1. 教程(点击的蓝色字体会跳转链接)

- chatgpt镜像
  学习的初步其实都是模仿，一开始都是不太懂的，所以可以让chatgpt给出大家示例代码来学习使用，需要什么就问什么，代码有bug也可以先问它，让他给出修改意见。
- CMAKE-官方教程
  这个需要在linux环境下，可以搜索资料`Cmake`的作用，后面应该会教大家安装`WSL2-linux-ubuntu22.04`
- 菜鸟教程
  速成可以先过一遍再说细节，有个大体的了解。
- 黑马程序员STL-P185后
  很长但很全面一般来说看不完的，里面的阶段编程大案例可以先不看，主要是学习核心语法.建议大家可以倍速跟着写一遍程序。表层语法有，深层讲解也有，所以时间很长，不可能一遍全部学会的。第一遍核心先学习表面的语法，所以里面有的课程实在难懂不知道在干什么就先听个基本概念再说，毕竟实际编程时候也不是所有高级用法都要用到的，简单高效代码也是很好的。我把教程分成了四部分，可以结合实际情况食用。
  - **[P1-P83]** 编程基础
    - `C`语言内容(`变量`、`数组`、`函数`和`指针`等等)
    - 没学过C语言的就从`P1`开始
  - **[P84-P142] 核心语法**
    - CPP语言相对C语言的扩充(引用、函数重载、函数默认参数、类封装,继承,多态等等)
    - 学过C语言的可以从`P84`开始看起
  - **[P167-P184] 模板**
    - 泛式编程，很有用，主要是要看懂别人的代码
    - 学过`Cpp`的就从P167开始看
  - **[P185-P263]**
    - `STL`讲解的非常详细，主要是会调用，接口都很好用，`std::string`就是`STL`的一部分，用过的都说好，相当于数据结构这门课程的实际代码实现。
    - 重点，其他的时间不够可以先搁置
      - `vector`,`deque`,`stack`
      - `string`
      - `map`
      - `algorithm`
        - 排序
        - 查找

下面是我提取出的比赛常用到的重点知识。

2. 数据类型和namespace

2.1. auto和for

`auto`非常有用，到后续会用到`stl`模板类,嵌套嵌套，类型名称会特别长且难记忆.
主要是可以偷懒，不用自己写数据类型，而让编译器自己推理，但使用的前提是编译器需要能自己推理出来。

```
int arr[10] = {0};
for (int i = 0; i<10; i++){
    arr[i] = i;
}
for (auto var:arr) {
    std::cout<< "arr:" <<var <<" ";
}
```

2.2. 自定义结构体和ns

`namespace`命名空间分割限制域和`::`界限访问符
`using namespace std;`就是最典型的`namespace`用法

```
// 数据类型
namespace temp_ns{
    int val =5;
}
namespace rm_autoaim{
    int val = 10;
    enum type {BIG, SMALL};

    //先定义Armor结构体以及讲解Cpp和C的区别
    struct Armor{
        int x{0}; //可以默认初始化为0
        int y{0}; //同上
        int z{0};
        Armor(int xin, int yin, int zin){
            x = xin;
            y = yin;
            z = zin;
        }
        int addAll() const{ // 结构内部函数
            return x+y+z;
        }
    };
}
//main-------------------------
//以下放在main函数内
//结构体
int val = 30;
rm_autoaim::Armor armor;
std::cout << armor.addAll() << std::endl;
//三个 namespace val
std::cout << "three val:" <<std::endl;
std::cout << val << std::endl;
std::cout << temp_ns::val << std::endl;
std::cout << rm_autoaim::val << std::endl;
```

3. 指针和引用

3.1. 指针->

这几个区别就看自己理解了，这里只是为了全面，理解清楚指针本身的应用即可。**常量指针和指针常量可以先搁着，后续再看**

```
//指针
//指针基础用法
int a = 10;
int* b = &a;
std::cout << "*b:" << *b << std::endl;
std::cout << "b:" << b << " &a:" << &a << std::endl;

Armor armor;
Armor* ptr = &armor;
//必须带(*ptr),因为.的优先级高于*
std::cout << "x:" << (*ptr).x << " y:" << (*ptr).y << " z:" << (*ptr).z << std::endl;
std::cout << "x:" << ptr->x << " y:" << ptr->y << " z:" << ptr->z << std::endl;

// 常量指针和指针常量
int a1 = 1, a2 = 2;
int* common_ptr = &a1;       // 普通指针
const int* const_ptr = &a1; // 常量指针(指向常量的指针,不能通过指针修改指向的变量)
int* const ptr_const = &a1; // 指针常量(指针是常量,不能修改指针的指向)

//1.修改 a1
a1 = 11;
std::cout << "change a1= 11 \t\t  " << "*common_ptr:" <<*common_ptr << "  *const_ptr:" <<
*const_ptr << "  *ptr_const:" << *ptr_const << std::endl;

//2.修改 *common_ptr来修改其指向的a1变量
*common_ptr = 111;
std::cout << "change *common_ptr = 111 " << " *common_ptr:" <<*common_ptr << " *const_ptr:"
<< *const_ptr << " *ptr_const:" << *ptr_const << std::endl;

//3.修改 *const_ptr来修改其指向的a1变量
// *const_ptr = 1111; //报错 不能这样修改指向的变量
const_ptr = &a2; // 但是可以修改他指向的地址
std::cout << "chage const_ptr-> \t  *const_ptr:" << *const_ptr << std::endl;

//4.修改 *ptr_const来修改其指向的a1变量
*ptr_const = 22;
std::cout << "change *ptr_const = 22   " << " *common_ptr:" <<*common_ptr << " *const_ptr:"
<< *const_ptr << " *ptr_const:" << *ptr_const << std::endl;
// ptr_const = &a2; //报错，不能修改其指向的地址

//5.const int* const ptr_dc = &a1;
int a1 = 1;
const int* const ptr_dc = &a1;//缝合怪，兼具两个特征
std::cout << "*ptr_dc" << *ptr_dc << std::endl; //唯一绑定，只可读
```

