C++ 学习笔记

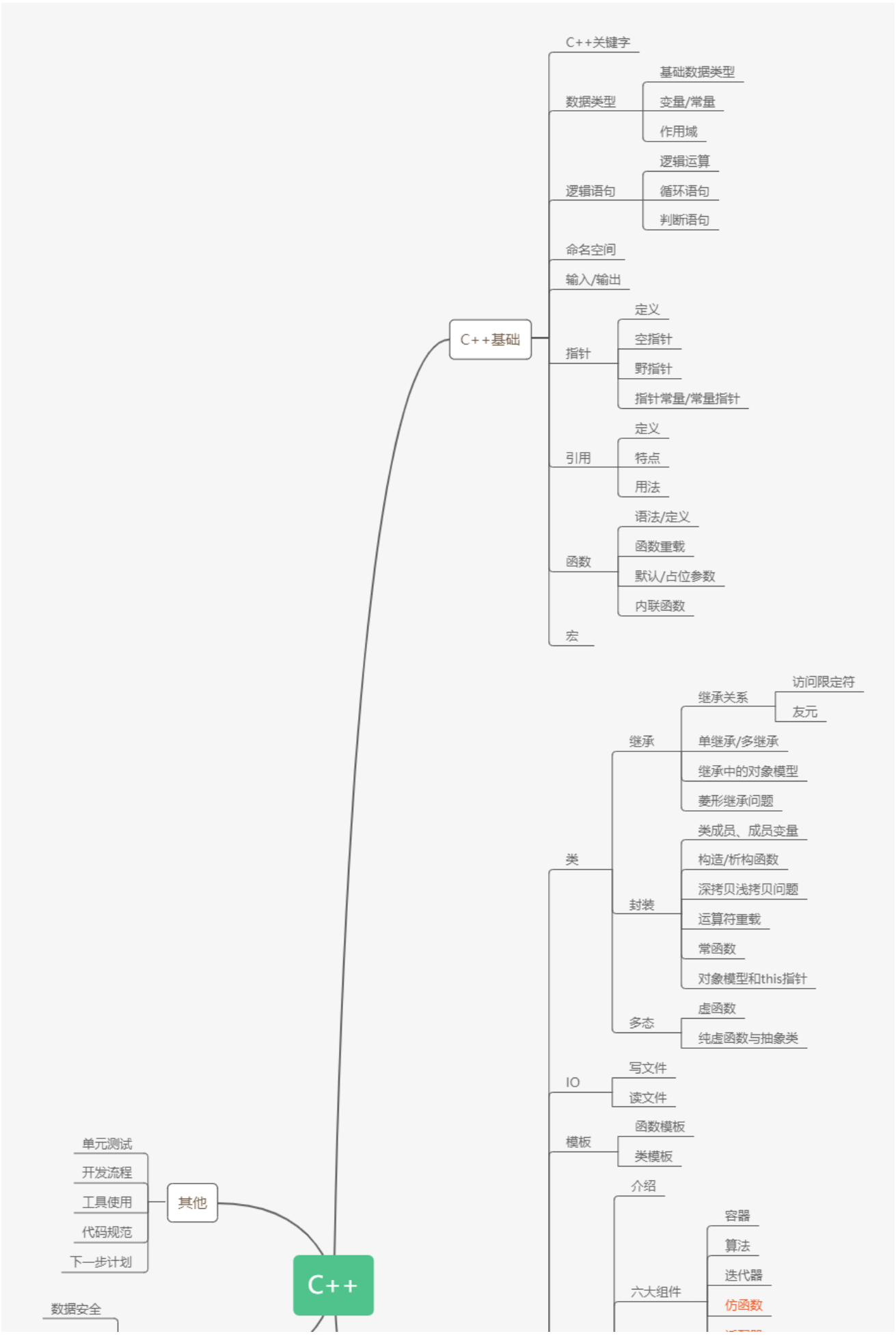
前言

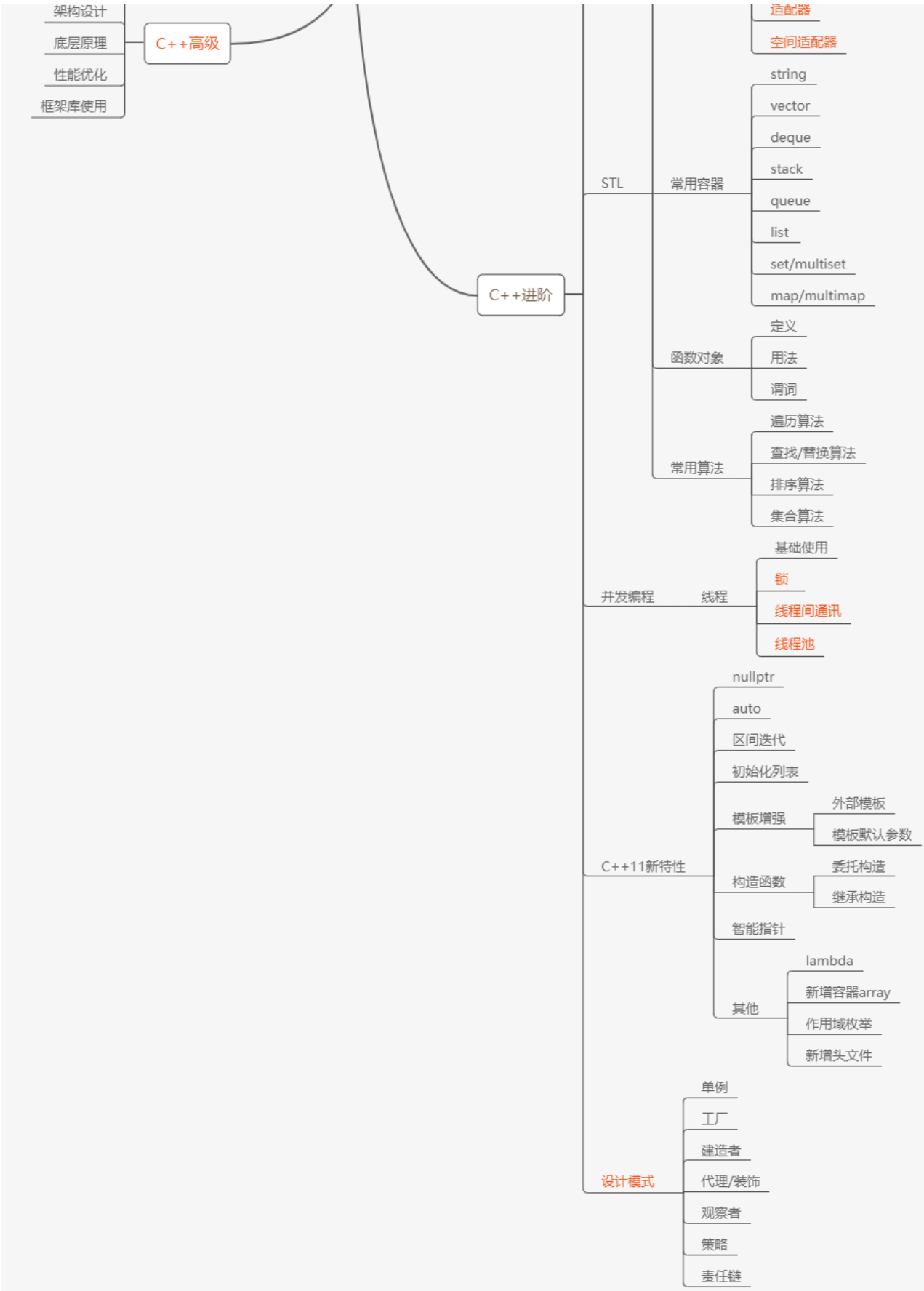
学习资料：

- C++学习视频 链接: https://pan.baidu.com/s/1TB3XvyRnkZMnAFBlVNP9Lw 提取码: wrx9；
- 《C++ primer plus》；
- 其他开发者分享的学习笔记，具体链接在下面各章节中；

学习中测试的代码维护在：git@gitlab3qa.jqdev.saic-gm.com:Picher/SelfSpace.git

知识体系


C++
C++基础
C++关键字
数据类型
基础数据类型
变量/常量
作用域
逻辑语句
逻辑运算
循环语句
判断语句
命名空间
输入/输出
指针
定义
空指针
野指针
指针常量/常量指针
引用
定义
特点
用法
函数
语法/定义
函数重载
默认/占位参数
内联函数
宏
类
继承
继承关系
访问限定符
友元
单继承/多继承
继承中的对象模型
菱形继承问题
封装
类成员、成员变量
构造/析构函数
深拷贝浅拷贝问题
运算符重载
常函数
对象模型和this指针
多态
虚函数
纯虚函数与抽象类
IO
写文件
读文件
模板
函数模板
类模板
介绍
六大组件
容器
算法
迭代器
仿函数
其他
单元测试
开发流程
工具使用
代码规范
下一步计划
数据安全


架构设计
底层原理
性能优化
框架库使用
C++高级
C++进阶
STL
常用容器
string
vector
deque
stack
queue
list
set/multiset
map/multimap
适配器
空间适配器
函数对象
定义
用法
谓词
常用算法
遍历算法
查找/替换算法
排序算法
集合算法
并发编程
线程
基础使用
锁
线程间通讯
线程池
C++11新特性
nullptr
auto
区间迭代
初始化列表
模板增强
外部模板
模板默认参数
构造函数
委托构造
继承构造
智能指针
其他
lambda
新增容器array
作用域枚举
新增头文件
设计模式
单例
工厂
建造者
代理/装饰
观察者
策略
责任链

C++关键字

保留字(关键字)						
C++系统中预定义的、在语言或编译系统的实现中具有特殊含义的单词：						
if	else	while	signed	throw	union	this
int	char	double	unsigned	const	goto	virtual
for	float	break	auto	class	operator	case
do	long	typedef	static	friend	template	default
new	void	register	extern	return	enum	inline
try	short	continue	sizeof	switch	private	protected
asm	while	catch	delete	public	volatile	struct

数据类型

基本数据类型

bool，布尔型，1 个字节

char，字符型，1 个字节

int，整型，4个字节

float，浮点型，4 个字节

double，双浮点型，8 个字节

wchar_t，宽字符型，2 个字节

变量

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

常量

`#define LENGTH 10`

