1. new

1. A *a = new A; a->num = 0;发生了什么
    1. A* a：在栈区开辟4字节的指针空间给a
    2. new A：在堆区分配内存空间，调用A的构造函数，返回指针
    3. a = new A；a获取动态内存指针
    4. a->num = 0；从指针a找到该对象首地址，根据num的内偏移量找到对应的int地址赋值
2. new是否调用构造函数
    1. new自定类型时自动调用构造函数
    2. new内置数据类型，int *a = new int;

       不加（）不调用构造函数初始化
3. std::bad_alloc异常
4. new数组默认初始化
5. new和malloc的区别（使用malloc一定注意判断返回指针是否为NULL）

   new是运算符可以重载，malloc是函数

   new不可改变空间大小，malloc可以调用realloc

   delete[]多次调用析构函数，malloc同时销毁

特征	new/delete	malloc/free
分配内存的位置	自由存储区	堆
内存分配失败返回值	完整类型指针	void*
内存分配失败返回值	默认抛出异常	返回NULL
分配内存的大小	由编译器根据类型计算得出	必须显式指定字节数
处理数组	有处理数组的new版本new[]	需要用户计算数组的大小后进行内存分配
已分配内存的扩充	无法直观地处理	使用realloc简单完成
是否相互调用	可以，看具体的operator new/delete实现	不可调用new
分配内存时内存不足	客户能够指定处理函数或重新制定分配器	无法通过用户代码进行处理
函数重载	允许	不允许
构造函数与析构函数	调用	不调用

2. static

1. static关键字--静态全局变量和静态局部变量，分布在.data / .bss段
2. 生成local符号，外文件不可访问，起到外文件隐藏作用
3. static声明的成员变量不能通过构造函数进行初始化，在类外初始化（仅一次）

```
int Myclass::data1 = 20;
int Myclass::data2 = 30;
//初始化方式为作用域+变量+初始值
```

4. static成员没有this指针，只能调用static成员函数和成员方法，不能声明为const、虚函数
5. static保存在函数虚拟地址区域，作用域仅声明区域可用
6. 1.静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；
   2.非静态成员函数可以任意地访问静态成员函数和静态数据成员；
   3.静态成员函数不能访问非静态成员函数和非静态数据成员；
   4.调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数,也可以用类名::函数名调用(因为他本来就是属于类的，用类名调用很正常)
7. static关键字至少有下列n个作用：
   (1) 函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；
   (2) 在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；
   (3) 在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；
   (4) 在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；
   (5) 在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员变量。
8. const的作用：const需要进行初始化

   const关键字至少有下列n个作用：
   (1) 欲阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；
   (2) 对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为const；
   (3) 在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
   (4) 对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的 成员变量；
   (5) 对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为"左值"。例如：