3.2. 引用&

引用这里其实核心是需要了解到引用作为函数参数的应用以及与指针作为函数参数的相同处和不同处,知道以下例子即可。

```
void changeNumToOne(float &num){
    num = 1; //传进来的参数也会被修改
}
void printNum(const float &num){
    //num = 9931;
    // ↑ 禁止，因为是const float&num 常引用，防止篡改
    std::cout << num << std::endl;
}
float num_out = 10;
changeNumToOne(num_out);
printNum(num_out);
```

3.2.1. 基础用法

```
int a = 10;
int &b = a;          // b <----> a  别称，绑定，改变一个另一个也会变
const int&c = a;     // 引用不可赋值，可以给一个人起不同的别名
//int &b = temp;     //不可以给两个人起同样的别名

// 三者的地址一样
std::cout << "&a:" << &a << " &b:" << &b << " &c:" << &c << std::endl;
//改变a，一起变
a = 5;
std::cout << "a:" << a << " b:" << b << " c:" << c << std::endl;
//改变b，一起变
b = 1;
std::cout << "a:" << a << " b:" << b << " c:" << c << std::endl;

/*    //改变c    //不可通过改变c 来改变变量名，因为是常引用
c = 1;
*/
```

3.2.2. 引用的本质 -- 指针常量

特征

- 必须初始化
- 只能赋值。初始化之后，便和变量一一绑定，例如int& ref = a1; //ref只是a1的的别称，不能将其改成其他变量的别称，ref = a2，这里其实只是将a2的值赋给ref(a1也会变)，仅仅是赋值，不会改变朝向，可自实践一下，和指针常量是不是很像，不能改变其指向
- const int& b

```
int a1 = 1,a2 = 2;
int* const ptr = &a1;
int& ref = a1;
std::cout << "address: " << ptr <<" " << &ref << std::endl;
```

3.2.3. 引用的应用

- 作为函数参数，后续讲到函数时应用

4. 函数

4.1. 重载和默认参数

```
// 原函数
int addNumbers(int num1, int num2){
    std::cout<<"call two int"<< std::endl;
    return num1 + num2;
}
// 重载函数-参数数量不同
int addNumbers(int num1, int num2, int num3){
    std::cout<<"call three int"<< std::endl;
    return  num1 + num2 + num3;
}

// 重载函数-参数类型不同
double addNumbers(double num1, double num2){
    std::cout<<"call two double"<< std::endl;
    return num1 + num2;
}

// 重载函数-默认参数
int addNumbers(int num1 = 1){
    std::cout<<"call one int"<< std::endl;
    return num1;
}
//main--------------------------------
//可以根据提示看看下面几种调用方式实际调用对应上面的哪一个
addNumbers(1,1);
addNumbers(1,1,1);
addNumbers(1.1,1.1);
addNumbers();
addNumbers(10);
```

4.2. 模板

template

```
template<typename T>
T addTwoNums(T num1, T num2){
```

```
    std::cout<< "T"<<std::endl;
    return num1 + num2;
}

template<typename T1, typename T2>
T2 addTwoNums(T1 num1, T2 num2){
    std::cout<< "T1 T2"<<std::endl;
    return num1 + num2;
}
//main----------------------------
//显式调用
std::cout<< addTwoNums<int,double>(1,9.1) <<std::endl;
//隐式调用-编译器自己找最合适的
std::cout<< addTwoNums(1,9.1) <<std::endl;
```

4.3. 模板结构体和函数参数

```
namespace rm_auto_aim{
    template<typename T>
    struct Armor{
        T x;
        T y;
        T z;
        Armor(T in_x, T in_y, T in_z){ x = in_x; y = in_y; z = in_z;};
    };
}

//模板函数-普通作为参数
template <typename T>
T calcDist(rm_auto_aim::Armor<T> in){
    std::cout << "com %in:" << &in <<std::endl;
    return sqrt(pow(in.x,2) + pow(in.y,2) + pow(in.z,2));
}

//模板函数-指针作为参数
template <typename T>
T calcDist(const rm_auto_aim::Armor<T>* in){
    std::cout << "ptr %in:" << in <<std::endl;
    return sqrt(pow(in->x,2) + pow(in->y,2) + pow(in->z,2));
}

//模板函数-引用作为参数
template <typename T>
T calcDist(rm_auto_aim::Armor<T>& in,int unuse){
    std::cout << "ref %in:" << &in <<std::endl;
    return sqrt(pow(in.x,2) + pow(in.y,2));
}
//main--------------------------------
rm_auto_aim::Armor<double> armor(3.0,4.0,10.0);
std::cout << "addr:   " << &armor <<std::endl;
std::cout << calcDist(armor) <<std::endl;
std::cout << calcDist(&armor) <<std::endl;
```

```
std::cout << calcDist(armor,0) <<std::endl;
```