`const int LENGTH = 10;`

逻辑语句

运算符

同Java

算数运算符： +、-、*、/、%、++、--

关系运算符：==、!=、<、>、<=、>=

逻辑运算符：&&、||、!

位运算符：&、|、^、~、<<、>>

赋值运算符：=、+=、-=、*=、/=、%=、<<=、>>=...

其他运算符：sizeof 返回变量大小、&返回地址、*指向一个变量

循环

同Java

while 循环

```
int a = 10;
while( a < 20 )
   {
       cout << "a 的值: " << a << endl;
       a++;
   }

   // do 循环执行
   do
   {
       cout << "a 的值: " << a << endl;
       a = a + 1;
   }while( a < 20 );
```

for 循环

```
for (int i = 1; i <= 9; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            cout << i << "*" << j << "=" << i * j<<" ";
        }
        cout << endl;
    }
```

数组

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  初始化索引号为 0 的行 */
 {4, 5, 6, 7} ,   /*  初始化索引号为 1 的行 */
 {8, 9, 10, 11}   /*  初始化索引号为 2 的行 */
};
```

```
    // 冒泡
    int arr[] = { 1,2,3,6,2,1,4,3,8,3,1,2,5,7 };
    int length = sizeof(arr) / sizeof(arr[0]);
    int temp;
    for (int j = 0;j < length - 1 ;j++)
    {
        for (int i = 0; i < length - 1 - j; i++)
        {
            if (arr[i] > arr[i + 1]) {
                temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
            }
        }
    }

    for (int i=0;i<length;i++)
    {
        cout << arr[i] << endl;
    }
```

指针

- 指针前加*表示解引用
- 常量指针,指针指向可以修改，值不可修改
- 指针常量，值可以改，指向不可改

```
int a = 10;
int* p = &a;
cout << &a << endl; // -->00D3F82C
cout << p << endl; // -->00D3F82C
cout << sizeof(p) << endl;// --> 4  32位系统里4字节，64位系统8字节
// 指针前加*表示解引用
*p = 1000;
cout << a << endl; // --> 1000
cout << *p << endl; // --> 1000
```