3. 多态

静态多态：重载、模板

静态多态的函数地址在编译阶段绑定

动态多态：虚函数

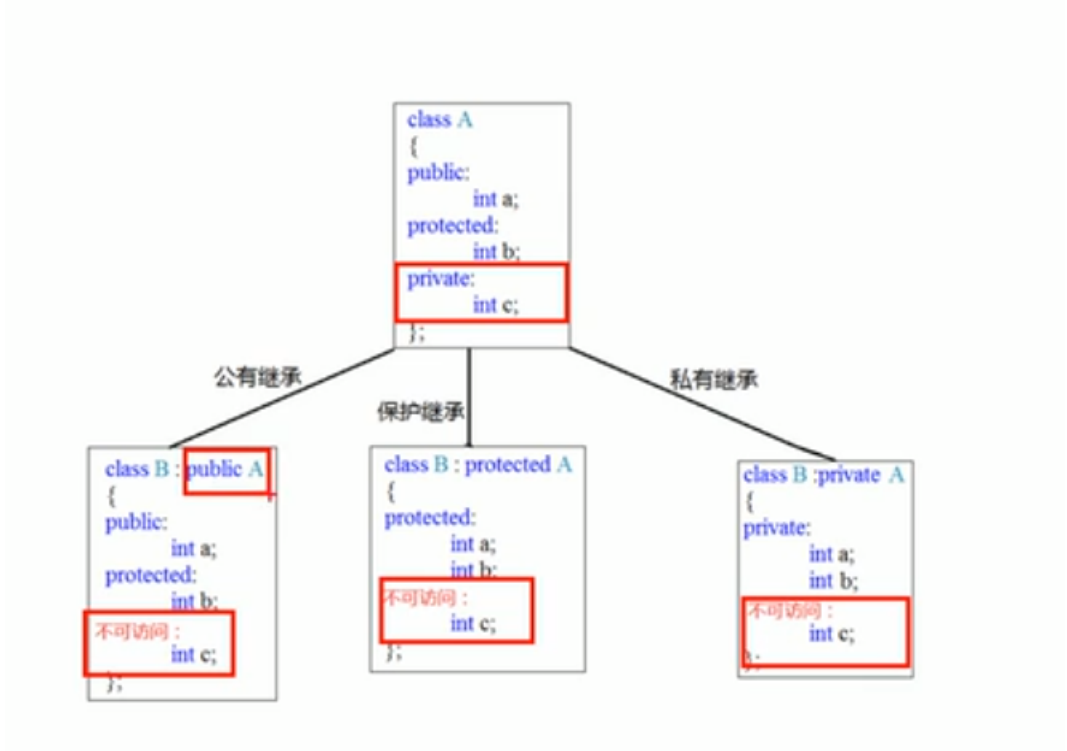
函数地址在运行时再绑定

实现条件：虚函数、一个基类的指针指向派生类对象

1. 虚析构函数的作用

   当基类指针指向了派生类对象时，假设析构函数不为虚函数，那么释放指针时仅能够访问到基类的析构函数，派生类对象的其余内容无法释放造成内存泄露，所以对析构函数进行动态重载。使基类指针能够动态联编到派生类对象的析构函数。

2. 为什么析构函数不是默认虚函数
   1. 虚析构函数是为了处理继承关系中，基类指针无法调用派生类的析构函数导致派生类对象无法释放的问题
   2. 因为虚函数需要虚函数表和虚函数指针，占用额外内存，那么内存就会被浪费
3. 继承：减少重复代码



4. 继承的对象模型：

   父类中所有的非静态成员属性都会被子类继承下去

   sizeof会查看到具体大小

   如果父类为空类，继承时省略占位的1B大小

   父类中的私有成员是被编译器隐藏了，但是真实存在

   查看内容：reportSingleClassLayoutxxx

5. 继承的构造和析构顺序

   基类的构造函数-派生类成员对象的构造函数-派生类的构造函数

   派生类的析构函数-成员对象的析构函数-基类的析构函数

6. 继承中同名成员的处理方式

   1. 直接调用调用到的为子类
   2. 子类重名会覆盖父类同名成员
   3. 使用域标识符可以区分父子类成员

7. 继承中同名静态成员处理方式

   1. 同上
   2. 使用域标识符访问
   3. 对于同名函数，子类覆盖所有父类

8. 菱形继承中，访问父类的同名成员需要加作用域，否则产生二义性

9. 虚继承：继承前加virtual

   虚继承所有成员变量仅保存一份

10. 模板

11. 使用关键字template

12. 模板两种方式：自动类型推导，显示指定类型

    1. 自动类型推导

       swaps(a,b);

       显式指定类型：

swaps(a,b);

2. 自动类型推导：必须推导出一致的数据类型才能使用
3. 模板必须给出T的数据类型

13. 模板的目的：提高复用性，类型参数化
14. 普通函数与模板函数的区别

    只有明确给定函数参数类型的才能隐式类型转换

    1. 普通函数调用可以隐式类型转换
    2. 函数模板自动类型推导不能隐式类型转换
    3. 函数模板显示指定可以转换
15. 调用规则

```
//普通函数与函数模板调用规则
//1、如果函数模板和普通函数都可以调用，优先调用普通函数
//2、可以通过空模板参数列表 强制调用 函数模板
//3、函数模板可以发生函数重载
//4、如果函数模板可以产生更好的匹配，优先调用函数模板
```

16. 某些特定的数据类型不能使用模板，需要自定义具体化实现

```
//利用具体化Person的版本实现代码，具体化优先调用
template<> bool myCompare(Person &p1, Person &p2)
```