用&和*的一个好处就是不用复制一份原变量的副本，从而提升其函数运行效率，但是也带来了另一个问题，就是可能在函数内会对原样本数据进行修改，然后破坏了原数据样本的安全性。所以再引入const。如下图所示(不写函数体了)，可以结合上面的指针和引用再理解一下为什么要用`(const ...)*or*(const ...& in) `*:

```
//指针参数
template <typename T>
T calcDist(const rm_auto_aim::Armor<T>* in){
    // in->x = 1; //修改他 编译器会报错
    std::cout << "ptr %in:" << in <<std::endl;
    return sqrt(pow(in->x,2) + pow(in->y,2));
}

//引用参数
template <typename T>
T calcDist(const rm_auto_aim::Armor<T>& in,int unuse){
    // in.x = 44; 同上
    std::cout << "ref %in:" << &in <<std::endl;
    return sqrt(pow(in.x,2) + pow(in.y,2));
}
```

5. class

5.1. 封装

triangle.hpp

```
#ifndef TEST1_TRIANGLE_HPP //
#define TEST1_TRIANGLE_HPP
namespace my_ns{
    class Triangle{
    public:// public function
        //构造函数
        Triangle(float in_a, float in_b, float in_c);
        //拷贝构造
        Triangle(const Triangle& tri);
        //类内重载=运算符构造，实现类变量=赋值
        Triangle& operator=(const Triangle& tri){
            if(&tri != this){ // 取地址 和 避免自赋值
                this->a_ = tri.a_;
                this->b_ = tri.b_;
                this->c_ = tri.c_;
                this->is_triangle_ = tri.is_triangle_;
                return *this;
            } else {
                return *this;
            }
```

```
        }
        //普通函数
        float slovePerimeter();
        float sloveArea();
    private:// 函数private封装
        //isTriangle
        bool isTriangle();
    private: // 数据private封装
        bool is_triangle_;
        float a_;
        float b_;
        float c_;
    };
}
#endif //TEST1_TRIANGLE_HPP
```

triangle.cpp

```
#include "triangle.hpp"
#include <cmath>
//构造函数
my_ns::Triangle::Triangle(float in_a, float in_b, float in_c) {
    a_ = in_a;
    b_ = in_b;
    c_ = in_c;
    is_triangle_ = isTriangle();
}
//拷贝构造
my_ns::Triangle::Triangle(const my_ns::Triangle &tri) {
    this->a_ = tri.a_;
    this->b_ = tri.b_;
    this->c_ = tri.c_;
    this->is_triangle_ = tri.is_triangle_;
}
bool my_ns::Triangle::isTriangle() {
    if(a_+b_>c_ && a_+c_>b_ && b_+c_>a_){
        return true;
    } else {
        return false;
    }
}
// 求面积
float my_ns::Triangle::sloveArea() {
    if(is_triangle_){
        //海伦公式
        float p = slovePerimeter()/2;
        return std::sqrt(p*(p-a_)*(p-b_)*(p-c_));
    } else {
        return 0;
    }
}
/**
 * 求周长
```

```
 */
float my_ns::Triangle::slovePerimeter() {
    if(is_triangle_){
        return a_+b_+c_;
    } else {
        return 0;
    }
}
```

5.2. 继承

因为等边三角形也需要计算面积和周长，因为`my_ns::Triangle`已经写了，我们没必要再重新写，等边三角形是三角形的特例，所以继承`my_ns::Triangle`类，就可以调用里面的方法从而方便得多

```
//继承
#include "triangle.hpp"
class EquTriangle : public my_ns::Triangle {
public://function
    //construction 构造EquTriangle时需要对Triangle也初始化
    EquTriangle(float len) : Triangle(len, len, len), len_(len) {}

    float getLen() const {
        return len_;
    }

    void setLen(float len) {
        EquTriangle::len_ = len;
    }

private://membership
    float len_;
};
//main-------------------------------------------------
//triangle实例化以及应用
my_ns::Triangle triangle(3, 4, 5);
std::cout << triangle.sloveArea() << std::endl;
std::cout << triangle.slovePerimeter() << std::endl;

//EquTriangle实例化以及应用
EquTriangle equ_triangle(3);
std::cout << equ_triangle.sloveArea() << std::endl;
std::cout << equ_triangle.slovePerimeter() << std::endl;

//复制构造函数
//重载运算符'='构造
my_ns::Triangle tri1(3,4,5);
my_ns::Triangle tri2(tri1);  //拷贝构造函数
my_ns::Triangle tri3 = tri1; //重载运算符'='进行构造
std::cout << tri2.sloveArea() << std::endl;
std::cout << tri3.sloveArea() << std::endl;
```

5.3. 多态

```
class Node{
    virtual void loop(){
        std::cout << "default Node loop" << std::endl;
    };
};

class Node1 : public Node{
    void loop() override{
        std::cout << "Node1 loop" << std::endl;
    }
};
class Node2 : public Node{
    void loop() override{
        std::cout << "Node1 loop" << std::endl;
    }
};
//---main
Node* node_ptr;
Node1 node1;
Node2 node2;

node_ptr = &node1;
node_ptr->loop(); //调用Node1的loop函数

node_ptr = &node2;
node_ptr->loop(); //调用Node2的loop函数
```

5.4. 抽象接口类

略

6. stl

黑马程序员STL-P185后
主要会用以下几种就可以,理解会用就行，不用太过深入.