```
    //常量指针,指针指向可以修改，值不可修改
    int a = 10;
    int b = 20;
    const int* p = &a;
    p = &b;
    //*p = 100; //error
```

```
    //指针常量，值可以改，指向不可改
    int* const p1 = &a;
    *p1 = 100;
    //p1 = &b; //error

    const int* const p2 = &a;
    //p2 = 100; //error
    p2 = &b;
```

命名空间

```
#include <iostream>
using namespace std;

int main()
{

    system("pause");
    return 0;
}
```

- 是一种作用域，可防止命名冲突。
- 为了和C区分开，头文件不适用后缀.h

语法

```
namespace namespace_name {
   // 代码声明
}
```

命名空间定义可嵌套

```
namespace namespace_name1 {
   // 代码声明
   namespace namespace_name2 {
      // 代码声明
   }
}
```

可以使用using来引用命名空间。

```
namespace nameSpaceA
{
    int age = 10;
}

namespace nameSpaceB
{
    int age = 20;
    namespace nameSpaceC
```

```
    {
        struct Teacher
        {
            int age = 30;
        };
    }
}
...

    using namespace nameSpaceA;
    using namespace nameSpaceB;

    // 命名空间成员名重复，需要加上作用域来访问
    cout << nameSpaceA::age << endl;

    //使用作用域或者using 引入命名空间来访问成员
    /*nameSpaceB::nameSpaceC::Teacher t;
    cout << t.age << endl;*/

    using nameSpaceC::Teacher;
    Teacher t;
    cout << t.age << endl;
```

#include和namespace的区别

- #include某个头文件，是为了引用其他文件中的内容
- using namespace 是为了代码整洁，降低命名冲突

https://blog.csdn.net/u013719339/article/details/80221899

函数

函数的默认参数

1. 若函数形参有默认参数，则从这个参数往后，都需要有默认值
2. 若函数申明有默认参数，函数实现就不能有默认参数

```
//Error sample
int func(int a = 10, int b = 20);
int func(int a = 1, int b = 2)
{
    return a + b;
}
```

函数的占位参数

```
int func(int a, int b,int )
{
    return a + b;
}
```

函数的重载

函数重载引用，无const 的引用形参无法直接传具体值，因为int& a本质上是int* contst a; 作用：函数名可以相同，提高复用性 条件：

1. 同一作用域下
2. 函数名相同
3. 函数参数类型不同、个数不同、顺序不同

```
void func1(int a) {}
void func1(double a) {}
void func1(int a, double b){}
void func1(double a, int b){}
```

函数返回值不同不能作为重载条件 引用可以作为重载条件：

```
// int a = 10;func1(a); 会走该方法
void func1(int& a)
{
    cout << "func1 not const" << endl;
}

// 直接func1(10);会走该方法 const int &a = 10;合法
void func1(const int& a)
{
    cout << "func1 const" << endl;
}
```

函数重载碰到默认参数，func(10) 会有二义性问题

```
int func(int a, double d = 10)
{
    return a + d;
}

int func(int a)
{
    return a;
}
```

内联函数

语法

```
inline 函数定义
//例如
inline int Max (int a, int b)
{
    if(a >b)
        return a;
    return b;
}
```

内联函数和普通函数的区别在于：当编译器处理调用内联函数的语句时，不会将该语句编译成函数调用的指令，而是直接将整个函数体的代码插人调用语句处，就像整个函数体在调用处被重写了一遍一样。**但会增加代码体积，以空间换时间。**

- 建议内联函数定义放在头文件中
- 只适合涵数体内代码简单的涵数使用
- 不要包含循环，函数本身不要递归
- 放在函数声明前面无效，需放在定义前

结构体

```
struct Student
{
    string name;
    int age;
    int score;
};

//值传递
void printStu(struct Student *s)
{
    cout << s->name << endl;
    s->name = "fix";
}

//指针传递
void printStu(struct Student s)
{
    cout << s.name << endl;
    s.name = "fix";
}
```

内存分区模型

- 代码区：存放函数的二进制代码，由操作系统统一管理
- 全局区：存放全局变量以及常量，程序结束由系统管理释放
- 栈区:由编译器自动分配释放，存放局部变量，形参等
- 堆区：由程序员分配和释放，若不释放，程序结束后由系统释放
- 不同的内存区域，控制数据的生命周期； ***代码区，全局区在程序运行之前划分；不要返回局部变量地址；***

引用

- 给变量起别名，本质上是指针常量，int* const p;

```
c++
int a = 10;
int& a1 = a; //int* const a1 = &a;
a1 = 20;
```

1. 引用必须初始化
2. 引用一旦初始化后就不可更改
3. 若函数值是一个引用，则这个函数调用可以作为左值

```
{
    int a = 10;
    return a;
}
...
int &a = funcReturnRef();
cout << a << endl;// 10
funcReturnRef() = 1000;
cout << a << endl;// 1000
```

常量引用，修饰形参，防止误操作

宏

- 将一个标识符定义为一个字符串
- 宏名一般用大写
- 宏定义末尾不加分号；

```
#define <宏名>    <字符串>
//例如:
#define _PI_ 3.1415926
```

NULL的宏定义示例

```
#undef NULL
#if defined(__cplusplus)
#define NULL 0
#else
#define NULL ((void*)0)
#endif
```

C++中的预定义宏

- **LINE** //这会在程序编译时包含当前行号。 **FILE** //这会在程序编译时包含当前文件名。 **DATE** //把源文件转换为目标代码的日期，month/day/year **TIME** //返回程序被编译的时间，hour:minute:second 。

https://blog.csdn.net/cn_wk/article/details/55215730

类和对象

基本语法：

```
class Student
{
    string name;
    string sId;
public:
    Student(string n, string s)
    {
        name = n;
        sId = s;
    };

void print()
{
    cout << name << "," << sId << endl;
}
};
```

访问权限：public,protected,private 类似java。 struct 默认权限为public，class 默认权限为private。

```
class Person
{
public:
    Person()
    {
        cout << "person" << endl;
    }
    // 析构
    ~Person()
    {
        cout << "~person" << endl;
    }
};
```

- 拷贝构造函数

