P13 VS的目录设置

1、不将项目文件和解决方案文件放在一起

2、将源文件放在专门的src里面，

3、更改 所有配置 和 所有平台 的输出目录 和 中间目录，分别为：

$(SolutionDIr)\bin\$(Platform)\$(Configuration)\ 和

$(SolutionDIr)\bin\intermediates\$(Platform)\$(Configuration)\

于是最终生成的文件和中间文件都会按配置和平台存放在不同文件夹下面

P17 引用

1、引用只在定义时绑定，如果在定义后将变量赋值给引用，则不会重新绑定，而只是幅值。

P21 static

1、类外面的static：表示在link阶段这个变量（函数）只在自身所在的编译单元(.obj)可见，也就是在link阶段，此文件里的static变量不能被其他文件里找到。

2、如果要在A文件定义一个变量，而在B文件使用，就在B文件里用extern声明。

3、注意包含include头文件的情况：相当于把头文件复制进来。所以include之后就相当于在同一编译单元里面。

P22 static

1、类里面的static：在所有实例中都只存在同一个，相当于一个类内的全局变量。在使用的时候需要在全局某个地方声明，如：int Entity::x; 在调用的时候可以用实例名也可以用类名，如 e.x 或 Entity::x

P23 static

1、函数内的static变量，不能在函数作用域外被访问，但是多次调用该函数，其值有记忆性，也就是说函数调用结束后，其内存仍然没被释放掉，延长了其生命周期。此方法适用的一个例子是单例Singleton：

Class Singleton  
{  
public:  
 static Singleton& Get()

{

static Singleton instance;

return instance;

}

void Hello() {}

};

int main()

{

Singleton::Get().Hello();

}

如果在不把instance定义为static，则在Get()作用域结束后，该实例就会被销毁，由于返回的是Singleton的引用，就会导致引用错误，如果是返回复制就没问题。

P24 ENUMS

1、枚举类型用法：

enum Level

{

LevelError = 0, LevelWarning, LevelInfo //（注意此处没有分号 ; ）

}

可以指定第一个从多少开始，默认是0，也可以指定整型类型：enum Level:unsigned int 等

枚举类型其实和整数一样，只不过有了命名，且自动设置大小

2、在外面调用类内部的枚举变量时，相当于类是作用域，如Log::LevelError，枚举变量的计算和整数一样，都可以比较大小，运算之类的

P25 构造器

1、构造函数主要目的是在实例化的时候初始化，其名字和类名一致，在其里面初始化成员。

P26 析构器

1、析构函数是在对象被清理时自动调用的，一般在出作用域的时候会自动销毁，则会调用析构，析构里面主要是手动释放在堆heap上面手动分配的内存

P27 继承

1、一个简单的继承示例：Player继承自Entity

class Entity

{

};

class Player : public Entity

{

};

P28 虚函数

1、用virtual在基类函数前面进行标记，在子类函数末尾用override进行重写标记。

2、一个简单的示例：

class Entity

{

public:

virtual std::string GetName(){return "Entity"; }

};

class Player : public Entity

{

private:

std::string m\_Name;

public:

Player(const std::string name):m\_Name(name){}

std::string GetName() override {return m\_Name;}

};

void PrintName(Entity\* entity)

{

std::cout<<entity->GetName()<<std::endl;

}

int main()

{

Player \*p = new Player("ydc");

PrintName(p);

std::cin.get();

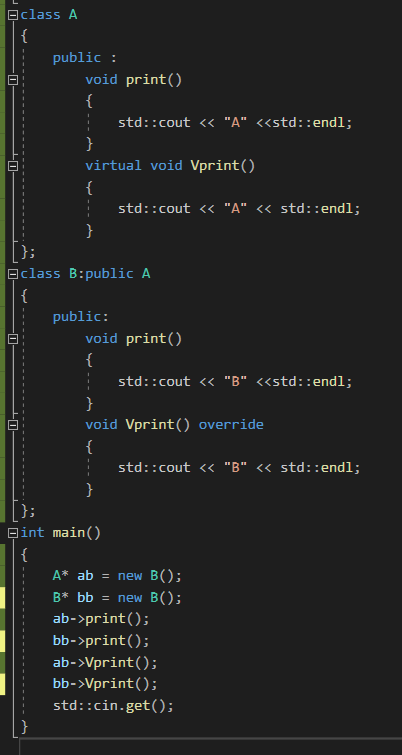
}

P29 纯虚函数

1、在虚函数后面加上 “ =0 ”标志为纯虚函数，则此基类为抽象类，不能被实例化，若子类没有实现这个纯虚函数，则也不能被实例化。

2、一个例子，如B继承自A，创建对象B的时候如果用A指针，则调用A的非虚函数，B的虚函数；如果用B指针，则调用B的非虚函数，B的虚函数。

即调用规则：指针决定调用非虚函数，对象决定调用虚函数



P30 访问控制

1、private：只有类内和友元可以访问，子类也不能访问，对象在外面也不能访问。

2、protected：类内和子类（或者子类的子类）可以访问

3、public：任何地方都可以访问

4、访问控制不只是作为给计算机看的，而是更多的作为一种标记，给自己或者别人看， 以怎样来调用代码

P31 数组

1、求数组(int)的长度：sizeof(a)/sizeof(int)

2、分别在栈和堆上面分配数组的内存：（栈：离开作用域自动释放；堆：必须手动释放delete）

栈：int a[5]; // a的名字的值是一个地址，这个地址的值是数组首元素，打印首元素地址

堆：int \*p = new int[5]; //p的值是一个地址，这个地址的值是数组首元素地址，打印首元素地址

P32 string

1、传统的char数组有两种形式：就类似数组的两种形式

const char \*p = “ydc”;

char c[3] = {‘y’,’d’,’c’};

P33 string literal

1、允许我们定义raw string的字符串常量，以R”(…)”的形式，括号里面所见即所得，也支持换行，不再需要转译

P34 const

1、const放在变量类型前，默认其不能被改，如：const int a = 20;

2、const放在指针类型前，默认指针所指的地址的值不能被改，如：const int\* p = new int;则\*p不能被改，另一种写法是：int const\* p = new int;

3、const放在指针类型后，默认指针不能被改，如：int\* const p = &a;则p只能指向a，不能指向其他地址

4、const放在成员函数后({前面)，代表这个函数不能对值进行修改，里面不能修改变量值，和指针的值（所指的地址），但是所指的地址处的值可以改，如：int Get() const {}

5、const在函数的形参列表里面，如果是复制和引用，则不能更改，如果是指针，则指针所指的地址可改，指针所指的地址的值不能改

6、如果想在const成员函数内改动变量，则把变量修饰为mutable

P35 mutable

1、一个用法是在const成员函数里面修改变量

2、另一个用法是在lambda表达式里面进行修饰

P36 成员列表初始化

1、示例：

Entity():m\_Name(“ydc”){}

Entity(const std::string& name):m\_Name(name){}

P37 Condition Assignment

1、条件赋值：int a = 5;int b = a > 3 ? 1:2;

