mutable
{ a = 5; std::cout <<"value:"<<value<<std::endl; };
```

- ◆ **trailing-return-type** 返回值类型，一般可以省略掉，由编译器来推导。
- ◆ **body** 函数体，函数的具体逻辑。
- 替换函数指针作为参数传递给函数。
 - Lambda 表达式返回类型一般是会自动推导，因此不写。

```
// 第二个参数为函数指针
void ForEach(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};
    int a = 5;

    // 此Lambda表达式接收一个int类型的值且名为value，在函数中有所体现，a在此处仅做mutable关键字的示例。
    class lambda [](int value) mutable->void
    auto lambda = [=](int value) mutable
    { a = 5; std::cout <<"value:"<<value<<std::endl; };

    ForEach(values, lambda);

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

- lambda 表达式显式的设置返回类型。

```
// 如果要显式的设置返回类型，则用->，如果非void则要在body中return。
auto lambda = [=](int value) mutable -> void
{ a = 5; std::cout <<"value:"<<value<<std::endl; };
```

- 关于 lambda 表达式形参格式。
 - **std::function** 是一个通用的函数封装器，它可以存储、调用和引用任何可调用的目标——函数、Lambda 表达式、绑定表达式或其他函数对象，只要它们具有相同的签名。这意味 std::function 允许你以统一的方式处理不同类型的可调用对象。
 - ForEach 函数中，第二个参数被定义为 const std::function<void(int)>& func，这样做有几个好处：
 - ◆ **灵活性**：你可以传递任何类型的可调用对象给 ForEach，只要它接受一个 int 参数并返回 void。这包括普通函数、Lambda 表达式、函数对象等。
 - ◆ **类型安全**：std::function 确保传递给 ForEach 的可调用对象符合预期的签名，这有助于避免类型错误。
 - ◆ **易用性**：使用 std::function 作为参数类型使得函数接口更清晰，调用者知道他们需要提供什么类型的函数。

```
void ForEach(const std::vector<int>& values, const std::function<void(int)>& func)
```


三十四、C++的命名空间

- 命名空间的主要目的是避免命名冲突。

```
namespace apple
{
    void print(const char* text)
    {
        std::cout << text << std::endl;
    }
}

namespace orange
{
    void print(const char* text)
    {
        std::string temp = text;
        std::cout << temp << std::endl;
    }
}
```

- 类也是一种命名空间，同样的道理也适用于静态函数或者类中的符号，方法，类等。
- 嵌套命名空间

```
namespace apple
{
    namespace function
    {
        void print(const char* text)
        {
            std::cout << text << std::endl;
        }
    }
}

namespace orange
{
    void print(const char* text)
    {
        std::string temp = text;
        std::cout << temp << std::endl;
    }
}

int main()
{
    using namespace apple::function;
    apple::function::print("aaa");
    std::cin.get();
}
```

三十五、线程

- `static bool s_Finished = false;` 定义了一个静态布尔变量 `s_Finished`，初始值为 `false`。这个变量用来控制线程何时停止工作。

- `void DoWork()` 函数是线程将要执行的工作。它使用了 `std::literals::chrono_literals`，这允许我们使用 `1s` 这样的字面量来表示时间。
- 在 `DoWork` 函数中，有一个 `while` 循环，它会一直执行，直到 `s_Finished` 变为 `true`。在循环体内，它打印 “Working...” 并且每秒暂停一次，这是通过 `std::this_thread::sleep_for(1s)` 实现的。
 - 在 `std::cout << "Working...\n"`；这行代码中，使用 `\n` 来换行而不是 `std::endl` 有以下好处：
 - **性能提升：**`\n` 只是简单地添加一个换行符，而 `std::endl` 会在添加换行符后强制刷新输出缓冲区。在不需要立即看到输出的情况下，避免频繁刷新缓冲区可以提高性能，特别是在大量输出或高频率循环中。
 - **缓冲控制：**如果你想要更细粒度的控制何时刷新输出，使用 `\n` 可以让你自己决定何时使用 `std::flush` 或其他方法来刷新缓冲区，而不是每次打印时都自动刷新。
 - 因此，在不需要立即反馈到控制台的情况下，使用 `\n` 可以提供更好的性能和更灵活的控制。这在处理大量数据或在性能敏感的应用程序中尤其有用。如果你需要确保每条消息都立即显示，那么使用 `std::endl` 是合适的选择。由于 `while` 循环可能会非常快速地连续运行，使用 `\n` 而不是 `std::endl` 可以减少性能开销。
- `int main()` 函数中，创建了一个 `std::thread` 对象 `worker`，并传递了 `DoWork` 函数作为线程要执行的任务。
- `std::cin.get()`；是一个阻塞调用，等待用户输入后才会继续执行。这允许 `DoWork` 在后台运行。
- 当用户按下任意键后，`s_Finished` 被设置为 `true`，这会导致 `DoWork` 中的 `while` 循环结束。
- `worker.join()`；是至关重要的一步，它会阻塞主线程直到 `worker` 线程完成它的工作。这确保了程序不会在 `worker` 线程还在运行时就退出。
- 最后，打印 “Finished” 并再次等待用户输入，以便看到程序输出结果。

```
static bool s_Finished = false;

void DoWork()
{
    using namespace std::literals::chrono_literals;

    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();
    std::cout << "Finished" << std::endl;
    std::cin.get();
}
```