17. 类模板和函数模板的区别

    1. 类模板没有自动类型推导--**必须显示给定**
    2. 类模板在模板参数列表可以有默认参数

       函数模板不可以有默认参数

```
template<class NameType,class AgeType = int>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
```

18. 类模板中，成员函数的创建时机

    1. 类模板在调用时才创建
    2. 普通成员函数在一开始创建
19. 类模板基类进行继承时，必须指定基类的模板参数

```
template<class T>
class Base
{
    T m;
};

//class Son :public Base //错误，必须要知道父类中的T类型，才能继承给子类
class Son:public Base<int>
{

};
```

或派生类也为模板类

```
//如果想灵活指定父类中T类型，子类也需要变类模板
template<class T1,class T2>
class Son2 :public Base<T2>
{

    T1 obj;
};
```

20. 模板类的成员函数的类外实现，在类作用域前也需要加上模板的参数

    没用到也必须加

```
//构造函数类外实现
template<class T1,class T2>
Person<T1,T2>::Person(T1 name, T2 age);
```

21. 类模板成员函数不能分文件编写，编译时可能连接不到

    1. 直接包含cpp源文件
    2. 声明和实现写到一个文件中改名为hpp

22. 重载的底层实现机制

    源文件通过编译后，将相同函数名，按照一定的格式，改变成可以区分的，去除了函数在调用时的二义性，从而实现函数的重载。

    c语言编译时只会给函数名加上简单的重命名：_

    C++根据参数个数、类型将函数复杂重命名，编译器会根据某个函数的参数修改其调用的函数名称

    extern C告诉编译器这段使用C编译器，使用C型的函数重命名机制

4. C++的内存管理

1. 内存泄露：经过动态分配的内存块必须进行释放，如果没有进行free或delete，会导致这部分内存无法被释放
2. 使用linux的ps指令查看 ps -aux查看进程状态
3. Linux内存泄露检测：

   __wrap_malloc()

   __wrap_free()

   mtrace--检测malloc/realloc/free

   memwatch--能双重释放、错误释放、溢出、下溢、泄露

   valgrind（gcc -g）--溢出、非法访问、未初始化内存

   4. 垃圾回收算法
   5. 引用计数算法：指向该位置设定计数器，如果到0自动请0
   6. 标记清除算法：从有向图中标记所有可达节点并染色，未染色的节点将被清除
   7. 节点复制算法：引用过的对象复制到新区域中，旧区域全部释放，只维护新区域
   8. 分代回收：如果遇到老生代的话，不再进行递归
   9. free、delete、**wrap_malloc、** warp_free

4. 重写和重载的区别

No	区别点	重载	重写
1	单词	Overloading	Overriding
2	定义	方法名称相同，参数的类型或个数不同	方法名称、参数的类型、返回值类型全部相同
3		对权限没有要求	被重写的方法不能拥有比父类更加严格的权限
4	范围	发生在一个类中	发生在继承中

1. 重载规则
    1. 参数列表必须不同
    2. 可以有不同的访问修饰符
    3. 可以抛出不同的异常
2. 重写规则
    1. 参数列表必须完全相同。否则称之为重载
    2. 返回的类型必须相同
    3. 重写方法一定不能抛出新的异常/更宽泛的异常
3. 重写是子类与父类的关系，是垂直关系

    重载是同一个类方法之间的关系，是水平关系

5. UTF-8和UTF-16的区别

Unicode字符集的抽象码映射到8位，16位，用于数据存储和传递

1. UTF-16无法兼容ASCII码，但是可以存储大部分字符
2. UTF-8可以通过屏蔽位、移位的方式进行读写
3. UTF-8可以用一个字节表示字符，也可以用多个字节表示字符

    110xxxxx 10xxxxxx.如果是这样的格式,则把两个字节当一个单元

    1110xxxx 10xxxxxx 10xxxxxx 如果是这种格式则是三个字节当一个单元.