P38 如何创建对象

1、栈和堆的区别：栈：生命周期完全由作用域控制，其总内存相对较小；堆：需要手动控制内存分配与释放，速度较栈慢

2、一个使用堆的例子，在作用域外定义一个空指针，在作用域内new一个对象，传递给域外的指针，那么出作用域时，这块内存空间仍然存在

P39 new关键字

1、new和delete在处理数组的时候：int \*p = new int[5]; delete[] p;

2、new和delete底层在调用malloc和free

P40 隐式类型转换

1、可直接用构造函数形参列表的同类型的常量来进行初始化，会进行隐式转换，如：

Entity(const std::string& name):m\_Name(name){}

则可以直接：Entity e = “ydc”;

Entity(int x):m\_x(x){}

则可以直接：Entity e = 2;

若想禁止此类隐式转换，则在构造函数前面加explicit

2、注意字符串常量””ydc”实际类型为char[]，有时被隐式转换为string，如果不能隐式转换，则强制转换std::string(“ydc”)

P41 操作符重载

1、示例：用operator+作为函数名，作为成员函数

vec2 Add(const vec2& other) const

{

return vec2(m\_x+other.m\_x,m\_y+other.m\_y);

}

vec2 operator+(const vec2& other) const

{

return Add(other);

}

2、对<<的重载示例：非成员函数

std::ostream& operator<<(std::ostream& stream, const vec2& v)

{

stream<<v.m\_x<<","<<v.m\_y<<std::endl;

return stream;

}

3、对于==和！=的重载：

bool operator==(const vec2& other) const

{

return m\_x==other.m\_x && m\_y==other.m\_y;

}

bool operator!=(const vec2& other) const

{

return !(\*this==other);

}

注意此处的\*this，代表对象本身

P42 this关键字

1、this代表指向该对象的指针

P43 栈里的对象生命周期

1、构造和析构的顺序：成员比对象先构造，后析构

2、一个既可以在堆上分配内存，又可以在出作用域时自动释放的例子：

class ScopePtr

{

private:

Example\* m\_Ptr;

public:

ScopePtr(Example\* ptr):m\_Ptr(ptr){}

~ScopePtr(){delete m\_Ptr;}

};

int main()

{ {

ScopePtr e = new Example();

}

std::cin.get();

}

P44 智能指针

1、unique\_ptr，不能被复制，示例：

std::unique\_ptr<Entity> e = std::make\_unique<Entity>();

2、shared\_ptr，原理时引用计数，可以被复制，比如多个指针都存储某个对象的地址，只有当所有指针都被销毁时，这个对象才被析构，示例：

{

std::shared\_ptr<Entity> e;

{

std::shared\_ptr<Entity> e2 = std::make\_shared<Entity>();

e = e2;

}//出此作用域时，e2被释放，但是entity对象没被释放，因为e还没被销毁

}//出此作用域时，e也被销毁，此处entity也被销毁

P45 拷贝构造函数

1、以string的拷贝构造为例：

class String

{

private:

char\* m\_Buffer;

unsigned 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);

}

String(const String& other):m\_Size(other.m\_Size)

{

m\_Buffer = new char[m\_Size+1];

memcpy(m\_Buffer,other.m\_Buffer,m\_Size+1);

} // 此函数为拷贝构造函数，new出一块内存，复制原来的数组

~String()

{

delete [] m\_Buffer;

}

char& operator[](const unsigned 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;

}

int main()

{

String s = "ydc";

String s2 = s;

std::cin.get();

}

P46 箭头操作符

1、可用于指针调用成员：p->x 等价于 (\*p).x

2、可用于计算成员变量的offset：

struct vec2

{

int x,y;

float pos,v;

};

int main()

{

int offset = (int)&((vec2\*)nullptr)->x; // x,y,pos,v的offset分别为0,4,8,12

std::cout<<offset<<std::endl;

std::cin.get();

}

P47 std::vector

1、形式：std::vector<int> a;

2、a.push\_back(element);// 后面插入

3、size = a.size();// 返回元素数量

4、a.erase(index);// 删除某一个元素，index为迭代器类型，如 ：a.begin()+2

5、范围for循环：for(const int & element : a) {}

P48 std::vector 的优化使用

1、示例：预先预留(reserve)内存；用emplace\_back而不是push\_back

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;

}

};

int main()

{

std::vector<Vertex> vertices;

//ver 1 : copy 6 times

vertices.push\_back({ 1,2,3 });

vertices.push\_back({ 4,5,6 });

vertices.push\_back({ 7,8,9 });

//ver 2 : copy 3 times

vertices.reserve(3);

vertices.push\_back({ 1,2,3 });

vertices.push\_back({ 4,5,6 });

vertices.push\_back({ 7,8,9 });

//ver 3 : copy 0 times

vertices.reserve(3);

vertices.emplace\_back(1, 2, 3);

vertices.emplace\_back(4, 5, 6);

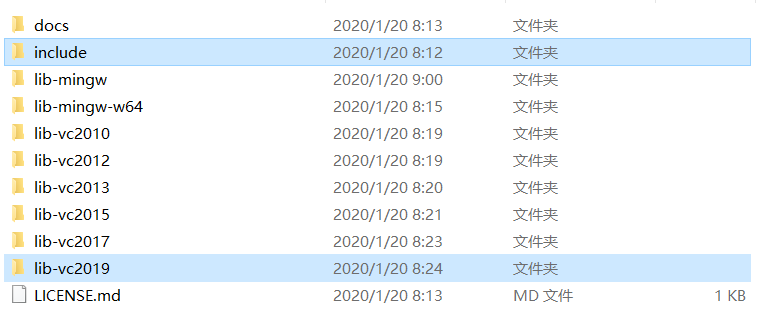
vertices.emplace\_back(7, 8, 9);

std::cin.get();

}

P49 static link 静态库的使用

1、以GLFW为例，在www.glfw.org 网站上面下载32位(此处判断位数，是看编译目标平台)的windows二进制文件，得到include文件和libraries文件（有很多个，选择最新的）







2、在vs工程sln同级目录下面新建文件夹 Dependencies\GLFW，将include和libraries文件拷贝进去

3、在工程名上右键，点击 属性-配置属性-C/C++-常规-附加包含目录，确保上面的配置是 所有配置，平台是 Win32 。在附加包含目录填入 $(SolutionDir)Dependencies\GLFW\include，此为相对目录，可以看见include下面一个头文件为 GLFW/glfw3.h，故可以#include"GLFW/glfw3.h"

4、完成上述步骤后，可以编译通过了，但是link会出问题，原因是没有导入libraries，头文件的声明找不到源文件的定义。

5、在工程名上右键，点击 属性-配置属性-链接器-常规-附加库目录，同样确保配置和平台准确，在附加库目录填入 $(SolutionDir)Dependencies\lib-vc2019,然后在 链接器-输入-附加依赖项 内填入 glfw3.lib，注意不要把原来的继承项%(AdditionalDependencies)删去