三十六、计时

- 构造析构函数法

```
struct Timer
{
    std::chrono::time_point<std::chrono::steady_clock> start, end;
    std::chrono::duration<float> duartion;

    Timer()
    {
        start = std::chrono::high_resolution_clock::now(); 没有与这些操作数
    }
    ~Timer()
    {
        end = std::chrono::high_resolution_clock::now(); 没有与这些操作数
        duartion = end - start;

        float ms = duartion.count() * 1000.0f;
        std::cout << "Timer took" << ms << "ms" << std::endl;
    }
};

void Function()
{
    Timer timer;
    for (int i = 0; i < 100; i++)
    {
        std::cout << "Hello\n";
    }
}
```

- 普通方法

```
int main()
{
    Function();
    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;

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

三十七、多维数组

- 分配内存以及初始化。‘

```
int main()
{
    int *array = new int[50];
    int **a2d = new int *[50]; // 我们有一个包含五十个数组内存位置的数组，即二维数组。
    for (int i = 0; i < 50; i++)
    {
        a2d[i] = new int[50]; // a2d[i] 是 int 指针型，50*50 的二维数组。
        a2d[0][0] = 0;
        int ***a3d = new int **[50];
        for (int i = 0; i < 50; i++)
        {
            a3d[i] = new int *[50]; // 二维数组
            for (int j = 0; j < 50; j++)
            {
                a3d[i][j] = new int[50]; // 三维数组，解引用。
            }
        }
        a3d[0][0][0] = 0;

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

- 而后还要 delete 数组(以二维为例)
 - 如果直接执行 delete[] a2d，则直接会删除保存的 50 个含有 50 个数据的指针，造成内存泄露。

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

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

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

- 处理数组的数组，会造成内存碎片的问题，且一维数组比二维数组效率更高，因为二维数组相当于分配 x 个数组，每个数组里面有 y 个数据，而 x 个数组可能分配在 ram 的不同地方，cpu 读取会跳来跳去花费时间。

三十八、排序（std::sort）

排序与 lambda 表达式

- 在 std::sort 函数中使用 lambda 表达式时，返回 true 和 false 决定了元素排序的顺序。当排序函数返回 true 时，它表示第一个参数应该在第二个参数之前；当返回 false 时，表示第一个参数应该在第二个参数之后。
 - 所有的 1 都会被放在数组的末尾，而其他元素则按照从大到小的顺序排列。这是因为在比较过程中，当 1 作为比较元素出现时，它总是被推向数组的后端。
 - 如果 a 是 1，lambda 表达式返回 false，意味着 a 不应该在 b 之前，因此 1 会被放在数组的后面。
 - 如果 b 是 1，lambda 表达式返回 true，意味着 b 应该在 a 之前，因此 1 会被放在数组的后面。
 - 如果 a 和 b 都不是 1，lambda 表达式通过比较 $a > b$ 来返回 true 或 false，实现降序排序。

```
std::vector<int> values = {3, 5, 1, 4, 2};
std::sort(values.begin(), values.end(), [](int a, int b)
{
    if(a == 1)
        return false;
    if(b == 1)
        return true;
    return a > b;
});
```

三十九、类型双关

- 类型双关是花哨的术语，用来在 C++ 中绕过类型系统。
- 我们告诉编译器：“虽然 a 是一个 int，但请把它当作 double 来处理。”然后，我们尝试读取这个 int 值所在内存位置的 double 值。这是不安全的，因为 int 和 double 在内存中的表示方式和大小通常是不同的。int 通常是 4 个字节，而 double 通常是 8 个字节。这意味着，当我们尝试将 a 的内存当作 double 来读取时，我们可能会读取到相邻内存区域的数据，这可能会导致未定义的行为。

```
int main()
{
    int a = 50;
    double value = *(double *)&a;

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

- 当只是想访问而不想创建一个全新的变量，则可以使用引用。