```
 //拷贝构造
 Person(const Person &p)
    {
        age = p.age;
    }
    // 值传递给一个函数参数传值，值方式返回局部对象
```

封装

构造函数调用规则

调用方式：括号、显示、隐式转换

```
Person p;
person p1(10);
Person p2 = Person(10);
Person p3 = 10;// Person p = Person(10)
// 匿名对象
// 当前执行结束后，立刻销毁
// 不要用拷贝构造函数 初始化匿名对象
Person(p3);//Error
```

C++拷贝构造函数调用时机

- 使用已经创建好的对象初始化一个新对象
- 值传递方式给一个函数参数传值
- 以值的方式返回局部对象

构造函数创建规则 默认情况C++编译器个一个类至少有添加三个函数：默认构造，拷贝函数，析构函数

- 若定义有参构造函数，则编译器不提供无参构造，但会提供拷贝构造
- 若定义了拷贝构造函数，则编译器不提供其他构造函数

深拷贝与浅拷贝

浅拷贝：编译器默认提供给的拷贝构造函数，主要做了赋值操作 出现问题：指针指向同一块内存地址，堆区内存重复释放

深拷贝：重新在堆区申请空间 解决浅拷贝问题：自己实现拷贝构造，给指针变量重新申请空间

C++对象模型和this指针

- 成员变量与成员函数分开存储
- 空对象大小为1
- 静态成员变量、成员方法不属于类对象上，不计入类大小

this指针 本质是指针常量，不可修改指针的指向

```
class Person
{
    string m_Name;

public:
    void printName()
    {
        cout << m_Name << endl;
    }

    void printAny()
    {
        cout << "Any" << endl;
    }
};
...
Person* p1 = NULL;
p1->printAny();
p1->printName();// ERROR
```

常函数

```
class Person
{
    string m_Name;

public:

    // 常函数内部不可修改成员属性，若加mutable 修饰则可修改
    void func() const
    {
        //this = NULL;//ERROR "this" is "Person* contst this" ;
        //this->m_Name = "11";// ERROR 常函数 is "const Person* contst this"
    }
};

const Person p;// 常对象只能调用常函数
```

友元

访问类得私有成员

```
class Building
{
    friend void goodGay(Building& b);// 全局函数做友元
    friend class GoodGay;// 类做友元

public :
    string m_CustomerRoom;
    Building()
    {
        m_CustomerRoom = "客厅";
        m_BedRoom = "卧室";
    }
private:
    string m_BedRoom;
};
class GoodGay
{
public :
    GoodGay();
    void visit();
private :
    Building* b;
};

GoodGay::GoodGay()
{
    b = new Building;
}
```

```
void GoodGay::visit()
{
    cout << b->m_BedRoom<<endl;
}
```

运算符重载

operator+ operator()

```
class MyInteger
{
public:
    int m_Age;
    MyInteger operator+ (MyInteger &a) {
        this->m_Age += a.m_Age;
        return *this;
    }
};
...
    MyInteger a1;
    MyInteger a2;
    MyInteger a3 = a1 + a2;
```

类的继承

```
class Dog : Anim
{

};
```

- public 继承，继承父类public为public
- protected 继承，继承父类public为protected
- private 继承，继承父类public为private

```
class Anim
{
public:
    int m_A;
protected:
    int m_B;
private:
    int m_C;
};

class Dog :protected Anim
{
    int m_D;
};
...
Dog d;
```

```
cout << sizeof(d) << endl;// 16
```

继承中的对象模型

查看对象模型语句：`cl /d1 reportSingleClassLayoutDog main.cpp`

```
class Dog       size(16):
        +---
 0      | +--- (base class Anim)
 0      | | m_A
 4      | | m_B
 8      | | m_C
        | +---
12      | m_D
        +---
***继承中先调用父类构造函数在调用子类构造函数，析构顺序相反***
```

继承关系中同名成员如何处理

```
class Anim
{
public:
    int m_A = 0;
    void func()
    {
        cout << "Anim Func" << endl;
    }
};

class Dog :public Anim
{
public:
    int m_A = 1;
    void func()
    {
        cout << "Dog Func" << endl;
    }
};
...
    Dog d;
    cout << "Dog Class:" << d.m_A << endl;
    cout << "Anim Class:" << d.Anim::m_A << endl;
    d.func();
    d.Anim::func();
```

- 子类对象可以直接访问子类中同名成员
- 子类对象加作用域可以方法到父类同名成员
- 子类与父类有同名的成员函数，子类会隐藏父类中的成员函数，加作用域可以访问到父类中的同名函数

多继承

语法：class 子类： 继承方式 父类，继承方式 父类 问题：多继承引发同名成员，二义性问题，需要加作用域，不太建议使用；

```
class Dog :public Anim1,public Anim2
{
public:
    int m_A = 1;
    void func()
    {
        cout << "Dog Func" << endl;
    }
};
```

菱形继承问题

介绍：需要继承的父类只存在一份 方案：利用虚继承，vbprt 虚基类指针指向虚基类表，能找到同一块内存空间

```
class Anim
{
public:
    int m_Age;
};

class Cat :virtual public Anim{};

class Dog :virtual public Anim{};

class FaimalyAnim :public Cat, public Dog {};

...

FaimalyAnim f;
f.Cat::m_Age = 19;
f.Dog::m_Age = 28;

cout << "cat:" << f.Cat::m_Age << endl;// 28
cout << "dog:" << f.Dog::m_Age << endl;// 28
```

内存模型 未使用虚继承的内存模型

```
class FaimalyAnim       size(8):
        +---
 0      | +--- (base class Cat)
 0      | | +--- (base class Anim)
 0      | | | m_Age
        | | +---
        | +---
 4      | +--- (base class Dog)
 4      | | +--- (base class Anim)
 4      | | | m_Age
        | | +---
        | +---
        +---
```

使用虚继承后的内存模型

```
 class Anim
 {
 public:
     int m_Age;
 };

 class Cat :virtual public Anim{};

 class Dog :virtual public Anim{};

 class FaimalyAnim :public Cat, public Dog {};
```

```
class FaimalyAnim       size(12):
        +---
 0      | +--- (base class Cat)
 0      | | {vbptr}// v-virtual b- ptr-pointer
        | +---
 4      | +--- (base class Dog)
 4      | | {vbptr}
        | +---
        +---
        +--- (virtual base Anim)
 8      | m_Age
        +---