6、用 int a = glfwInit();进行测试，打出a为1即为配置成功

P50 动态链接库

1、将上面的glfw3.lib换成glfw3dll.lib后，运行提示说找不到glfw3.dll，一个解决方法是将此文件直接复制到exe的生成目录里面

P51 在一个解决方案里面建多个项目，分别于用作静态库和应用

1、在一个名为Game的解决方案里面新建Game项目和Engine项目，分别修改配置类型为（.exe和.lib），点击全部折叠，分别在Game和Engine下面新建src文件夹用于存放代码，在Game/src下面新建Application.cpp，在Engine/src下面新建Engine.h和Engine.cpp用于编写函数，然后需要在Application.cpp里面include”../../Engine/src/Engine.h”，（../代表上级目录），但这不是一个好的方法，因为这是相对路径。另外一个办法是设置附加包含目录：在Game的属性-配置属性-C/C++-附加包含目录，设置为$(SolutionDir)Engine\src，于是就可以直接include”Engine.h”，但此时只能编译通过而会链接失败，所以需要在Game上右键添加-引用，选择Engine作为其引用，这样添加的好处就是以后生成Game的时候就会自动生成Engine

P52 如何处理返回值为多个类型变量的时候

1、比如要返回一个int和一个string，就创建一个struct包含int和string，然后返回类型设置为这个struct

2、传入形参为int和string的引用或者指针，然后在里面修改，而返回void

3、如果要返回同类型的值，可返回数组如std::array(存储在stack里，比heap快一点)或者std::vector（存储在heap里面）

4、另一种返回不同类型的方法，用std::tuple或者std::pair，但是在索引的时候其释义并不清晰，所以不如struct的方法好

P53 模板

1、与函数重载进行比较，如要定义一个print函数，需要打印不同类型的变量，用函数重载的方法则需要写多个print函数。

2、而用模板的方式，只需要写一次：

template<typename T>

void print(T value)

{

std::cout<<value<<std::endl;

}

typename也可以换成class，在使用时可指定类型，也可以不指定类型：

print<int>(5);

print("hello world");

3、模板还可用于创建一个类的时候，往里面传参数(见下例)或者往里面传类型（比如std::vector）

template<int N,typename T>

class Array

{

private:

T m\_Array[N];

public:

int GetSize(){return N;}

};

使用的时候：Array<5,int> array;

P54 stack vs heap memory

1、stack在分配内存的时候，是顺序分配，仅仅像堆一样移动指针，需要多少就移动多少，所以速度很快。如果连续定义几个变量，可以在内存里面看到是连续的，如果突然new一个变量，可以发现指针自己的地址也是连续存储的，但是每个指针所指的地址不是连续的

P55 宏

1、宏：在预处理器时执行查找和替换的功能，如：#define WAIT std::cin.get()

但是不建议这样做，会让代码更加难懂

2、另一种用法，如：

#define LOG(x) std::cout<<x<<std::endl;

使用时：LOG("ydc");

3、根据是debug还是release预定义不同的变量，在项目属性-C/C++-预处理器-预处理器定义，里面为debug和release分别添加不同的变量 PR\_REBUG 和 PR\_RELEASE，那么在项目选择不同编译模式的时候，就会分别预定义这两个变量，用法如下：

#ifdef PR\_DEBUG

#define LOG(x) std::cout<<x<<std::endl;

#else

#define LOG(x)

#endif

int main()

{

LOG("ydc");

std::cin.get();

}

也可以在预处理器定义里面写 PR\_DEBUG=1 那么在外面的判断就是 PR\_DEBUG == 1

4、\代表换行：

#define MAIN int main() \

{\

std::cin.get();\

}

MAIN

P56 auto

1、在使用iterator或者类型名过长的时候，可以用auto，如：

std::vector<std::string> strings;

strings.push\_back("Apple");

strings.push\_back("Orange");

for (std::vector<std::string>::iterator it = strings.begin();

it != strings.end(); it++)

{

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

}

for (auto it = strings.begin();it != strings.end(); it++)

{

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

}

2、当类型名过长的时候可以使用using或者typedef：

using DeviceMap = std::unordered\_map<std::string,std::vector<Device\*>>;

typedef std::unordered\_map<std::string, std::vector<Device\*>> DeviceMap;

P57 std::array

P58 Function Pointer

1、函数实际上是CPU的指令，本质是二进制文件，可以获取这些二进制文件的地址，即函数指针：

void HelloWorld()

{

std::cout<<"ydc"<<std::endl;

}

int main()

{

auto function = &HelloWorld; // & 可省略,其中的 function 可自定义

function();

std::cin.get();

}

此时function的类型为 void(\*function)()

2、另一种是带有传入参数的情况，且用typedef将复杂的函数类型名称进行更换：

void HelloWorld(int a)

{

std::cout<<"ydc: "<<a<<std::endl;

}

int main()

{

typedef void(\*HelloFunction)(int);

HelloFunction function = HelloWorld;

function(5);

std::cin.get();

}

3、1和2只是简单阐述了规则，现在来看一个实际的用法：可以将一个函数指针传到另外一个函数的形参列表里面去：

void PrintValue(int value)

{

std::cout<<value<<std::endl;

}

void ForEach(const std::vector<int>& values, void(\*func)(int))

{

for (int value : values)

{

func(value);

}

}

int main()

{

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

ForEach(values,PrintValue);

std::cin.get();

}

4、另一种方法是使用std::function,C++11引入

std::function <void(int)> func 替代 void(\*func)(int)

两者的区别见以下链接：当然优先使用std::function，因为函数指针不能capture一些东西

https://stackoverflow.com/questions/25848690/should-i-use-stdfunction-or-a-function-pointer-in-c

P59 Lambda

1、在使用函数指针的时候就可以考虑使用lambda表达式

2、使用详见以下链接：

<https://en.cppreference.com/w/cpp/language/lambda>

3、lambda的用法示例：

一个是作为函数指针传入某些函数，如此处的std::findif

另一个是捕获外面的变量，然后也可以直接调用lambda函数

int main()

{

std::vector<int> array = {1,2,3,4,5};

auto it = std::find\_if(array.begin(),array.end(),[](int v){return v > 3;});

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

std::string s = "value:";

auto lambda = [=](int value) {std::cout << s << value<< std::endl; };

lambda(10);

std::cin.get();

}

P60 为什么不使用 using namespace std;

1、会让里面的里面的函数和类变得混淆，不知道是否来自std；

2、如果公司有自己的模板库，然后里面有很多重名的类型或者函数，就容易弄混；

3、千万不要在头文件里面使用using namespace

P61 什么是namespace

1、比如GLFW库，是一个C语言的库，没有namespace，就需要在函数名里面嵌入GL以进行区分,但在C++里面可以用namespace解决这个问题

2、可以在namespace里面再定义namespace

3、也可将namespace指代给另外一个变量，如：

namespace a = apple;

使用时直接：

a:print();

P62 thread

1、

2、来一个代码示例：

#include<iostream>

#include<thread>

static bool is\_Finished = false;

void DoWork()

{

using namespace std::literals::chrono\_literals; // 为 1s 提供作用域

std::cout << "Started thread ID: "<<std::this\_thread::get\_id()<<std::endl;

while (!is\_Finished)

{

std::cout<<"Working..."<<std::endl;

std::this\_thread::sleep\_for(1s);//等待1s

}

}

int main()

{

std::thread worker(DoWork);

std::cin.get(); // 其作用是阻塞worker线程

is\_Finished = true;// 让worker线程终止的条件

worker.join();// 让主线程等待worker线程

std::cout << "Finished thread ID: " << std::this\_thread::get\_id() << std::endl;

std::cin.get();