4. 常用中文字符集有3个字节，更大范围的中文字符集常见4个字节

6. 常见图片格式

JPG:有损、不支持透明、动画，非矢量，色彩还原度好

JPEG：存储了相机的一些参数，饱和度亮度等

PNG：无损，支持透明、不支持压缩、非矢量

GIF：小、不支持透明、非矢量

BMP：色彩真实，但是大

7. memset

1. 使用memset初始化类对象造成的问题

    每个包含虚函数的类对象都包含一个虚函数指针4B指向虚函数表

    该指针隐藏，且不可取

    进行memset操作时，这个指针的值要被复写，一旦调用虚函数，就会发生非法访问情况。

8. include

1. 在预编译的时候，以递归调用的方式加载到代码中
2. <>表示标准/系统提供的头文件，直接到保存系统头文件的位置查找头文件
3. “”常用于程序员自己定义的头文件，C编译器先去当前目录查找是否有指定的头文件，再去标准头文件中查找
4. 头文件引用顺序

   在A.h中声明一个b.h里面有的的重名变量，必须先引用b.h再引用a.h，不然变量类型未声明
5. 对于使用双引号包含的头文件，查找头文件路径的顺序为：
   当前头文件目录
   编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
   系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径
   对于使用尖括号包含的头文件，查找头文件的路径顺序为：
   编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
   系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

9. map，set底层结构

1. map multimap set multiset底层使用红黑树
2. unordered_map unordered_set底层使用哈希表

10. 虚函数

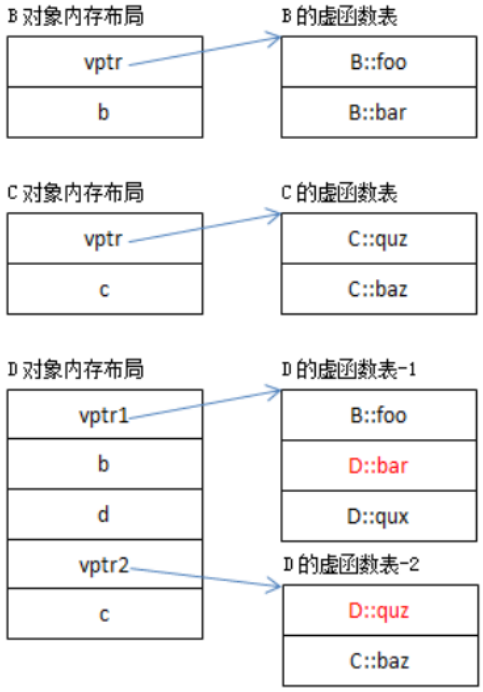
1. 虚函数表vtpr

   如果类中有虚函数/继承的父类中有虚函数，那么类存在一个虚函数表

   在子类的构造过程中，先对vtpr初始化，使其指向父类的虚函数表，等到父类构造完成，子类的构造函数调用后，才会指向子类的虚函数表，在构造函数中使用虚函数不会引发多态

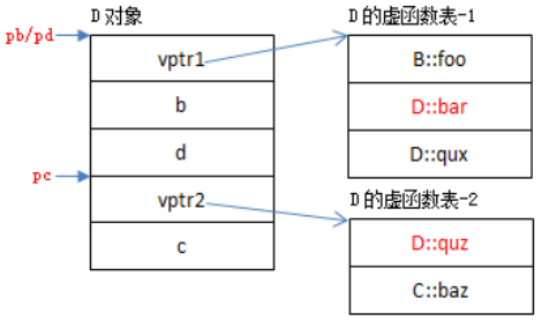
   虚函数表就是为了保存类中的虚函数的地址。我们可以把虚函数表理解成一个数组，数组中的每个元素存放的就是类中虚函数的地址。当调用虚函数的时候，程序不是像普通函数那样直接跳到函数的代码处，而是先取出vptr即得到虚函数表的地址，根据这个来到虚函数表里，从这个表里取出该函数的地址，最后调用该函数。所以只要不同类的vptr不同，他对应的vtbl就不同，不同的vtbl装着对应类的虚函数地址，这样虚函数就可以完成它的任务了。
2. 虚函数：如果虚函数在基类中声明为private，仍然可以进行重写，但是无法访问基类的该函数
3. 虚函数表：
   1. 在编译时生成，放在只读数据段
   2. 每个类一个，同类不同对象使用一张虚函数表
4. 虚函数底层实现机制：虚函数表和虚函数指针
   1. 为类对象添加一个隐藏的虚函数指针指向该对象的虚函数表
   2. 派生类重写了基类的虚方法，虚函数表将重写虚函数地址
   3. 若为重写，派生类继承基类的虚方法，虚函数保存基类中未被重写的虚函数地址，如果添加了新的虚函数，那么添加到虚函数表中
5. 虚函数表的构造过程
   1. 首先拷贝基类的虚函数表
   2. 替换已被重写的虚函数表中的虚函数地址
   3. 追加子类自己的虚函数指针（保证偏移位置相同）
6. 类指针指向谁，就调用谁的虚函数指针vtpr
7. 多重继承且基类都含有虚函数，子类将含有多个虚函数表的指针

```
struct B {
    long b;
    virtual void foo() {}
    virtual void bar() {}
};
struct C {
    long c;
    virtual void quz() {}
    virtual void baz() {}
};
struct D : public B, public C {
    long d;
    virtual void bar() {} // override B::bar
    virtual void quz() {} // override C::quz
    virtual void qux() {}
};
```