FaimalyAnim::$vbtable@Cat@:
 0      | 0
 1      | 8 (FaimalyAnimd(Cat+0)Anim)

FaimalyAnim::$vbtable@Dog@:
 0      | 0
 1      | 4 (FaimalyAnimd(Dog+0)Anim)
vbi:       class  offset o.vbptr  o.vbte fVtorDisp
            Anim       8       0       4 0
```

多态

动态多态：有继承关系，子类重写父类虚函数

虚函数

```
class Anim
{
public:
    int m_Age;
    // 虚函数
    virtual void speak()
    {
        cout << "anim speak" << endl;
    }

};

class Cat :public Anim{
    void speak()
    {
        cout << "Cat speak" << endl;
    }
};

class Dog :public Anim{
    void speak()
    {
        cout << "Dog speak" << endl;
    }
};
//地址晚绑定
void doSpeak(Anim &a) // 父类的指针或者引用指向子类对象 Anim &a = Cat
{
    a.speak();
}
```

父类中使用虚函数

```
class Anim
{
public:
    Anim(){
        speak();// 不会走父类的实现
        eat();// 报错，因为此时子类还未创建
    }
    int m_Age;
    // 虚函数
    virtual void speak()
    {
        cout << "anim speak" << endl;
    }
```

```
    //纯虚函数
    virtual void eat() = 0;

};
```

虚函数内存模型

```
class Anim      size(8):
        +---
 0      | {vfptr}
 4      | m_Age
        +---

Anim::$vftable@:
        | &Anim_meta
        |  0
 0      | &Anim::speak

Anim::speak this adjustor: 0
```

纯虚函数

纯虚函数，子类需要重写父类的纯虚函数，否则即是抽象类，无法实例化

```
class Base
{
public:
    virtual void func() = 0;
};
```

虚析构和纯虚析构

// 父类指针在析构的时候，不会调用子类的析构函数，导致子类如果有堆区属性，出现内存泄漏 方案：把父类析构改为虚析构

文件操作

写文件

```
// 写文件
void test01()
{
    ofstream ofs;
    ofs.open("demo.txt", ios::out);
    ofs << "Name:Picher" << endl;
    ofs << "Age:26" << endl;
    ofs << "Area:Hubei" << endl;
    ofs.close();
}
```

读文件

```
//读文件
void test02()
{
    ifstream ifs;
    //方法一
    ifs.open("demo.txt",ios::in);
    char buff[1024] = {};
    /*while (ifs >> buff)
    {
        cout << buff << endl;
    }*/

    //方法二
    /*while (ifs.getline(buff,sizeof(buff)))
    {
        cout << buff << endl;
    }*/

    //方法三
    /*string buf;
    while (getline(ifs, buf))
    {
        cout << buf << endl;
    }*/

    //方法四
    char c;
    while ((c = ifs.get()) != EOF)
    {
        cout << c;
    }

    ifs.close();
}
```

模板

- C++泛型编程，使用的技术是模板
- C++提供两种模板，类模板，函数模板

函数模板

语法

```
template<typename T>
函数声明或定义
```

普通函数与函数模板调用规则

- 若都可以调用，优先调用普通函数
- 通过空模板参数列表，强制调用函数模板
- 函数模板可以发生函数重载
- 函数模板可以更好匹配，有限使用函数模板

模板的局限性：自定义类无法自动识别

```
template<class T>
bool compare(T& a, T& b)
{
    return a == b;
}
...
Person p1;
Person p2;
cout << compare(p1, p2) << endl; // ERROR
```

类模板

语法

```
template<typename T>
类
```

- 无自动类型推导，只能显示指定类型
- 模板参数列表可以有默认参数

类模板中成员函数创建时机：调用时才创建

使用类模板做函数参数

```
template<class T>
class Student
{
public:
    T data;
};

// 1. 指定数据类型
void func(Student<int>& s)
{
    cout << s.data << endl;
}

// 2. 参数模板化
template<class T>
void func1(Student<T>& s)
{
    cout << s.data << endl;
}

// 3. 整个类模板化
```

```
template<class T>
void func2(T& s)
{
    cout << s.data << endl;
}
```

类模板与继承

```
template<class T>
class Base
{
    Base(T age);
    void printAge();
    T m_Age;
};

template<class T>
class Son :public Base<int>
{
    T m_Name;
};

template<class T>
Base<T>::Base(T age)
{
    this->m_Age = age;
}

// 类外实现
template<class T>
void Base<T>::printAge()
{
    cout << this->m_Age << endl;
}
```

类似Java代码：

```
public abstract class Base<T>{
    abstract Base(T age);
    abstract void printAge();
    T m_Age;
}

public class Son<T> extents Base<Integer>{
    T mName;
}
...
```

STL

- Standard Template Library,标准模板库

- 广义上分为容器，算法和迭代器，容器和算法通过迭代器链接
- 几乎所有代码都采用了模板类和模板函数

六大组件

容器、算法、迭代器、仿函数、适配器、空间配置器

1. 容器：各种数据接口，如vector,list,deque,set,map，用来存放数据。
2. 算法：常用算法，sort，find，copy，for_each。
3. 迭代器：容器与算法间的桥梁。
4. 仿函数：行为类似函数，可作为算法的某种策略。
5. 适配器：一种用来修饰容器或仿函数或迭代器接口的东西。
6. 空间配置器：负责空间的配置和管理。

容器、算法、迭代器

SLT常用容器

string容器

rfind与find区别：rfind从右往左查，find从左往右查

```
//string 操作，类似java
void test03()
{
    string s = "Hello,Word!";
    int pos = s.find("Word");
    // 查找
    cout << "find:" << pos << endl;
    cout << "rfind:" << s.rfind("Word") << endl;
    // 替换
    s = s.replace(pos, s.length(), "picher");
    cout << "replace:" << s << endl;
    // 剪裁
    s = s.substr(0, pos);
    cout << "substr:" << s << endl;
}
```

vector容器

vector 动态拓展，单端数组，一段连续的内存空间，访问数据快