}

P63 Timing

1、有两种选择，一种是用平台特定的API，另一种是用std::chrono，此处推荐后者

2、一个比较好的方法是建立一个Timer类，在其构造函数里面记下开始时刻，在其析构函数里面记下结束时刻，并打印从构造到析构所用的时间。如此就可以用这样一个类来对一个作用域进行计时：

#include<iostream>

#include<chrono>

struct Timer

{

std::chrono::time\_point<std::chrono::steady\_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;

}

};

void Function()

{

Timer timer;

for (int i = 0;i<100;i++)

std::cout<<"Hello"<<std::endl;

}

int main()

{

Function();

std::cin.get();

}

P64 多维数组

1、通过指针的方式在heap上分配空间：

int\* array = new int[50];

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

int\*\*\* a3d = new int\*\*[50];

for (int i = 0; i < 50; i++)

a2s = new int[50];

for (int i = 0; i < 50; i++)

delete[] a2d[i];

delete[] a2d;

2、存储二维数组一个很好的优化方法就是：存储在一维数组里面：

int\*\* a2d = new int\*[5];

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

{

a2d[i] = new int[5];

for (int j = 0; j < 5; j++)

{

a2d[i][j] = 2;

}

}

int\* a1d = new int[5\*5];

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

{

for (int j = 0; j < 5; j++)

{

a1d[i+j\*5] = 2;

}

}

P65 Sorting

1、此处主要介绍std::sort，并结合lambda表达式可进行很灵活的排序：

#include<iostream>

#include<vector>

#include<algorithm>

int main()

{

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

std::sort(values.begin(), values.end(), [](int a, int b)

{

// 此处两个判断可以将等于2的值放到末尾

if(a == 2)

return false;

if(b == 2)

return true;

return a < b;

});

// 此处输出为 1，3，4，5，2

for(const int &v:values)

std::cout<<v<<std::endl;

std::cin.get();

}

P66 type punning

1、可以将同一块内存的东西通过不同type的指针给取出来

2、指针的类型只是决定了其+1或者-1时地址的偏移量

3、以下这个示例说明了：弄清楚内存分布的重要性

struct Entity

{

int x,y;

};

int main()

{

Entity e = {2,3};

int\* pos = (int\*)&e;

std::cout<<pos[0]<<","<<pos[1]<<std::endl;

int y = \*(int\*)((char\*)&e+4);

std::cout << y << std::endl;

std::cin.get();

}



上图为e的内存分布，只有八个字节

P67 union

1、如果想要以不同形式去取出同一块内存的东西，可以用type punning，也可以使用union

#include<iostream>

#include<vector>

#include<algorithm>

struct vec2

{

float x,y;

};

struct vec4

{

union

{

struct

{

float x,y,z,w;

};

struct

{

vec2 a,b;

};

};

};

void PrintVec2(const vec2& vec)

{

std::cout<<vec.x<<","<<vec.y<<std::endl;

}

int main()

{

vec4 vector = {1.0f,2.0f,3.0f,4.0f};

PrintVec2(vector.a); // 输出 1，2

PrintVec2(vector.b); // 输出 3，4

vector.z = 10.0f;

PrintVec2(vector.a); // 输出 1，2

PrintVec2(vector.b); // 输出 10，4

std::cin.get();

}

P68 Virtual Destructor

1、当A继承自B，且new一个B\*的指针指向A对象时，在delete时，会遗漏A的析构函数：

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; // base constructor-base destructor

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

Derived\* derived = new Derived();

delete derived; // base constructor-derived constructor-derived destructor-base destructor

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

Base\* poly = new Derived();

delete poly; // base constructor-derived constructor- base destructor此处少了一次析构

std::cin.get();

}

2、要解决这个问题，只需要把析构函数定义为虚函数即可，即在基类的析构函数前面加上：virtual

P69 Casting

1、C类型的cast：如：

double b = 1.35;