此时多态的虚函数指针的起始位置不同，各自调用自己的

```
void test() {
    D* pd = new D();
    B* pb = pd;
    C* pc = pd; // pc points to (char*)pd + 24
    pb->bar();
    pc->quz();
}
```



8. 虚函数重写，派生类可以不加函数的virtual标识符
9. 纯虚函数

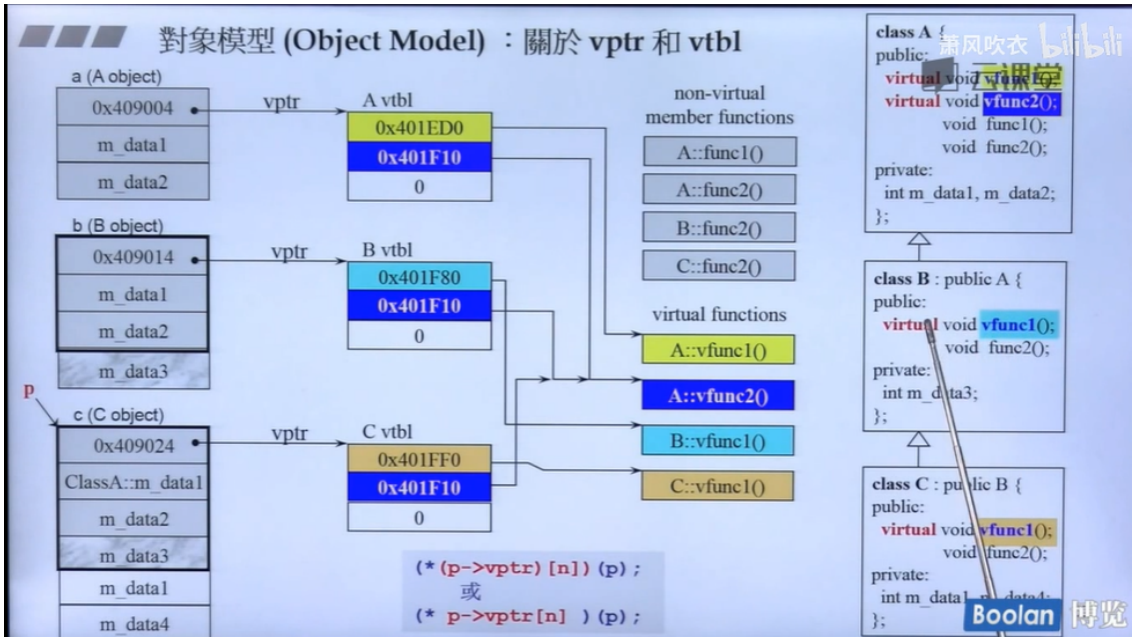
纯虚函数语法：`virtual 返回值类型 函数名 （参数列表）= 0 ;`

若类含有纯虚函数，称之为抽象类

抽象类特点：

1. 无法实例化对象
2. 子类必须重写纯虚函数，不然也无法实例化对象

10. 虚析构函数与纯虚析构
    1. 如果子类有堆区动态变量，父类析构函数必须声明为虚析构函数，否则无法调用子类的析构函数
    2. 纯虚析构在类内声明，类外必须实现，同时该类也是抽象类，无法实例化
11.