- `vector`,`deque`,`stack`
- `string`
- `map`
- `algorithm`
    - 排序
    - 查找

6.1. vector和map最简单示例

```cpp
#include <iostream>
#include <vector>
#include <map>

namespace stl{
    struct Point{
        float x{0},y{0},z{0};////初始化变量为0
    };
}
int main() {
    //point1-3
    stl::Point point1{1,1,1};
    stl::Point point2{2,2,2};
    stl::Point point3{3,3,3};

    //vector 动态数组
    std::vector<stl::Point> pvec;
    pvec.push_back(point1); //数组尾部追加point1
    pvec.push_back(point2);
    pvec.push_back(point3);
    ////数组一样访问
    std::cout << pvec[2].z << std::endl;


    //map key-value 键值对
    std::map<std::string,stl::Point> pmap;
    //// 写
    pmap["point1"] = point1;
    pmap["point2"] = point2;
    pmap["point3"] = point3;

    //// 读
    std::cout << pmap["point2"].y << std::endl;

    return 0;
}
```

6.2. algorithm

```cpp
#include <iostream>
#include <algorithm>
//main----------------
int main(){
    int arr[10] = {0,8,3,5,2,7,4,1,9,6};
    std::sort(arr,arr+10);
    for(auto val : arr) {
        std::cout << val << " ";
    }
}
```

7. smart pointer

7.1. new & delete

```
//new & delete
auto* ptr_common = new my_ns::Triangle(6,8,10);
std::cout << ptr_common->sloveArea() << std::endl;
delete ptr_common;
// clion会提示ptr_common 指向位置地址
std::cout << ptr_common->sloveArea() << std::endl;
```

7.2. 智能指针

这里就首先学会最主要的`std::shared_ptr`的使用方法即可，先关注下面的程序，后面的详细的示例和区别可以先不管。就把它当作指针使用即可。

```
#include <iostream>
#include <memory>
namespace stl{
    struct Point{
        float x{0},y{0},z{0};////初始化变量为0
    };
}
int main() {
    std::shared_ptr<stl::Point> pptr;
    //// 一下会报错，因为没有分配内存，pptr仅仅是一个指针，没有分配空间，所以赋值的时候出错
    /*pptr->x = 0;
    std::cout << pptr->x << std::endl;*/

    //// 如下需要调用make_shared函数进行分配空间
    stl::Point point1 = stl::Point{1,2,3};
    pptr = std::make_shared<stl::Point>(point1);
    std::cout << pptr->x << std::endl;

    //// 不需要delete 也不需要new,所以这就是stl库的好用之处
    return 0;
}
```

7.2.1. shared_ptr

- 共享：多个shared_ptr 可指向同一个对象 `shared_ptr : obj = n : 1`

```
#include <memory>
#include "triangle.hpp"
//main----------------
//shared_ptr
std::shared_ptr<my_ns::Triangle> ptr_shared1,ptr_shared2,ptr_shared3;
ptr_shared1 = std::make_shared<my_ns::Triangle>(3,4,5);
ptr_shared2 = ptr_shared1; // 重载赋值'='符号
ptr_shared3 = ptr_shared1;
// ptr_shared1 = ptr_shared3; // 循环赋值,然后引入weak_ptr
std::cout << "triangle sloveArea: " << ptr_shared1->sloveArea() << " " << ptr_shared2-
>sloveArea() << " " << ptr_shared3->sloveArea() << std::endl;
std::cout << "shared_ptr   count: " << ptr_shared1.use_count() << " " <<
ptr_shared2.use_count() << " " << ptr_shared3.use_count() << std::endl;
```

7.2.2. unique_ptr

- 独占 - 一个对象智能由一个unique_ptr指向. `unique_ptr : obj = 1 : 1`

```
//main----------------
std::unique_ptr<my_ns::Triangle> ptr_unique1,ptr_unique2;
// ptr_unique1 = std::make_unique<my_ns::Triangle>(3,4,5);  //make_unique C++14的新特性
ptr_unique1 = std::unique_ptr<my_ns::Triangle>(new my_ns::Triangle(3,4,5));
// ptr_unique2 = ptr_unique1; // 赋值构造 X
// ptr_unique2 = std::unique_ptr<my_ns::Triangle>(ptr_unique1);    // 赋值构造 X
// std::cout << "shared_ptr   count: " << ptr_unique1.use_count(); // use_count X

// 但是可以move
ptr_unique2 = std::move(ptr_unique1);
std::cout << "ptr_unique2 sloveArea: " << ptr_unique2->sloveArea() << std::endl; // ok，因为
是将ptr1移动到了ptr2
std::cout << "ptr_unique1 sloveArea: " << ptr_unique1->sloveArea() << std::endl; // exit,
因为ptr1已经移动本身对象到ptr2了，自己已经不指向对象了，也就是我们说的独占，一个对象只能有一个unique_ptr来
使用
```