int a = b; // 隐式转换

int c = (int)b; // 显式转换

2、C++的cast：

static\_cast:基础的类型转换，结合隐式转换和用户自定义的转换来进行类型转换

dynamic\_cast:安全地在继承体系里面向上、向下或横向转换指针和引用的类型，多态转换

reinterpret\_cast：通过解释底层位模式来进行类型转换

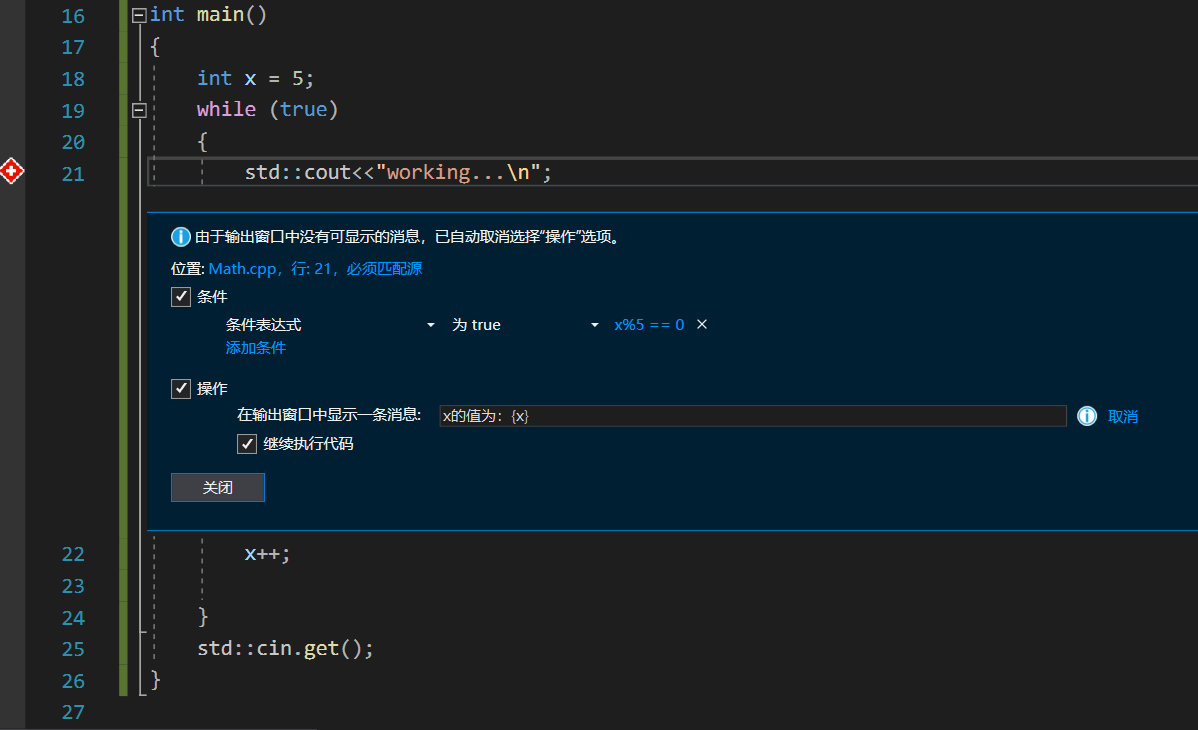
const\_cast：添加或者移除const性质

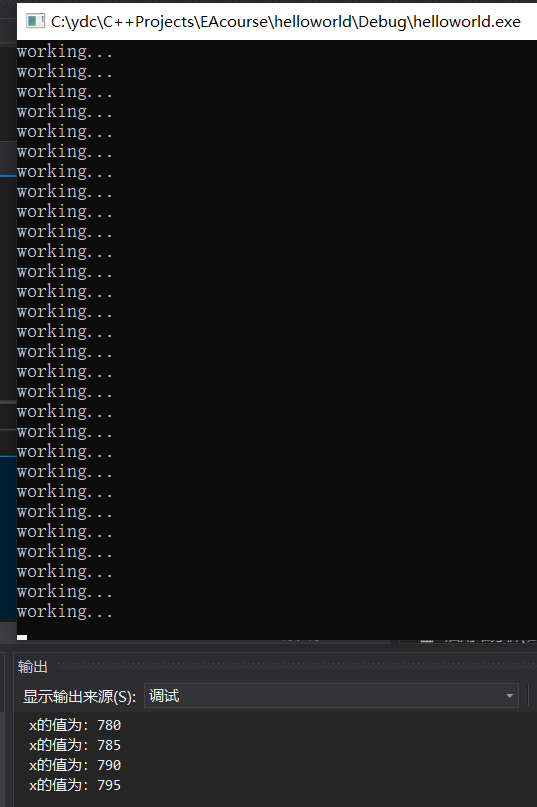
可详见链接：<https://en.cppreference.com/w/cpp/language/static_cast>

P70 条件断点和断点操作

1、条件断点，当达到什么条件触发断点；断点操作：当触发断点后执行什么操作（在窗口输出什么）

2、一个示例，在一个死循环里面，x每次加一，当x被5整除时触发断点，触发断点后打出x的值，并且可以在调试过程中，随时更改断点的条件和动作，并且可以设置是否让程序继续运行





P71 Safety in C++

P72 Precompiled Headers （预编译头文件）

1、由于每次编译时，都需要对头文件以及头文件里面包含的头文件进行编译，所以编译时间会很长。而预编译头文件则是将头文件预先编译为二进制文件，如果此后不修改的话，在编译工程的时候就直接用编译好的二进制文件，会大大缩短编译时间。

2、只把那些不太（经常）会被修改的头文件进行预编译，如std，如windows API或者一些其他的库，如GLFW。

3、如果进行预编译头文件，一个例子：

新建一个工程和解决方案，添加Main.cpp,pch.cpp,pch.h三个文件，内容分别如下：

// Main.cpp

#include"pch.h"

int main()

{

std::cout<<"Hello!"<<std::endl;

std::cin.get();

}

// pch.cpp

#include"pch.h"

// pch.h

#pragma once

#include<iostream>

#include<vector>

#include<memory>

#include<string>

#include<thread>

#include<chrono>

#include<unordered\_map>

#include<Windows.h>

在pch.cpp右键，属性-配置属性-C/C++-预编译头-预编译头，里面选择创建, 并在下一行预编译头文件里面添加 pch.h

在项目名称上右键，属性-配置属性-C/C++-预编译头-预编译头，里面选择使用，并在下一行预编译头文件里面添加 pch.h

打开计时工具：工具-选项-项目和解决方案-VC++项目设置-生成计时，就可以看到每次编译的时间

进行对比：

进行预编译头文件前后的首次编译耗时分别为：2634ms和1745ms

进行预编译头文件前后的二次编译（即修改Main.cpp内容后）的耗时分别为：1235ms和312ms

可以看到进行预编译头文件后，时间大大降低

P73 Dynamic Casting

1、dynamic\_cast可以在继承体系里面向上、向下或者平级进行类型转换，自动判断类型，如果转换失败会返回NULL，使用时需要保证是多态，即基类里面含有虚函数。由于dynamic\_cast使用了RTTI（运行时类型识别），所以会对性能增加负担

#include<iostream>

class Base

{

public:

virtual void print(){}

};

class Player : public Base

{

};

class Enemy : public Base

{

};

int main()

{

Player\* player = new Player();

Base\* base = new Base();

Base\* actualEnemy = new Enemy();

Base\* actualPlayer = new Player();

// 旧式转换

Base\* pb1 = player; // 从下往上，是隐式转换，安全

Player\* bp1 = (Player\*)base; // 从上往下，可以用显式转换，危险

Enemy\* pe1 = (Enemy\*)player; // 平级转换，可以用显式转换，危险

// dynamic\_cast

Base\* pb2 = dynamic\_cast<Base\*>(player); // 从下往上，成功转换

Player\* bp2 = dynamic\_cast<Player\*>(base); // 从上往下，返回NULL

if(bp2) { } // 可以判断是否转换成功

Enemy\* pe2 = dynamic\_cast<Enemy\*>(player); // 平级转换，返回NULL

Player\* aep = dynamic\_cast<Player\*>(actualEnemy); // 平级转换，返回NULL

Player\* app = dynamic\_cast<Player\*>(actualPlayer); // 虽然是从上往下，但是实际对象是player，所以成功转换

}

P74 Benchmarking（如何衡量代码性能）

1、用P63里面讲到的Timer来对一个scope进行计时

P75 Structured Binding

1、C++17引入的新特性，可以在将函数返回为tuple、pair、struct等结构时且赋值给另外变量的时候，直接得到成员，而不是结构。（确保在项目属性-C/C++-语言-C++语言标准，里面打开C++17）

#include<iostream>

#include<tuple>

#include<string>

// 此处tuple换成pair或者struct结构也是一样的

std::tuple<std::string, int> CreatePerson()

{

return {"ydc",24};

}

int main()

{

auto[name,age] = CreatePerson();

std::cout<<name<<","<<age<<std::endl;

std::cin.get();

}

P76 std::optional

1、比如在读取文件内容的时候，往往需要判断读取是否成功，常用的方法是传入一个引用变量或者判断返回的std::string是否为空，C++17引入了一个更好的方法，std::optional

#include<iostream>

#include<fstream>

#include<optional>

std::optional<std::string> ReadFileAsString(const std::string& filepath)

{

std::ifstream stream(filepath);

if (stream)

{

std::string result;

//read file

stream.close();

return result;

}

return {};

}

int main()

{

std::optional<std::string> data = ReadFileAsString("data.txt");

// 可以用has\_value()来判断是否读取成功

if (data.has\_value())

{

std::cout<<"File read successfully!\n";

}

else

{

std::cout<<"File not found!\n";

}

// 也可以用value\_or()来判断是否读取成功

std::string result = data.value\_or("Not resprent");

std::cout<<result<<std::endl;

std::cin.get();

}

P77 std::variant

1、C++17引入一种可以容纳多种类型变量的结构，std::variant

#include<iostream>

#include<variant>

int main()

{

std::variant<std::string,int> data; // <>里面的类型不能重复

data = "ydc";

// 索引的第一种方式：std::get，但是要与上一次赋值类型相同，不然会报错

std::cout<<std::get<std::string>(data)<<std::endl;

// 索引的第二种方式，std::get\_if，传入地址，返回为指针

if (auto value = std::get\_if<std::string>(&data))

{

std::string& v = \*value;

}

data = 2;

std::cout<<std::get<int>(data)<<std::endl;

std::cin.get();