静态绑定：编译器会编译成call，一定指向某个地址

动态绑定：指针+向上转型+调用虚函数，由于p内容不一定，所以需要执行时才能找到

```
(*(p->vptr)[n])(p);
```

```
call        dword ptr [edx]
```

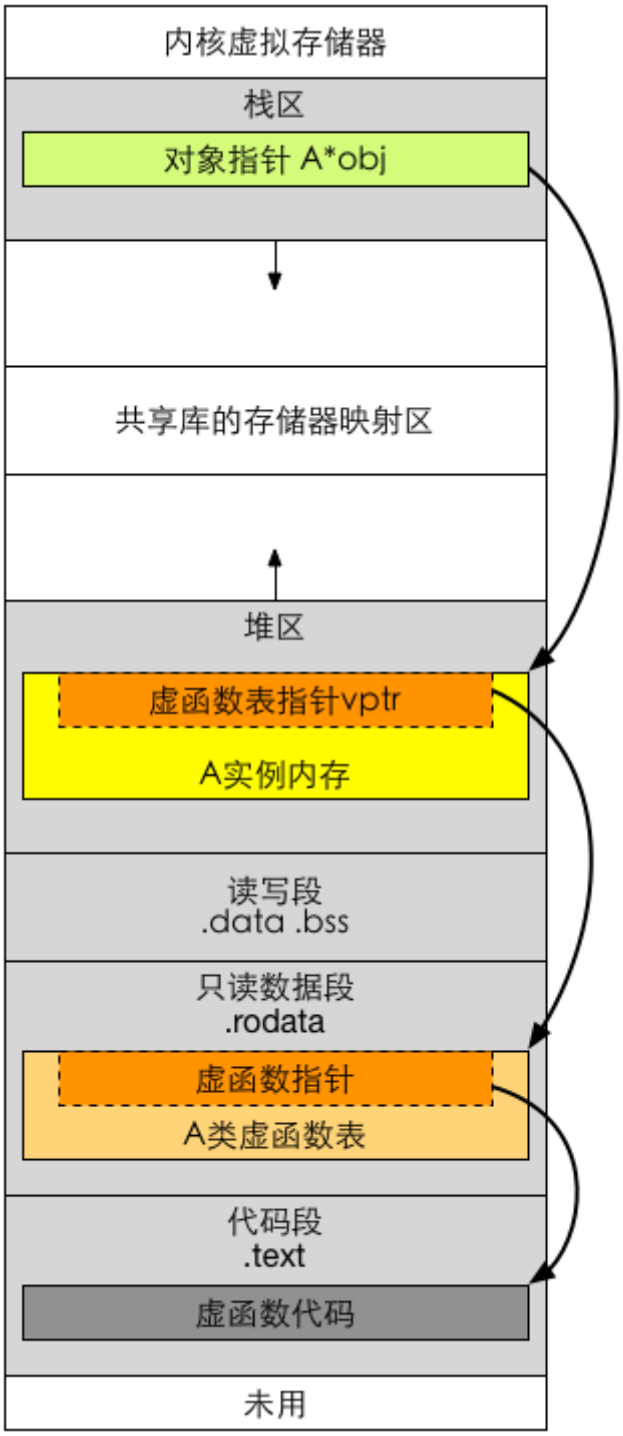
12. 构造函数为什么不能是虚函数

    虚函数表在构造函数调用后才能建立，所以构造函数不能是虚函数。

    构造函数和虚函数表在构造的时候是为本类进行构造的，不需要考虑后续派生类的构造过程。

    在虚函数表构造完毕后才会修改虚函数指针指向。

13.