7.2.3. weak_ptr

- 可以先不管

```
std::shared_ptr<my_ns::Triangle> ptr_shared1;
ptr_shared1 = std::make_shared<my_ns::Triangle>(3,4,5);
std::weak_ptr<my_ns::Triangle> ptr_weak1,ptr_weak2,ptr_weak3;
ptr_weak1 = std::weak_ptr<my_ns::Triangle>(ptr_shared1);
ptr_weak2 = ptr_weak1;
ptr_weak3 = ptr_weak2;
ptr_weak3 = ptr_weak1;
std::cout<< ptr_weak3.use_count() << std::endl;
```

8. callback

这边最重要的思想就是要记住函数也可以作为函数的参数传入，具体实现都可以向`chatgpt`提需求让他帮我们写。

8.1. 同步和异步与回调函数

- **回调函数**
  可以作为函数参数传入另一个函数中的函数叫做回调函数
- **同步和异步**

8.2. 函数指针

以下主要是加深对函数和指针的理解，这个主要是写库的时候为了抽象和通用性才会用到的操作，本身自己编程可以不用，但是需要看懂别人写的代码，函数名本质上也是一个存储的一个具体操作的实现地址，而**函数的返回类型和参数个数以及类型的集合也就可以看作函数的类型**，例如下面提到的`int (*ptr_fun)(int,int);`，所以也可以抽象出函数类型的指针来指向函数。

```
// 函数指针 两元函数指针
int (*ptr_fun)(int,int);

// 两元函数
int add(int a, int b){
    return a+b;
}
int minus(int a, int b){
    return a-b;
}

// 抽象出一个两参数的操作操作
int twoNumOper(int num1, int num2, int (*op)(int,int)){
    return (*op)(num1,num2);
}
//main----------------------------
ptr_fun = add;
std::cout << ptr_fun(1,1) << std::endl;

ptr_fun = minus;
std::cout << ptr_fun(1,1) << std::endl;

std::cout << twoNumOper(1,1,add) << std::endl;
std::cout << twoNumOper(1,1,minus) << std::endl;
```

此外还可以更通用一些，利用**模板**和**可变参数(参考printf)**，例如以下例子，这个主要是`chatgpt`写的，我只是提出了需求。比较难理解，初学者不用太在意，直到有这个东西就可以。

```
#include <iostream>
#include <memory>
namespace stl {
    struct Point {
        float x{0}, y{0}, z{0}; // 初始化变量为0
    };
}

template<typename Func, typename... Args>
```

```cpp
void callAnyFunction(Func func, Args&&... args) {//// 这里有个右引用，cpp的用法
    func(std::forward<Args>(args)...); //// std::forward 完美转发
}

void movePointPosition(stl::Point& point, float dx, float dy, float dz) {
    point.x += dx;
    point.y += dy;
    point.z += dz;
}

int main() {
    stl::Point point1 = stl::Point{1, 2, 3};
    callAnyFunction(movePointPosition, point1, 3, 3, 3);
    std::cout << point1.z << std::endl;
    return 0;
}
```

8.3. 函数对象(仿函数)

重载`class`类的`()`符号,从而在使用时可以当函数使用，但是本质上是一个`class`

```cpp
#include <iostream>

// Add仿函数
template <typename T>
class Add {
public:
    T operator()(T a, T b) {
        return a + b;
    }
};

// Minus仿函数
template <typename T>
class Minus {
public:
    T operator()(T a, T b) {
        return a - b;
    }
};

template <typename T, typename Func>
T twoNumOper(T num1, T num2, Func func) {
    return func(num1, num2);
}

int main() {
    Add<int> add;
    Minus<int> minus;

    //// 仿函数使用
```

```
    std::cout << add(1,1) << std::endl;
    std::cout << minus(1,1) << std::endl;

    //// 抽象
    int result1 = twoNumOper(5, 3, add);
    int result2 = twoNumOper(5, 3, minus);

    std::cout << "Add result: " << result1 << std::endl;
    std::cout << "Minus result: " << result2 << std::endl;

    return 0;
}
```

8.4. 匿名函数(lambda函数)

简单使用 `[capture](parameters)->ret_type{body}`

- `[capture](parameters)->ret_type{body}` 整体赋值给一个变量，cpp中为 `std::function` 类型，但我们偷懒就直接 `auto` 即可
- `[capture]` 相当于不能传参进入，只能调用上面已经出现的变量
  - 不带引用符号，例如下面 `a1`，就只能读取 `a1` 而不能修改
  - 带引用符号，例如下面 `a2`，可以读取 `a2` 且可修改 `a2`
- `(parameters)` 相当于函数参数，需要传参进入
- `->ret_type` 相当于函数的返回值类型
- `{body}` 就写函数的操作就行

```
int a1 = 1, a2 = 2, a3 = 3;
auto addNums = [a1,&a2](int x, int y) -> int {
    a2 = 22;
    // a1 = 11; 值捕获can't change
    //其实[]里的东西就是从外界传进去的变量，只是不用通过参数传进去
    return x+y+a1+a2;
};
std::cout << addNums(-1,0) << std::endl;
//↑↑↑ x+y+a1+a2 = -1 + 0 + 1 + 22

std::cout << "a1:" << a1 << " a2:" << a2 << std::endl;
//↑↑↑ a1 = 1, a2 = 22
```