}

2、std::variant的大小是<>里面的大小之和，与union不一样，union的大小是类型的大小最大值

P78 std::any

1、也是C++17引入的可以存储多种类型变量的结构，其本质是一个union，但是不像std::variant那样需要列出类型

#include<iostream>

#include<any>

// 此处写一个new的函数，是为了断点，看主函数里面哪里调用了new，来看其堆栈

void\* operator new(size\_t size)

{

return malloc(size);

}

int main()

{

std::any data;

data = 2;

data = std::string("ydc");

std::string& s = std::any\_cast<std::string&>(data);

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

std::cin.get();

}

P79 std::async

1、std::async封装了异步编程的操作，具体没弄怎么明白，参考以下链接吧：

<https://en.cppreference.com/w/cpp/thread/async>

P80 如何让string运行更快

1、一种调试在heap上分配内存的方法，自己写一个new的方法，然后设置断点或者打出log，就可以知道每次分配了多少内存，以及分配了几次：

#include<iostream>

#include<string>

static uint32\_t s\_AllocCount = 0;

void\* operator new(size\_t size)

{

s\_AllocCount++;

std::cout<<"Allocing: "<<size<<" bytes\n";

return malloc(size);

}

void PrintName(const std::string& name)

{

std::cout<<name<<std::endl;

}

int main()

{

std::string fullName = "yang dingchao";

std::string firstName = fullName.substr(0,4);

std::string lastName = fullName.substr(5,8);

PrintName(firstName);

PrintName(lastName);

std::cout<<s\_AllocCount<<" allocations\n";

std::cin.get();

}

Allocing: 8 bytes

Allocing: 8 bytes

Allocing: 8 bytes

yang

dingchao

3 allocations

这个程序仅仅是从一个string取子字符串，就多分配了两次内存，下面来改进它

2、用C++17引入的std::string\_view来对同一块内存的string进行截取

#include<iostream>

#include<string>

static uint32\_t s\_AllocCount = 0;

void\* operator new(size\_t size)

{

s\_AllocCount++;

std::cout<<"Allocing: "<<size<<" bytes\n";

return malloc(size);

}

void PrintName(std::string\_view name)

{

std::cout<<name<<std::endl;

}

int main()

{

std::string fullName = "yang dingchao";

std::string\_view firstName(fullName.c\_str(),4);

std::string\_view lastName(fullName.c\_str()+5,8);

PrintName(firstName);

PrintName(lastName);

std::cout<<s\_AllocCount<<" allocations\n";

std::cin.get();

}

输出如下：

Allocing: 8 bytes

yang

dingchao

1 allocations

3、上面的程序还是有一次分配，如果把std::string改成const char\*，就变成了0次分配：

#include<iostream>

#include<string>

static uint32\_t s\_AllocCount = 0;

void\* operator new(size\_t size)

{

s\_AllocCount++;

std::cout<<"Allocing: "<<size<<" bytes\n";

return malloc(size);

}

void PrintName(std::string\_view name)

{

std::cout<<name<<std::endl;

}

int main()

{

const char\* fullName = "yang dingchao";

std::string\_view firstName(fullName,4);

std::string\_view lastName(fullName+5,8);

PrintName(firstName);

PrintName(lastName);

std::cout<<s\_AllocCount<<" allocations\n";

std::cin.get();

}

输出如下：

yang

dingchao

0 allocations

P81 使用Chrome://tracing工具来可视化性能

1、工作原理：通过在一个作用域添加一个点，采集这个作用域的耗时（类似于Timer），随后将这些信息生成json文件，然后拖入Chrome://tracing这个界面，即可可视化benchmark，详见以下链接：

<https://www.cnblogs.com/yangxunwu1992/p/14020364.html>

P82 Singleton单例

1、Singleton只允许被实例化一次，用于组织一系列全局的函数或者变量，与namespace很像。例子：随机数产生的类、渲染器类。

#include<iostream>

class Singleton

{

public:

Singleton(const Singleton&) = delete; // 删除拷贝复制函数

static Singleton& Get() // 通过Get函数来获取唯一的一个实例，其定义为static也是为了能直接用类名调用

{

return s\_Instance;

}

void Function(){} // 执行功能的函数

private:

Singleton(){} // 不能让别人实例化，所以要把构造函数放进private

static Singleton s\_Instance; // 定义为static，让其唯一

};

Singleton Singleton::s\_Instance; // 唯一的实例化的地方

int main()

{

Singleton::Get().Function();

}

2、具体的一个简单的随机数类的例子：

#include<iostream>

class Random

{

public:

Random(const Random&) = delete; // 删除拷贝复制函数

static Random& Get() // 通过Get函数来获取唯一的一个实例

{

static Random instance; // 在此处实例化一次

return instance;

}

static float Float(){ return Get().IFloat();} // 调用内部函数,可用类名调用

private:

float IFloat() { return m\_RandomGenerator; } // 将函数的实现放进private

Random(){} // 不能让别人实例化，所以要把构造函数放进private

float m\_RandomGenerator = 0.5f;

};

// 与namespace很像

namespace RandomClass {

static float s\_RandomGenerator = 0.5f;

static float Float(){return s\_RandomGenerator;}

}

int main()

{

float randomNum = Random::Float();

std::cout<<randomNum<<std::endl;

std::cin.get();

}

P83 使用小的string

1、在release模式下面，使用size小于16的string，不会分配内存，而大于等于16的string，则会分配32bytes内存以及更多，所以16个字符是一个分界线

#include<iostream>

void\* operator new(size\_t size)

{

std::cout<<"Allocated: "<<size<<" bytes\n";

return malloc(size);

}

int main()

{

std::string longName = "ydc ydc ydc ydc ydc";

std::string shortName = "ydc";

std::cin.get();

}

Release模式，只有longName在heap上面分配内存了，输出如下：

Allocated: 32 bytes

P84 跟踪内存分配的简易办法

1. 重写new和delete操作符函数，并在里面打印分配和释放了多少内存，也可在重载的这两个函数里面设置断点，通过查看调用栈即可知道什么地方分配或者释放了内存

#include<iostream>

void\* operator new(size\_t size)

{

std::cout<<"Allocing "<<size<<" bytes\n";

return malloc(size);

}

void operator delete(void\* memory, size\_t size)

{

std::cout<<"Free "<<size<<" bytes\n";

free(memory);

}

struct Entity

{

int x,y,z;

};

int main()

{

{

std::string name = "ydc";

}

Entity\* e = new Entity();

delete e;

std::cin.get();

}

1. 还可以写一个简单统计内存分配的类，在每次new的时候统计分配内存，在每次delete时统计释放内存，可计算出已经分配的总内存：

#include<iostream>

struct AllocationMertics

{

uint32\_t TotalAllocated = 0;

uint32\_t TotalFreed = 0;

uint32\_t CurrentUsage() {return TotalAllocated - TotalFreed;}

};

static AllocationMertics s\_AllocationMetrics;

void\* operator new(size\_t size)

{

s\_AllocationMetrics.TotalAllocated+=size;

return malloc(size);

}

void operator delete(void\* memory, size\_t size)

{

s\_AllocationMetrics.TotalFreed += size;

free(memory);

}

static void PrintMemoryUsage()

{

std::cout<<"Memory usage: "<<s\_AllocationMetrics.CurrentUsage()<<" bytes\n";

}

int main()