14. 使用函数指针强制访问虚函数表

11. sizeof关键字

1. sizeof：编译时运行，查看的是一个变量在栈上的投影大小
2. sizeof char a[20] = 20

   sizeof(char *a) = 4

3. 一个数组传入函数参数，退化为指针

12. decltype关键字

1. 表达式是变量的话，返回变量的类型

   const int a, &b =a;

   decltype(a) c = 0;

   decltype(b) d = b;

2. 与auto不同，会保留顶层的const和引用
3. 表达式不是变量，返回表达式对应的类型

   int a = 0, *p = &a, &c = a;

   decltype(a) -->int

   decltype(*p) -->& 必须初始化

   decltype(c) -->&

   decltype(c+0) -->int

4. 给内部变量加上括号后，变量也会转化为表达式形式--返回引用形式

13. auto关键字

auto会忽略顶层const、引用

int a = 0, &b =a;

auto c = b; 此时c为一个int

14. explicit关键字

避免不合时宜的隐式类型转换

只能修饰含有一个参数的类构造函数

```
CxString string1(24);     // 这样是OK的
CxString string2 = 10;    // 这样是不行的，因为explicit关键字取消了隐式转换
```

只有最左侧参数没有默认值时，也可以用explicit

15. const关键字

1. 常成员函数

   在设计类的时候，要对不改变数据成员的成员函数后加const

   1. 有const修饰的成员函数（指 const 放在函数参数表的后面，而不是在函数前面或者参数表内）只能读取数据成员，不能改变数据成员，没有const修饰的成员函数，对成员可读可写
   2. 常量对象可以调用const成员函数，但是不能调用非const成员函数

16. 深拷贝与浅拷贝

1. 将对象与其所拥有的资源一起拷贝的称为深拷贝，必须定义拷贝构造函数才能实现深拷贝
2. 如果类有指针成员变量，一般都需要深拷贝，因为只有这样，才能将指针指向的内容再复制出一份来，让原有对象和新生对象相互独立，彼此之间不受影响。
3. 如果类成员变量没有指针一般不需要深拷贝。另外一种需要深拷贝的情况就是在创建对象时进行一些预处理工作。

17. 面向对象

封装、继承、多态

18. 空对象指针问题

1. 普通成员函数--无this -- √
2. 静态成员函数--无this -- √
3. 虚函数--需要this -- 崩溃 虚函数通过this指针动态连编实现多态
4. 普通成员函数--使用this，崩溃

19. memcpy和strcpy的区别

1. memcpy是复制一块内存，strcpy是复制字符串
2. memcpy严格遵循给定地址空间大小复制，strcpy不需要指定长度，自动判断'\0'

```
char * strCpy(char * strDest,const char * strSrc)   //[1]
{   //函数assert的头文件为#include<assert.h>
    assert((strDest != NULL)&&(strSrc != NULL));      //[2]

    char * strDestCopy=strDest;                    //[3]
    while ((*strDest++ = *strSrc++) != '\0');         //[4]

    return strDestCopy;                             //[5]
}
```

3. 返回char*为了链式调用
4. strncpy：如果源小于复制个数，需要补零

```
char * my_strncpy(char *strDest, const char *strSrc, int num)
  {
      assert((strDest != NULL) && (strSrc != NULL));
      //if (strDest == NULL || strSrc == NULL) return NULL;

       //保存目标字符串的首地址
      char *strDestcopy = strDest;
      while ((num--)&&(*strDest++ = *strSrc++) != '\0');
      //如果num大于strSrc的字符个数，将自动补'\0'
      if (num > 0)
      {
          while(--num)
          {
              *strDest++ = '\0';
          }
      }
      return strDestcopy;}
```