8.5. std::bind & std::function

std::function 可以包装任何可调用的**函数**(其实就是可以赋值等于，std::function相当于一个通用包装类，也可以理解成通用的一个塑料袋，什么零食(可调用的函数)都可以往里装，然后统一被快递发送出去，然后送到你手上处理掉他())，例如普通函数、函数指针、仿函数、lambda、模板函数、**类成员函数(较为特殊，需要用到std::bind进行绑定转换，但其实只会这一种就行了，因为ros的我们一般都是写类，绑定回调函数一般就是这种)**

```
//类实例化，可包装一些形如 <int (int,int)> 的函数，也就是可以直接赋值
```

```
std::function<int (int,int)> callback;
// 普通函数
int minus(int a, int b){
    return a-b;
}
// 仿函数
template <typename T>
class Add {
public:
    T operator()(T a,T b){
        return  a+b;
    }
};
// lambda
//num1^(num2)
auto lambda_pow = [](int num1, int num2)->int{
    return pow(num1,num2);
};
// 模板函数 自行体会
// 类成员函数
class Test{
public:
    int mult(int a,int b){
        return a*b;
    }
};

//大一统的处理函数，已function类作为参数，接受一样的类型函数作为内部处理函数，想用什么操作就传什么操作
int twoNumOper(int num1, int num2, std::function<int (int,int)> cb){
    return cb(num1,num2);
}
```

以上函数都可以被std::function类包装，其实就是**赋值**

```
//main-----------------------
int main(int argc, char *argv[]) {

    callback = std::bind(minus,std::placeholders::_1,std::placeholders::_2);

    //普通函数
    callback = minus;
    std::cout << callback(1,5) << std::endl;

    //仿函数
    Add<int> add;
    callback = add;
    std::cout << callback(1,5) << std::endl;

    //lambda
    callback = lambda_pow;
    std::cout << callback(1,5) << std::endl; //1的5次方

    //类成员函数--必须用std::bind来绑定
```

```
    Test test; // 初始化类对象
    /**
     * &Test::mult : 取类成员函数地址(看参数类型是&&)
     * test        : 类对象实例化
     * std::placeholders::_1 : 占位参数
     * std::placeholders::_2 : 占位参数
     */
    callback = std::bind(&Test::mult,test,std::placeholders::_1,std::placeholders::_2);
    std::cout << callback(1,5) << std::endl; //1*5

    // 大一统的两数操作函数，传操作函数作为参数进去 (可见std::function 可以接受任何的函数，非常满足我们回调函数的需求--课传进去任何函数作为数据操作)
    std::cout << twoNumOper(1,5,minus) << std::endl;        //普通函数
    std::cout << twoNumOper(1,1,add) << std::endl;          //仿函数
    std::cout << twoNumOper(1,1,lambda_pow) << std::endl;   //lambda
    // 类成员函数传参
    std::cout <<
twoNumOper(1,5,std::bind(&Test::mult,test,std::placeholders::_1,std::placeholders::_2)) <<
std::endl;

    return 0;
}
```

9. thread

多线程`thread`和信号量机制`mutex`.

- `std::thread`创建线程，接收传入自定义函数和传入参数，所以可以看到上面讲的`callback`是一种非常好用的东西
- `std::mutex`线程锁，两个线程同时访问同一个共享资源时(例如下面例子的`std::cout<<`就是共享的输出窗口资源)，如果不加管理打印的数据会乱掉，因为相当于两个人抢着用了，所以这里用`std::mutex`给它上锁，谁先用就上锁，用完了再释放。如果发现资源已经上锁那么久等待别人用完。道理很简单，代码也很简单，下面十个例子。

```
#include <iostream>
#include <thread>
#include <mutex>
namespace stl {
    struct Point {
        float x{0}, y{0}, z{0}; // 初始化变量为0
    };
}
std::mutex mtx;
void movePointPosition(stl::Point point, float dx, float dy, float dz) {
    while (true){
        point.x += dx;
        point.y += dy;
        point.z += dz;
        mtx.lock(); //即将输出，上锁
```