{

PrintMemoryUsage();

{

std::string name = "ydc";

PrintMemoryUsage();

}

PrintMemoryUsage();

std::cin.get();

}

P85 lvalue and rvalue（左值和右值）

1. 左值：有存储空间的值，往往长期存在；右值：没有存储空间的短暂存在的值
2. 一般而言，赋值符号=左边的是左值，右边的是右值
3. 在给函数形参列表传参时，有四种情况：

#include<iostream>

void PrintName(std::string name) // 可接受左值和右值

{

std::cout<<name<<std::endl;

}

void PrintName(std::string& name) // 只接受左值引用，不接受右值

{

std::cout << name << std::endl;

}

void PrintName(const std::string& name) // 接受左值和右值，把右值当作const lvalue&

{

std::cout << name << std::endl;

}

void PrintName(std::string&& name) // 接受右值引用

{

std::cout << name << std::endl;

}

int main()

{

std::string firstName = "yang";

std::string lastName = "dingchao";

std::string fullName = firstName + lastName;

PrintName(fullName);

PrintName(firstName+lastName);

std::cin.get();

}

P86 CI（continuous integration）

1、 CI(Continuous integration，中文意思是持续集成)是一种软件开发时间。持续集成强调开发人员提交了新代码之后，立刻进行构建、（单元）测试。根据测试结果，我们可以确定新代码和原有代码能否正确地集成在一起。

2、主要讲解如何在linode租一个服务器，来运行Jenkins

P87 static analyze

1、主要讲了一个工具PVS-studio的用法，可以static analyze代码

P88 Argument Evaluation Order

1、讲了一个undefined behavior的例子：

#include<iostream>

void PrintSum(int a, int b)

{

std::cout<<a<<"+"<<b<<"="<<a+b<<std::endl;

}

int main()

{

int value = 0;

PrintSum(value++,++value);

std::cin.get();

}

类似这样在传参时使用++，这种行为是不确定的，在不同编译器不同语言版本和配置下，其行为不一致，所以严禁这样使用

P89 move semantics

1、比如一个类Entity含有一个成员Name为String类型，如果要用常量字符串来初始化这个类，就会先调用String的构造函数，再调用String的拷贝构造函数（经Entity构造函数里面调用），然后再调用String的析构函数，但是使用move操作就可以让中间的一次拷贝变成move，就可以少一次new，我理解为浅拷贝的意思：

#include<iostream>

class String

{

public:

String() = default;

String(const char\* string) //构造函数

{

printf("Created\n");

m\_Size = strlen(string);

m\_Data = new char[m\_Size];

memcpy(m\_Data,string,m\_Size);

}

String(const String& other) // 拷贝构造函数

{

printf("Copied\n");

m\_Size = other.m\_Size;

m\_Data = new char[m\_Size];

memcpy(m\_Data,other.m\_Data,m\_Size);

}

String(String&& other) noexcept // 右值引用拷贝，相当于移动，就是把复制一次指针，原来的指针给nullptr

{

printf("Moved\n");

m\_Size = other.m\_Size;

m\_Data = other.m\_Data;

other.m\_Size = 0;

other.m\_Data = nullptr;

}

~String()

{

printf("Destroyed\n");

delete m\_Data;

}

private:

uint32\_t m\_Size;

char\* m\_Data;

};

class Entity

{

public:

Entity(const String& name) : m\_Name(name)

{

}

Entity(String&& name) : m\_Name(std::move(name)) // std::move(name)也可以换成(String&&)name

{

}

private:

String m\_Name;

};

int main()

{

Entity entity("ydc");

std::cin.get();

}

如此的代码，在实例化entity的时候，如果传入的是字符串常量（右值），则会调用拷贝的右值版本，避免了一次new，如果传入的是String（左值），则仍然会进行一次左值拷贝

P90 std::move

1、使用std::move可以将本来的copy操作变为move操作：

#include<iostream>

class String

{

public:

String() = default;

String(const char\* string)

{

printf("Created\n");

m\_Size = strlen(string);

m\_Data = new char[m\_Size];

memcpy(m\_Data,string,m\_Size);

}

String(const String& other)

{

printf("Copied\n");

m\_Size = other.m\_Size;

m\_Data = new char[m\_Size];

memcpy(m\_Data,other.m\_Data,m\_Size);

}

String& operator=(const String& other)

{

printf("Cpoy Assigned\n");

delete [] m\_Data;

m\_Size = other.m\_Size;

m\_Data = new char[m\_Size];

memcpy(m\_Data, other.m\_Data, m\_Size);

return \*this;

}

String(String&& other) noexcept

{

printf("Moved\n");

m\_Size = other.m\_Size;

m\_Data = other.m\_Data;

other.m\_Size = 0;

other.m\_Data = nullptr;

}

String& operator=(String&& other) noexcept

{

printf("Move Assigned\n");

if(this != &other)

{

delete [] m\_Data;

m\_Size = other.m\_Size;

m\_Data = other.m\_Data;

other.m\_Size = 0;

other.m\_Data = nullptr;

}

return \*this;

}

~String()

{

printf("Destroyed\n");

delete m\_Data;

}

private:

uint32\_t m\_Size;

char\* m\_Data;

};

int main()

{

String name = "ydc"; // String name("ydc");调用构造函数

String nameCopy = name; // String nameCopy(name);调用拷贝构造函数

String nameAssign;

nameAssign = name; // 调用拷贝赋值函数

String nameMove = std::move(name); // String nameMove(std::move(name));调用右值引用构造函数

String nameMoveAssign;

nameMoveAssign = std::move(name); // 调用右值引用赋值函数

std::cin.get();

}

输出如下：

Created

Copied

Cpoy Assigned

Moved

Move Assigned

P91 自己实现一个简单的Array类

1、代码如下：

#include<iostream>

template<typename T,size\_t S>

class Array

{

public:

constexpr int Size() const {return S;} // const放在成员函数后面，表示函数不能修改值；用constexpr来修饰表示返回值是常量字面值，可以被编译器优化

T& operator[](size\_t index) {return m\_Data[index]; } // 返回引用以对原数据进行修改

const T& operator[](size\_t index) const {return m\_Data[index]; }

T\* Data(){return m\_Data;} // 返回数组本身，实际上是个指针，其地址等价于&m\_Data[0]

const T\* Data() const {return m\_Data;}

private:

T m\_Data[S];

};

int main()

{

Array<int,5> data;

memset(&data[0],0,data.Size()\*sizeof(int));

data[2] = 2;

for(size\_t i = 0;i<data.Size();i++)

std::cout<<data[i]<<std::endl;

std::cin.get();

}

P92 自己实现一个简单的Vector类

1、代码如下：

template<typename T>

class Vector

{

public:

Vector()

{

// Alloc 2 memory

ReAlloc(2); // 初始化构造的时候默认分类两个内存

}

~Vector()

{

delete [] m\_Data; // 此处写的不完整，还需要释放每个元素的内存

}

void PushBack(T&& value) // 右值版本，用move

{

if(m\_Size >= m\_Capacity) // 发现当前分配内存快要满的时候，再多分配一点内存

ReAlloc(m\_Capacity + m\_Capacity/2);

m\_Data[m\_Size] = std::move(value);

m\_Size++;

