## C & C++

### C 和 C++的差别

* C是面向过程的，重点在于算法和数据结构，C++多了面向对象，支持继承、封装、多态
* C++是C的超集，多了string，布尔类型，STL
* C++有重载，模板实现静态多态，inline、extern函数
* C中其他可以调用main(), C++不可以

### struct 和 class 差别

* C++中两者的区别，成员默认访问权限和默认继承权限public和private的差别。模板中不能用struct
* C中的struct，不能有函数的存在，没有访问权限都类似于public，C语言中没有类的概念，可以在结构体内创建指针成员实现面向对象

### 宏定义 和 inline 的区别

Inline和普通函数相比可以加快程序运行的速度，因为编译的时候内联函数直接镶嵌到目标代码当中，不需要中断调用。内联函数会做类型检查，宏不是函数，只是一个简单的替换。

### extern “C” 的作用

C++支持重载，编译后库中的名字于C不同，如：int foo(int x, int y) （\_foo \_foo\_int\_int）。

作用是告诉编译器按C语言的方式编译

## C++11

### 左值、右值

* **左值**：能取地址，有名字。一般变量都是
* **右值**：不能取地址，没有名字，一般通过引用的方式找到他的存在。表达式的值函数返回值。
  + 纯右值：临时对象，常量
  + 将亡值：返回有值引用的函数返回值，std::move()的返回值
* **右值引用**：不具名变量的引用，都属于引用类型，必须初始化

只要临时对象的应用存在，临时对象就会一直存在，

不能绑定到左值对象。 Is\_rvalue\_reference()<type\_traits>

* **移动构造**：移动语义，有值引用作为参数，将原来指向堆的指针置空，新的指针指向原来的堆

避免了构造时的再次申请内存



### move 与forward

* std::move：将一个左值强制转化成右值引用。只做转换不做移动。确保是生命期即将结束的对象，可以调用移动构造函数。可以避免不必要的拷贝，为性能而生
* std::forward：完全依照模板的参数类型，将产生传递给模板函数中调用的另一个函数，左值对象仍然是左值对象，右值对象仍然是右值对象；

通过引用折叠实现，将复杂未知的表达式折叠为已知简单的表达式。



模板类型推导，不管TR的定义类型， TR是左值就左值，TR是右值就右值

### emplace\_back 与push\_back

* emplace\_back(临时对象)，只需一次构造，不需额外内存
* push\_back(),参数是右值引用是，一次构造，一次迁移拷贝
* push\_back(),左值引用，一次拷贝，一次构造

### 智能指针

* + 1. **设计思想**

**将基本类型指针封装为类对象指针，并在析构函数里编写delete语句删除指针指向的内存空间，**

explicit构造函数

赋值方式

执行深层拷贝，多出一个副本浪费空间

建立所有权概念，智能有一个智能指针可拥有，赋值操作转让所有权。

引用计数，赋值时加1，过期是减一

指针被auto\_ptr接管之后再去使用，导致运行时崩溃

如果被unique\_ptr接管之后，再使用是编译出错。不能赋值，可以move

**总之，当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做**，

 shared\_ptr 为强引用, 如果存在循环引用, 将导致内存泄露. 而 weak\_ptr 为弱引用, 可以避免此问题

循环引用

weak\_ptr 查看引用数

### lambda

匿名的函数，使用函数不需要给出名字，实现的函数名字和功能的分离

[ capture-list ] ( params ) -> ret { body }

capture父作用域的变量

### async

## 指针& 引用

### 指针和引用的区别

## mutable

需要在const方法中修改对象的数据成员时，可以在数据成员前使用mutable关键字，防止出现编译出错。

## static

* 静态成员函数属于类而不是类对象，所以没有this指针，就是读写静态成员的，不能访问普通成员
* 内存中只有一份数据，所有的对象共享，必须初始化，不能再类的定义中初始化
* 想当于nomember 函数，有利于成为callback，方便作为线程函数

## 虚函数virtual

### virtual 介绍和使用规则

* 虚函数的作用主要是实现了多态的机制，也就是通过基类指针调用实际派生类的成员函数
* 非类的成员函数不能定义为虚函数，
* 类的成员函数中静态成员函数和构造函数也不能定义为虚函数，但可以将析构函数定义为虚函数

### 虚析构

Delete一个指向派生类对象的基类指针，子类和基类的虚构函数都会调用。非虚析构的话，只调用基类的析构函数

### 虚函数表

 为了保证正确取到虚函数的偏移量，编译器必需要保证虚函数表的指针存在于对象实例中最前面的位置

typedef void(\*Fun)(void);

Base b;

Fun pFun = NULL;

cout << "虚函数表地址：" << (int\*)(&b) <<endl; //虚表的地址

cout << "虚函数表 — 第一个函数地址：" << (int\*)\*(int\*)(&b) <<endl;

pFun = (Fun)\*((int\*)\*(int\*)(&b)); // 将地址转化成函数地址

pFun();

(Fun)\*((int\*)\*(int\*)(&b)+0); // Base::f()

(Fun)\*((int\*)\*(int\*)(&b)+1); // Base::g()

(Fun)\*((int\*)\*(int\*)(&b)+2); // Base::h()

### 虚继承

在多继承中，子类可能同时拥有多个父类，如果这些父类还有相同的父类（祖先类），那么在子类中就会有多份祖先类。例如，类B和类C都继承与类A，如果类D派生于B和C，那么类D中就会有两份A。为了防止在多继承中子类存在重复的父类情况，可以在父类继承时使用虚函数，即在类B和类C继承类A时使用virtual关键字，

## 类对象的存储

（1）如果类中含有虚函数，则编译器需要为类构建虚函数表，类中需要存储一个指针指向这个虚函数表的首地址，注意不管有几个虚函数，都只建立一张表，所有的虚函数地址都存在这张表里，类中只需要一个指针指向虚函数表首地址即可。

（2）类中的静态成员是被类所有实例所共享的，它不计入sizeof计算的空间

（3）类中的普通函数或静态普通函数都存储在栈中，不计入sizeof计算的空间

（4）类成员采用字节对齐的方式分配空间

## 拷贝构造函数

（1）函数的参数为类对象且参数采用值传递方式；

（2）将类对象做为函数的返回值。

## 异常处理exception

## 虚函数virtual

## 虚函数virtual