```
double& value = *(double *)&a;
```

- 当我们尝试赋值的时候，这是非常危险的，它实际上要写的是 8 个字节而不是 4 个字节，但实际上我们只有四个字节的内存空间，这是非常危险的。

```
int main()
{
    int a = 50;
    double& value = *(double *)&a;

    value = 0.0;
    std::cin.get();
}
```

四十、联合体

- 联合体有点像类类型或者结构体类型，只不过它一次只能占用一个成员的内存，当想给同一个变量取两个不同的名字时，联合体就非常有用。
- 通常 union 是匿名使用的，但是匿名 union 不能含有成员函数。
- 联合体中有两个成员：一个 float 类型的 a 和一个 int 类型的 b。在 C++ 中，联合体的成员共享相同的内存位置，这意味着 a 和 b 占据相同的内存空间。
 - 由于 a 和 b 共享内存，这个操作实际上也改变了 b 所占据的内存内容。然后，代码打印出 u.a 和 u.b 的值。
 - 输出的结果将取决于 float 和 int 在内存中的表示方式，以及系统的字节序（大端或小端）。因此 float 值 2.0f 在内存中的表示将被解释为 int 类型的某个值。这个值不会是 2，因为 float 的二进制表示与 int 的不同。
 - 为了理解输出，我们需要知道 2.0f 作为 float 类型在内存中的二进制表示。当这个二进制序列被解释为 int 类型时，它将对对应于一个完全不同的整数值。
 - 输出 2.0f 对应的 float 值，以及由于类型双关（Type Punning）产生的 int 值。这个 int 值将是 float 值 2.0f 的内存表示被解释为 int 类型时的结果。

```
int main()
{
    struct Union
    {
        union
        {
            float a;
            int b;
        };
    };
    Union u;
    u.a = 2.0f;
    std::cout << u.a << "," << u.b << std::endl;
    std::cin.get();
}
```

四十一、类型转换

- 风格类型转换主要有四种：`static_cast`、`reinterpret_cast`、`dynamic_cast`、`const_cast`。
 - **`static_cast`**：这是最常用的类型转换运算符，用于执行编译时的类型安全转换。它可以用于基本数据类型之间的转换，如 `int` 转换为 `float`，或者在继承层次结构中的向上转换（从派生类到基类）。

```
int i = 10;
float f = static_cast<float>(i); // 安全转换int为float
```

- **`dynamic_cast`**：主要用于处理多态性，即在继承层次结构中的向下转换（从基类到派生类）。它在运行时进行类型检查，如果转换不合法，则返回空指针或抛出异常。

```
class Base { virtual void func() {} };
class Derived : public Base {};
Base* basePtr = new Derived();
Derived* derivedPtr = dynamic_cast<Derived*>(basePtr); // 向下转换
```

- **`const_cast`**：用于移除或添加对象的 `const` 属性。这是唯一能够改变对象 `const` 属性的 C++ 转换运算符。

```
const int ci = 10;
int* modifiable = const_cast<int*>(&ci); // 移除const属性
```

- **`reinterpret_cast`**：用于进行低级别的重新解释转换，例如将指针类型转换为足够大的整数类型，或者将一种类型的指针转换为另一种类型的指针。这种转换不安全，可能会导致未定义的行为。即不想转换任何东西，只是想把指针解释成别的东西，即把现有的地址解释为另一种类型。

```
int* iptr;
char* cptr = reinterpret_cast<char*>(iptr); // 重新解释int指针为char指针
```

四十二、多数据处理

Pairs 和 tuple

- **Pair 的用法和语法**：`pair` 是一个结构体，可以存储两个元素。它通常用于将两个相关联的数据（例如键和值）组合在一起。`pair` 的声明和使用如下：

```
int main() {
    // 创建一个pair对象
    std::pair<std::string, int> p1("Alice", 42);

    // 访问pair的元素
    std::cout << "Name: " << p1.first << ", Age: " << p1.second << std::endl;

    // 使用make_pair创建pair
    auto p2 = std::make_pair("Bob", 25);