}

void PushBack(const T& value) // 左值版本，用copy

{

if (m\_Size >= m\_Capacity)

ReAlloc(m\_Capacity + m\_Capacity / 2);

m\_Data[m\_Size] = value;

m\_Size++;

}

template<typename... Args>

T& EmplaceBack(Args&&... args) // Emplace与Push的区别在于，可以在里面调元素类的构造函数

{

if(m\_Size>= m\_Capacity)

ReAlloc(m\_Capacity+m\_Capacity/2);

new(&m\_Data[m\_Size]) T(std::forward<Args>(args)...); // 这句话看不太懂

// m\_Data[m\_Size] = T(std::forward<Args>(args)...); //可替代上一句话

return m\_Data[m\_Size++];

}

void PopBack()

{

if (m\_Size > 0)

{

m\_Size--;

m\_Data[m\_Size].~T(); // 弹出末尾元素，并且释放内存

}

}

void Clear()

{

for(size\_t i = 0;i<m\_Size;i++)

m\_Data[i].~T();

m\_Size = 0;

}

size\_t Size() const { return m\_Size;}

const T& operator[] (size\_t index) const { return m\_Data[index]; }

T& operator[] (size\_t index) { return m\_Data[index]; }

private:

void ReAlloc(size\_t newCapacity)

{

// 1. alloc a new block of memory

// 2. copy/move old elements to new block of memory

// 3. delete

T\* newBlock = new T[newCapacity];

if(m\_Size > newCapacity)

m\_Size = newCapacity;

for(size\_t i = 0;i<m\_Size;i++)

newBlock[i] = std::move(m\_Data[i]); // 使用move的方式，避免copy

delete[] m\_Data;

m\_Data = newBlock;

m\_Capacity = newCapacity;

}

T\* m\_Data = nullptr;

size\_t m\_Size;

size\_t m\_Capacity;

};

P93 iterator

1、iterator可用于有序或者无序结构的遍历，如：

#include<iostream>

#include<unordered\_map>

int main()

{

typedef std::unordered\_map<std::string,int> ScoreMap;

ScoreMap map;

map["ydc"] = 1;

map["tt"] = 2;

for (ScoreMap::iterator it = map.begin(); it != map.end(); it++)

{

auto& key = it->first;

auto& value = it->second;

std::cout<<key<<"="<<value<<std::endl;

}

for (auto& kv : map) // C++11引入的范围for循环

{

auto& key = kv.first;

auto& value = kv.second;

std::cout << key << "=" << value << std::endl;

}

for(auto& [key,value]:map) // C++17引入的structure binding

std::cout << key << "=" << value << std::endl;

std::cin.get();

}

P94 自己写一个iterator类

1、主要是针对vector写一个iterator类，成员是一个指针，主要在iterator类里实现++,--,!=,[],->等操作符，以及在Vector类里面实现begin(),end()等成员函数：

#include<iostream>

template<typename Vector>

class VectorIterator

{

public:

using ValueType = typename Vector::ValueType; // 此处要加上typename指出后面是一个类型名，以免引起歧义

using PointerType = ValueType\*;// 这里可加可不加typename

using ReferenceType = ValueType&;

public:

VectorIterator(PointerType ptr):m\_Ptr(ptr) {}

VectorIterator operator++()

{

m\_Ptr++;

return \*this;

}

VectorIterator operator++(int) // 这里看不懂

{

VectorIterator iterator = \*this;

++(\*this);

return iterator;

}

VectorIterator operator--()

{

m\_Ptr--;

return \*this;

}

VectorIterator operator--(int) // 同++，也看不懂

{

VectorIterator iterator = \*this;

--(\*this);

return iterator;

}

ReferenceType operator[](int index)

{

return \*(m\_Ptr+index);

}

PointerType operator->() // 不知道这里为啥返回一个指针

{

return m\_Ptr;

}

ReferenceType operator\*()

{

return \*m\_Ptr;

}

bool operator==(const VectorIterator& other) const

{

return m\_Ptr == other.m\_Ptr;

}

bool operator!=(const VectorIterator& other) const

{

return !(\*this==other);

}

private:

PointerType m\_Ptr; // 迭代器的一个主要成员就是一个指针

};

template<typename T>

class Vector

{

public:

using ValueType = T; // 为了将此类型传给VectorIterator使用，故命名

using Iterator = VectorIterator<Vector<T>>; // 用于begin和end函数返回类型

public:

Vector()

{

// Alloc 2 memory

ReAlloc(2); // 初始化构造的时候默认分类两个内存

}

~Vector()

{

delete[] m\_Data; // 此处写的不完整，还需要释放每个元素的内存

}

void PushBack(T&& value) // 右值版本，用move

{

if (m\_Size >= m\_Capacity) // 发现当前分配内存快要满的时候，再多分配一点内存

ReAlloc(m\_Capacity + m\_Capacity / 2);

m\_Data[m\_Size] = std::move(value);

m\_Size++;

}

void PushBack(const T& value) // 左值版本，用copy

{

if (m\_Size >= m\_Capacity)

ReAlloc(m\_Capacity + m\_Capacity / 2);

m\_Data[m\_Size] = value;

m\_Size++;

}

template<typename... Args>

T& EmplaceBack(Args&&... args) // Emplace与Push的区别在于，可以在里面调元素类的构造函数

{

if (m\_Size >= m\_Capacity)

ReAlloc(m\_Capacity + m\_Capacity / 2);

new(&m\_Data[m\_Size]) T(std::forward<Args>(args)...); // 这句话看不太懂

// m\_Data[m\_Size] = T(std::forward<Args>(args)...); //可替代上一句话

return m\_Data[m\_Size++];

}

void PopBack()

{

if (m\_Size > 0)

{

m\_Size--;

m\_Data[m\_Size].~T(); // 弹出末尾元素，并且释放内存

}

}

void Clear()

{

for (size\_t i = 0; i < m\_Size; i++)

m\_Data[i].~T();

m\_Size = 0;

}

Iterator begin()

{

return Iterator(m\_Data);

}

Iterator end()

{

return Iterator(m\_Data+m\_Size);

}

size\_t Size() const { return m\_Size; }

const T& operator[] (size\_t index) const { return m\_Data[index]; }

T& operator[] (size\_t index) { return m\_Data[index]; }

private:

void ReAlloc(size\_t newCapacity)

{

// 1. alloc a new block of memory

// 2. copy/move old elements to new block of memory

// 3. delete

T\* newBlock = new T[newCapacity];

if (m\_Size > newCapacity)

m\_Size = newCapacity;

for (size\_t i = 0; i < m\_Size; i++)

newBlock[i] = std::move(m\_Data[i]); // 使用move的方式，避免copy

delete[] m\_Data;

m\_Data = newBlock;

m\_Capacity = newCapacity;

}

T\* m\_Data = nullptr;

size\_t m\_Size;

size\_t m\_Capacity;

};

int main()

{

Vector<std::string> data;

data.EmplaceBack("ydc");

data.EmplaceBack("tt");

for (Vector<std::string>::Iterator it = data.begin(); it != data.end(); it++)

{

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

}

for (auto& v : data)

{

std::cout<<v<<std::endl;

}

std::cin.get();

}

P95 如何真正学好C++

看代码！

看代码！

看代码！