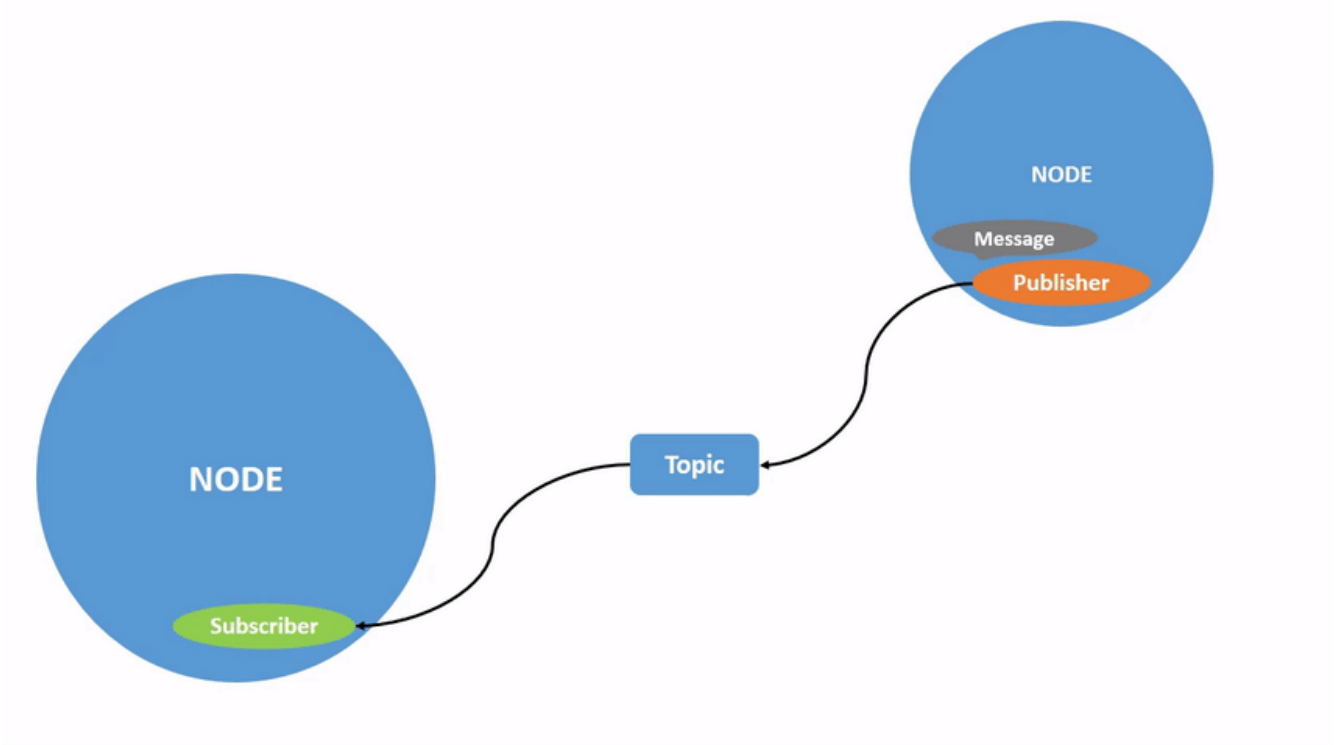
```
        std::cout << "point:[x y z]-" << point.x << " " << point.y << " " << point.z <<
std::endl;
        mtx.unlock(); //输出完毕，解锁
        //std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void printTest(){
    while(true){
        mtx.lock(); //即将输出，上锁
        std::cout << "aaaaaaa" << std::endl;
        mtx.unlock();//输出完毕，解锁
        //std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

int main() {
    stl::Point point1{1,1,1};
    std::thread thread1(movePointPosition,point1,1,1,1);
    std::thread thread2(printTest);

    while(1){}
    return 0;
}
```

10. 代码规范()

google规范

11. 综合案例，仿照ROS2写一个发布者和订阅者机制的cpp范例.

详细源码不给大家，下面记录的是要点讲解，源码参见视频一步一步写出来，会作为考核的一个题目。



- `SharedBuffer`：存储各种各样`Message`的`buffer`,用来为`Publisher`和`Subscriber`发布和接收`Message`
- `Publisher`：模板类，支持自定义结构体`message`，发布`message`(本质就是向`SharedBuffer`的`buffer_`中添加)，但需要触发时机
- `Timer`:模板类，支持自定义定时任务函数`std::function<void()>`(可优化适应可变个数参数)，基于线程`std::thread`周期性运行任务,简单使用触发`Publisher`发布`Message`
- `Subscriber`:模板类，支持自定义`message`和`message`对应的操作函数`std::function<void(const MessageT&)>`(可优化适应可变个数参数)
- `Node`:`Timer`,`Publisher`,`Subscriber`构造仓库和规范接口

11.1. `SharedBuffer`

用来存储来往的`Message`,因为可能会有多对`Publisher - Subscriber`,每对都拥有一个自定义的`Message`和`topic_name`，都暂存在一个`SharedBuffer`中，其中`Message`由`topic_name`唯一标识。所以这里这种需求很适合使用`std::unorded_map<std::string, std::any> buffer_`，其中`std::any`是`C++17`引入的类型,可以存储任意数据类型(s.t.`1.基本-int、double... 2.自定义-struct ③自定义-class etc`)，这里引入这个是因为不同的需求对应的`MessageT`可能是不一样的，所以需要自适应这种情况。在这里的话因为多组`pub-sub`都公用一个`SharedBuffer`，所以我们用一个单例模式的`SharedBuffer`类来维护这个东西，并且考虑到多线程问题需要对`buffer_`加`mutex`锁.