```
//vector测试
void test04()
{
    vector<int> v;
    for (int i = 0; i < 10; i++)
    {

        v.push_back(i);
    }
```

```
    cout << "大小: " << v.size() << endl;
    cout << "容量大小: " << v.capacity() << endl;
    printsVector(v);
    //重置大小，小于原始数据大小则删除之后数据，大于原始大小新的位置赋默认值
    v.resize(5);
    // 指定位置插入一个数
    v.insert(v.begin(), 1000);
    // 指定位置删除一个数
    v.erase(v.begin());
    // 获取元素
    v.at(2);
    //获取第一个和最后一个元素
    v.front();
    v.back();
    // 预留空间，减少动态扩容次数
    v.reserve(100);
}
```

swap收缩内存空间：vector(v).swap(v);// 匿名对象内存回收

deque 容器

双端数组，增删快，访问不如vector deque没有容量的概念

案例：比赛计分，取平均数

stack 容器

栈，先进后出，不允许有便利行为

queue 容器

队列，先进先出，不允许便利，只允许访问队头和队尾数据

list 容器

- 底层双向循环链表
- 增删快，遍历没有数组快
- 不是动态扩容的，不会造成原有迭代器的失效
- remove(elem) 删除容器中与elem相同的元素
- 不可以通过[]和at()来访问
- front()返回第一个元素，back()返回最后一个元素
- 不支持随机访问迭代器，不支持全局sort()算法

set/multiset容器

- 底层二叉树，关联式容器，插入同时会排序
- set不允许容器中有重复数据，multiset可以有重复数据
- 不支持resize()，不支持按照位置插入数据
- find(key)，找到元素返回该迭代器，找不到返回end()迭代器

map/multimap 容器

- 所有元素都是pair

```
class MyCompare
{
public:
    bool operator()(const int &v1,const int &v2) const
    {
        return v1 > v2;
    }
};

// 测试map
void test07()
{
    map<int, int, MyCompare> m;
    m.insert(pair<int, int>(1, 50));
    m.insert(make_pair(2, 20));
    m.insert(map<int, int>::value_type(3, 30));
    //m.insert(map<int, int>::value_type(3, 35));//无效

    cout << "key为3的: "<<m[3] <<"key为2的: "<<m.at(2)<< endl;
}
```

https://blog.csdn.net/iicy266/article/details/11906189

STL函数对象

- 重载了函数调用操作符的类，其创建出来的对象叫函数对象
- 函数对象使用重载的（）时，行为类似函数，也叫仿函数
- 本质是一个类，不是一个函数

```
class MyCompare
{
public:
    bool operator()(const int &v1,const int &v2) const
    {
        return v1 > v2;
    }
};
```

谓词

- 返回bool类型的仿函数叫谓词
- 一元谓词：operator接收一个参数叫一元谓词
- 二元谓词：operator接收两个参数叫二元谓词

内建函数对象

算数仿函数：加，减，乘，除，取模，取反 关系仿函数：大于、小于.. 逻辑仿函数：或与非

SLT常用算法

主要由头文件algorithem，functional,numeric组成;

遍历算法

```
// 遍历
for_each(m_L2.begin(), m_L2.end(), print);
```

```
// 拷贝并转换
class Transform
{
public:
    int operator()(int &a)
    {
        return a + 10;
    }
};
transform(m_L1.begin(), m_L1.end(), m_L2.begin(), Transform());
```

查找算法

- find //查找元素
- find_if // 按条件查找
- adjacent_find //查找相邻重复元素
- binary_search //二分查找，无序序列不可用
- count // 统计元素个数
- count_if // 按照条件统计元素个数

```
// 测试 查找算法
// 查找自定义类型 需要重载operator==
void test10()
{
    list<int> l;
    for (int i = 0; i < 10; i++)
    {
        l.push_back(i);
    }
    list<int>::iterator it = find(l.begin(), l.end(), 20);
    string str = it == l.end() ? "没找到! " : "找到了! ";
    cout << str << endl;
}
```

排序算法

- sort //对容器元素进行排序
- random_shuffle // 随机调整容器元素次数
- merge // 容器元素合并，并存储刀另一容器中
- reverse // 反转指定范围内容

拷贝和替换算法

- copy //容器中指定范围元素拷贝刀另一容器
- replace //修改指定元素
- replace_if //按条件修改元素
- swap //互换两个容器元素

算数生成算法

- 引入头文件 numeric
- accumulate // 计算区间内容器元素累计总和
- fill // 向容器填充数据

```
// accumulate计算自定义数据类型
class MyAccumlate
{
public:
    MyAccumlate(){}
    MyAccumlate(int age)
    {
        this->age = age;

    }
    int age;

    int operator()(int t1, const MyAccumlate& t2)
    {
        return (t1 + t2.age);
    }
};

// 测试accumlate 自定义数据类型
void test11()
{
    list<MyAccumlate> m_List;
    for (int i = 0; i < 20; i++)
    {
        m_List.push_back(MyAccumlate(i));
    }

    int result = accumulate(m_List.begin(), m_List.end(), 0, MyAccumlate());
    cout << result << endl;
}
```

集合算法

- set_intersection //求两个容器交集
- set_union //求两个容器并集
- set_difference //求两个容器差集

并发与多线程

线程实例

```
#include <iostream>
using namespace std;
#include <thread>
#include <mutex>

int m_Count = 20;
mutex m;

void thread1()
{
    while (m_Count > 0)
    {
        //mutex 容易死锁
        //m.lock();
        //lock_guard 能够自解锁,当生命周期结束时，它会自动析构，不影响其他线程
        lock_guard<mutex> lockGuard(m);
        if (m_Count > 0)
        {
            --m_Count;
            cout << m_Count << endl;
        }
        //m.unlock();
    }
}
void thread2()
{
    while (m_Count > 0)
    {
        //m.lock();
        lock_guard<mutex> lockGuard(m);
        if (m_Count > 0)
        {
            --m_Count;
            cout << m_Count << endl;
        }
        //m.unlock();
    }
}