    // 使用tie解包pair
    std::string name;
    int age;
    std::tie(name, age) = p2;
    std::cout << "Name: " << name << ", Age: " << age << std::endl;

    return 0;
}
```

- **Tuple 的用法和语法:** tuple 是一个固定大小的容器，可以包含多个（两个以上）不同类型的元素。tuple 的声明和使用如下：

```
int main() {
    // 创建一个tuple对象
    std::tuple<std::string, int, char> t1("Carol", 30, 'C');

    // 访问tuple的元素
    std::cout << "Name: " << std::get<0>(t1) << ", Age: " << std::get<1>(t1) << ", Initial: " << std::get<2>(t1) << std::endl;

    // 使用make_tuple创建tuple
    auto t2 = std::make_tuple("Dave", 45, 'D');

    // 使用tie解包tuple
    std::string name;
    int age;
    char initial;
    std::tie(name, age, initial) = t2;
    std::cout << "Name: " << name << ", Age: " << age << ", Initial: " << initial << std::endl;

    return 0;
}
```

结构化绑定

```
std::tuple<std::string, int> CreatePerson()
{
    return {"LVJIALE", 22};
}

int main()
{
    auto [name, age] = CreatePerson();

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


四十三、C++17 新增模板

Std::optional

- 在 C++ 中，std::optional 是一个模板类，用于表示可能存在也可能不存在的值。它是 C++17 标准库的一部分，非常适合用于那些可能失败并返回非错误但不确定值的函数。
- 在这个例子中，createNickname 函数尝试为给定的名字创建一个昵称。如果名字是 "Alex"，它会返回一个包含 "Lex" 的 std::optional<std::string>。否则，它会返回一个空的 std::optional，表示没有昵称。在 main 函数中，我们检查 nickname 是否有值，如果有，我们就解引用它并打印出来。如果没有，我们打印一条消息说没有提供昵称。这样的处理方式使得代码更加清晰和安全。

```
std::optional<std::string> createNickname(const std::string& name) {  
    if (name == "Alex") {  
        return "Lex"; // 返回一个包含昵称的optional  
    }  
    return std::nullopt; // 返回一个空的optional，等同于return {};  
}  
  
int main() {  
    auto nickname = createNickname("Alex");  
    if (nickname) {  
        std::cout << "Nickname: " << *nickname << std::endl;  
    } else {  
        std::cout << "No nickname provided." << std::endl;  
    }  
}
```

Std::variant

- std::variant 可以存储一组预定义类型中的任何一个类型的值。它类似于联合（union），但提供了类型安全，因为它总是知道它当前存储的是哪种类型的值。
- std::variant 的作用包括：
 - 提供一种类型安全的方式来存储多种不同类型的值。
 - 使得函数可以返回多种类型中的一种，而不是仅限于单一类型。
 - 在需要存储多种可能类型的值时，提供比联合更安全和灵活的选择。
- 在这个例子中，myVariant 可以存储 int、double 或 std::string 类型的值。我们使用 std::get 来获取存储的值，并使用 std::visit 来访问当前存储的值，无论它是哪种类型。这展示了 std::variant 的灵活性和类型安全性。

```

int main() {
    std::variant<int, double, std::string> myVariant;    命名空间 "std" 没有成员

    // 存储int类型
    myVariant = 10;
    std::cout << "int: " << std::get<int>(myVariant) << std::endl;    没有与参数

    // 存储double类型
    myVariant = 3.14;
    std::cout << "double: " << std::get<double>(myVariant) << std::endl;    没

    // 存储string类型
    myVariant = "Hello, Variant!";
    std::cout << "string: " << std::get<std::string>(myVariant) << std::endl;

    // 访问当前存储的值
    std::visit( {    命名空间 "std" 没有成员 "visit"
        std::cout << arg << std::endl;    未定义标识符 "arg"
    }, myVariant);    应输入声明

    return 0;    应输入声明
}    应输入声明

```

Std::any

- std::any 是 C++17 标准库中引入的一个类型，它提供了一种类型安全的方式来存储和操作任意类型的值。它的作用是允许一个变量持有任何类型的值，同时在需要的时候能够安全地提取出原始类型的值。
- std::any 的基本语法如下：

```

#include <any>

// 创建一个std::any对象
std::any a = 1; // 可以存储int
a = 3.14; // 可以存储double
a = true; // 可以存储bool

```

- 要从 std::any 对象中提取存储的值，你可以使用 std::any_cast。

```

// 提取值
try {
    int myInt = std::any_cast<int>(a);
    // 使用myInt
} catch (const std::bad_any_cast& e) {
    // 处理错误
}

```