- `buffer_`-数据结构组织
  - type：`std::unorded_map<std::string, std::any>`
  - read：`MessageT msg = std::any_cast<MessageT>(buffer_["topic_name"]);`
  - write:`buffer_["topic_name"] = message;`
- `SharedPtr`-单例模式(即此`class`仅有一个`static`实例化对象)，需要做以下处理
  - 普通构造函数设置为`private`-`private: SharedBuffer() = default;`
  - 禁止拷贝构造函数-`SharedBuffer(const SharedBuffer&) = delete;`

- 禁止赋值重载运算符-`SharedBuffer& operator=(const SharedBuffer&) = delete;`
- 提供给外界访问单例的唯一接口`getInstance`

```
class SharedBuffer {
public:
    static SharedBuffer& getInstance() {
        static SharedBuffer instance;
        return instance;
    }
    //others code....
};
```

- 多线程锁-(操作系统课程原理，问题和原因不赘述)
  - `method1 - std::mutex`

```
std::mutex mutex_;
mutex_.lock(); //访问前上锁
//TODO:访问共享资源buffer_(读or写)
mutex_.unlock(); //访问后解锁
```

  - `method2 - std::lock_guard`

```
std::mutex mutex_;
std::lock_guard<std::mutex> lock(mutex_);//访问前上锁
//TODO:访问共享资源buffer_(读or写)
/*****访问后无需显式解锁,lock_guard自动解锁*****/
```

11.2. `Publisher`

这个就非常简单了

- 模板类,需要外界用户传入其自定义的`<MessageT>`结构体类型
- `topic_name`:`pub-sub`一一对应的唯一标识
- `publish`函数：向`SharedBuffer`中添加`message`.所以这里设计如何访问单例`SharedBuffer`的问题以及调用`addMessage`(`SharedBuffer`中定义)的问题了,详细请见视频

11.3. `Timer`

这里用一个简化的定时器，主要涉及到以下几个点

- 自定义周期，作为构造函数参数
- 自定义函数，这里就直接简化了，用`std::function<void()> callback`来作为构造函数的参数以接收外界函数。
- 多线程, 因为是周期运行，死循环，所以开个线程来运行.`std::thread`.至于`std::thread`示例程序如何使用可问`chatgpt`,下面就是一个简单的例子

```
#include <iostream>
```

```
#include <thread>

void printNum(float num){
  while(true){//死循环进程运行函数
      std::cout << "num:" << num << std::endl;

      // 延迟500ms
      auto sleep_time = std::chrono::milliseconds(500);
      std::this_thread::sleep_for(sleep_time);
  }
}
int main() {
  //创建线程，也是使用函数作为构造函数的参数，后续是参数，std::thread库使用了自适应不定参数的技术(类似于printf),所以也可以传入两个参数，只需要和传入的函数对应上即可
  std::thread thread(printNum,100);

  while(1){} //卡住主线程
  return 0;
}
```

- 周期运行(死循环),所以涉及到如何获取标准时间,然后延迟固定时间.

```
auto sleep_time = std::chrono::milliseconds(500);//500ms,这里std::chrono是cpp标准时间库，会用就行，有需求就问chatgpt
std::this_thread::sleep_for(sleep_time);
```

11.4. `Subscriber`

基本技术点和`Timer`类似

- 模板类，使用时需要对应上`message`的`MessageT`结构体类型
- `topic_name`:指定订阅者的话题名以和`publisher`对应上
- 传入的回调函数类型为`std::function<void(const MessageT&)> callback`
- 多线程，每个订阅者需要开一个线程，方法和`timer`类似
- 调用时机(检测`SharedBuffer`中对应的`message`更新后调用)
  - 需要对`SharedBuffer`加一个新的处理，详见视频

11.5. `Node`

- `node_name`:节点名称
- `工厂`-创建以上几个类，为了统一接口和方便统一编写规范
  - `createPublisher`
    - `new`分配内存，因为函数中是局部变量，所以需要new到堆上
    - 可选，`std::shared_ptr`替代指针和`std::make_shared`替代`new`
    - example
      其中`createPublisher`函数的参数就是`Publisher`类的`构造函数`对应的参数

```
template<typename MessageT>
Publisher<MessageT>* createPublisher(const std::string& topic_name) {
    return new Publisher<MessageT>(topic_name);
}
```

    - `createSubscriber`同上
    - `createTimer`同上

- 继承
  我们要求所有其他结点创建时都需要继承`Node`类，并通过`工厂`来创建这些东西
  `publisher`,`timer`,`subscriber`...
- 多态
    - 有一个规定的`loop`函数,作为Node的主运行程序
    - 虚函数和函数重写(cpp的语法，可以自己看黑马程序员教程)

```
//Node class:
virtual void loop(){
    std::cout << "default node loop function" << std::endl;
}
//publisherNode class:
class PublisherNode :public Node{
    //其他略
    void loop() override{
        timer_->start();//开始定时发布message
    }
}
//SubscriberNode class:
class SubscriberNode :public Node{
    //其他略
    void loop() override{
        sub_->startSubCallback();;//开始接收处理订阅的message
    }
}
```

好的，大概就是这么多，如果你从头看到了尾，那恭喜你对cpp已经有较深的了解以及如何应用了，核心还是编程思想，语法都可以慢慢学，学不会就先搁下，时间到了再次回顾自然就看懂了。