// 测试线程加锁
void test07()
{
    cout << "threaId:" << this_thread::get_id() << endl;
    thread th1(thread1);
    thread th2(thread2);
    //th1.join(); //同步执行
    th1.detach(); //异步执行
}
```

使用了detach()后的问题：

- 使用thread构造传参的时候一定不能用指针，主线程退出后会有非法引用问题
- 原理：thread再创建时，若传入了临时变量，其会创建一个备份对象，防止原对象被回收；
- 建议：若传递int这种简单类型数据，建议值传递；若传递类对象，用引用接收

C++中的锁机制

- 线程中的锁分为两种，互斥锁和共享锁。

mutex

- 互斥量，当A持有锁，但是没有unlock的时候（可能是unlock前抛出异常，或忘了），B一直请求不到资源而等待，就会发生死锁。
- 还有recurisve_mutex，递归独占锁，允许被lock多次；time_mutex，使用try_lock_for 方法，在超时后还未拿到锁可以处理其他逻辑。

lock_guard

- lock_guard只有构造函数和析构函数。简单的来说：当调用构造函数时，会自动调用传入的对象的lock()函数，而当调用析构函数时，自动调用unlock()函数。

unique_lock

- 是个类模板，类似lock_grard，扩展了一些方法，析构时也会自动释放锁
- 效率没有lock_guard高，灵活性好
- 在已经持有锁的情况下，防止重复加锁，用adopt_lock管理该锁
- try_to_lock ，尝试拿锁，若拿不到不会卡着，可以执行别的事情
- defer_lock，不能先lock，初始化了一个没有加锁的mutex
- release，返回mutex指针，解除与mutex的关系

https://blog.csdn.net/guotianqing/article/details/104002449

异步任务

async创建后台任务并返回值

- std::async是一个函数模板，用来启动一个异步任务，启动起来的异步任务会返回一个std::futured对象，这个对象里面有异步任务的返回结果。
- 比直接使用线程好用，解耦了线程的创建和执行，提供了线程的创建策略 使用

```
future<int> f = async(func);
```

- async(launch::deferred,func)，launch::deferred表示延迟创建线程，调用feture.get 开始在主线程执行。

> 深入浅出 c++11 std::async

async与Thread的区别

1. threads会创建线程，若资源紧张，创建线程失败会报异常
2. async不一定创建线程，会自动判断延迟调用或创建线程。没有资源时，不会报错，再调用get后开始执行创建线程

packaged_task

- 任务包，可入容器，支持lambda表达式

```
// 测试package_task
void test12()
{
    cout << "主线程: " << this_thread::get_id() << endl;

    packaged_task<int(int)> mypt([](int inMemb)
        {
            cout << "执行线程: "<<this_thread::get_id() << endl;
            this_thread::sleep_for(chrono::milliseconds(2000));
            return 5;
        });
    /*thread t1(ref(mypt), 1);// 放入子线程中
    t1.join();*/

    mypt(10);// 此行在主线程调用
    int result = mypt.get_future().get();
    cout << result << endl;
}
```

线程间通讯

future

- future 的三种状态:

1. deferred // 异步操作还没开始
2. ready // 异步操作已经完成
3. timeout // 异步操作超时
4. 通过`future.wait_for()`查询状态;

- future只能get一次，若想get多次用share_future **promise**
- 可在某个线程中赋值，在其他线程中取出来使用

> c++11并发与多线程视频课程

线程池

https://www.cnblogs.com/lzpong/p/6397997.html

// TODO

进程间通讯

设计模式

单例模式

- 其特点是只提供唯一一个类的实例,具有全局变量的特点，在任何位置都可以通过接口获取到那个唯一实例;
- 全局只有一个实例：static 特性，同时禁止用户自己声明并定义实例（把构造函数设为 private）
- 线程安全
- 禁止赋值和拷贝
- 用户通过接口获取实例：使用 static 类成员函数

```
// 使用局部静态变量实现懒汉式单例
// 比较推荐的一种写法，不用使用共享指针
class SingTone
{
private:
    SingTone() { cout << "Cteate Class!" << endl; }
    ~SingTone() { cout << "Delete Class!" << endl; }
public:

    static SingTone& getInstance()
    {
        static SingTone instance;
        return instance;
    }
};
```

单例的模板

```
template<typename T>
class SingleToneTmp
{
protected:
    SingleToneTmp() { cout << "Cteate Class!" << endl; }
    ~SingleToneTmp() { cout << "Delete Class!" << endl; }
public:

    static T& getInstance()
    {
        static T instance;
        return instance;
    }
};

class StudSingle :public SingleToneTmp<StudSingle>
{

};
```

工厂模式

简单工厂

```
enum class CarType { BUICK, CHEVY, CADILIC };

class Car {
public:
    virtual void createdCar(void) = 0;
};
...
class Cadilic : public Car
{
```

```
public:
    virtual void createdCar(void)
    {
        cout << "Create Cadilic!" << endl;
    }
};

...

class CarFactory {
public :
    Car* createCarBuyType(CarType type)
    {
        switch (type)
        {
        case CarType::BUICK:
            return new Buick;
        case CarType::CHEVY:
            return new Chevy;
        case CarType::CADILIC:
            return new Cadilic;
        default:
            break;
        }
    }
};
```

抽象工厂

```
class Factory
{
public :
    virtual Car* createCar() = 0;
};


class CadilacFactory :public Factory
{
public:
    virtual Car* createCar()
    {
        cout << "造了一台cadilac" << endl;
        return new Cadilic;
    }
};

...

    shared_ptr<Factory> f_Sp(new CadilacFactory());
    f_Sp.get()->createCar();
```

策略模式

```
class Phone
{
public:
    virtual void placeCall() = 0;
    virtual void muteCall() = 0;
};

class Huawei :public Phone
{
public :
    Huawei() {}
    virtual void placeCall() {
        cout << "用huawei 打电话! " << endl;
    }
    virtual void muteCall() {
        cout << "用huawei 静音电话! " << endl;
    }
    ~Huawei() { cout << "Huawei析构" << endl; }
};

class Oppop :public Phone
{
public:
    Oppop() {}
    virtual void placeCall() {
        cout << "用Oppop 打电话! " << endl;
    }
    virtual void muteCall() {
        cout << "用Oppop 静音电话! " << endl;
    }
    ~Oppop(){cout << "Oppop析构" << endl;}
};

class Human
{
public :
    Human(Phone* p)
    {
        m_Phone = p;
    }
    void mackCall()
    {
        if (m_Phone != nullptr)
        {
            m_Phone ->placeCall();
        }
    }
    ~Human() {cout << "huamn析构" << endl;}
private :
    Phone* m_Phone;
};

...
```

```
    shared_ptr<Huawei> m_huaweiSp(new Huawei());
    shared_ptr<Oppop> m_oppoSp(new Oppop());

    shared_ptr<Human> m_hSp1(new Human(m_huaweiSp.get()));
    shared_ptr<Human> m_hSp2(new Human(m_oppoSp.get()));
    m_hSp1.get()->mackCall();
    m_hSp2.get()->mackCall();
```

建造者模式

c++设计模式之建造者模式

代理模式&装饰模式

观察者模式

责任链模式

其他设计模式

- 中介者模式
- 备忘录模式
- 解释器模式
- 迭代器模式
- 桥接模式
- 组合模式
- 外观模式
- 享元模式

C++11 语言特性

参考：C++11常用新特性快速一览

指针空值

旧版本中NULL兼容性问题

```
void Func(char *);
void Func(int);
...
// 下面代码调用的是Func(int)，但NULL作为一个void* 指针应该调用第一个函数
Func(NULL);
```

- nullptr 出现的目的是为了替代 NULL，用来区分空指针和0，是C++一个关键字
- sizeof(nullptr_t) == sizeof(void*)

https://www.cnblogs.com/developing/articles/10890886.html

auto关键字

表示自动推导

```
int x = 10;
auto *a = &x;       //auto被推导为int
auto b = &x;        //auto被推导为int*
auto &c = x;        //auto被推导为int
auto d = c;         //auto被推导为int

const auto e = x;   //auto被推导为int
auto f = e;         //auto被推导为int

const auto& g = x;  //auto被推导为int
auto& h = g;        //auto被推导为int

//int i = 10;
//h = i;//error,因为 h为const int& h
```

- 当表达式是一个引用类型时，auto会把引用类型抛弃，直接推导成原始类型int
- 当表达式带有const属性时，auto会把const属性抛弃掉，直接推导成int类型
- 当auto和引用结合时，auto的推导将保留变量的const属性

使用限制

- 不能作为函数的形参
- 不能用于非静态成员变量
- auto无法推导出模板参数
- auto无法定义数组

使用场景:

```
//旧特性中获取迭代器
  vector<int>::iterator it = MyVector.begin();
//新特性中获取迭代器
  auto MyIt = MyVector.begin();
```

decltype关键字

从表达式的类型推断出要定义的变量类型，但不初始化变量

```
int x = 1;
int y = 2;
decltype(x + y) z;// 推到出z是int类型
```

区间迭代

```
    //旧版本中for循环
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //新版本
    for (auto& i : v)
    {
        cout << i << " ";
    }
```

初始化列表

可直接使用{}完成初始化，如下示例

```
struct Student
{
    int m_Age;
    string m_Name;
};

// 测试初始化列表
void test05()
{
    Student s1 = { 1,"picher1" };
    Student s2 = { 2,"picher2" };

    cout << s1.m_Age << s1.m_Name << endl;
}
```

模板增强

- 外部模板，不在该编译文件中实例化模板
- 默认模板参数，指定模板的默认参数类型

构造函数

- 委托构造

```
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() {  // 委托 Base() 构造函数
        value2 = 2;
    }
};
```

- 继承构造

智能指针

- 是行为类似指针的类对象，目的是为了自动回收，避免内存泄漏
- 提供了四种智能指针auto_ptr（Depreced），unique_ptr(独占式),shared_ptr(有循环引用内存泄漏问题), weak_ptr(解决循环引用问题) 详解C++11智能指针

其他内容

- 支持lambda表达式
- 新增容器array
- 新增无序关联容器（unordered_set、unordered_multiset、unordered_map和 unordered_multimap）使用键和哈希表，以便能够快速存取数据。
- 统一了初始化列表方式，可不加=号，要注意防止类型收窄

```
    int a { 10 };
    int arr2[] {1,2,3,4,5};
    vector<int> v{ 1,2,3,4 };
    Student s{10,"picher"};
```

- 增强for循环

```
for (auto i : v) cout << i << endl;
```

- 作用域内枚举
- using模板命名替代typedef
- 建议忽略异常规范
- 新增头文件random、chrono(处理时间间隔)、tuple(pair增强，可存储两对以上)、regex(支持正则)、ratio

单元测试

GoogleTest

- 可通过 ASSERT* *和 EXPECT** 断言来对程序运行结果进行检查.
- ASSERT* *版本的断言失败时会产生致命失败，并结束当前函数；* EXPECT* 版本的断言失败时产生非致命失败，但不会中止当前函数

- 通过 RUN_ALL_TESTS() 来运行所有测试用例

IBM Developer-轻松编写C++ 单元测试

开发流程

遵循《UploadCodeProcess》

工具使用

VC常用快捷键

功能	VC	AS
代码补全	TAB	ALT + /
参数提示	CTRL + SHIFT + 空格	CTRL + P
多行注释/反注释	CTRL + K + C/CTRL + K + U	CTRL+/
代码格式化	CTRL + K + F	CTRL + ALT + L
转小写	CTRL + U	CTRL + U
转大写	CTRL + SHIFT + U	CTRL + U
转到行	CTRL + G	CTRL + G
重命名	CTRL + R + R	SHIFT + F6
删除一行	CTRL + SHIFT +L	CTRL + Y
复制一行	CTRL + D	CTRL + D
移动行	ALT + ↑/↓	CTRL + ALT + ↑/↓

代码规范

编码规范遵循《VCS Cluster HMI Development Guideline》 静态代码检测遵循《cpplint.docx》

总结

与Java的区别

总体来看，都是面向对象语言，思想基本一致。基础数据类型，类的特性，设计模式，容器用法也类似集合。 不同点在：指针，引用，函数定义方式，传参方式，类的继承方式，析构，对泛型的使用方式（C++是模板）,还有语言的特殊写法

下一步计划

- 用一个案例巩固知识，查缺补漏
- 配置Kanzi开发环境

- 了解Kanzi相关知识
- 阅读Cluster相关源码