20. cout

cout的计算顺序是从右向左，cout的输出顺序是从左往右

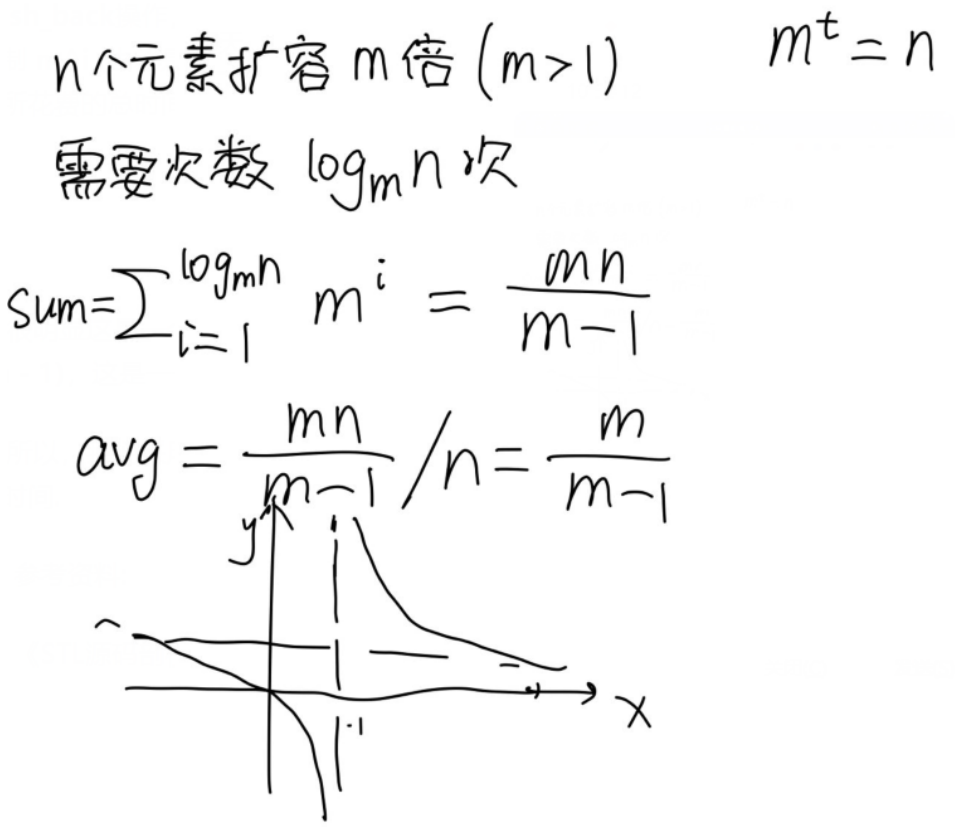
21. STL

1. STL包含六大组件：容器、迭代器、算法、仿函数、适配器、空间适配器
    1. 容器：数据结构，vector、list、deque

    2. 算法：sort、find。。。 algorithm
    3. 迭代器：容器和算法胶合剂
    4. 仿函数：重载小括号（）
    5. 适配器：修饰容器接口
    6. 空间配置器
2. 容器分类：
    1. 序列式容器
    2. 关联式容器
    3. 链式容器
3. 容器
    1. vector--线程安全，性能不如array
        1. vector达到扩容因子时，扩容至2倍（1.5倍）

           push_back线性增长的话，时间复杂度（O(n)）

           扩容两倍，分摊时间复杂度（O(1)）



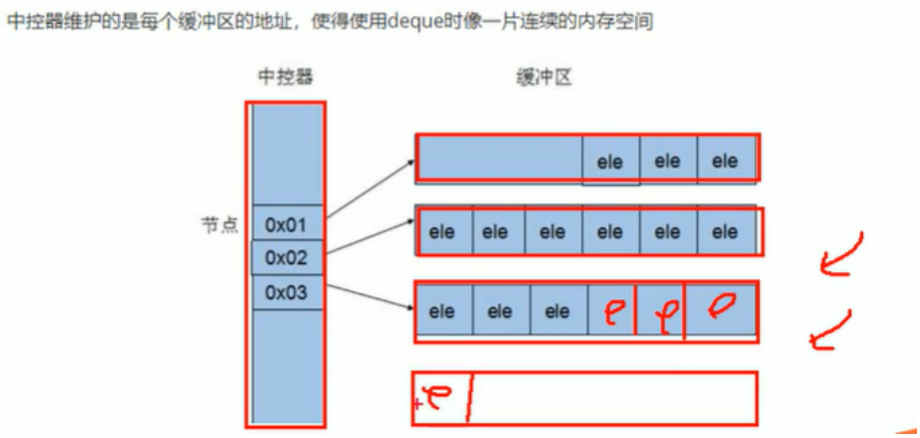
2. vector之间进行交换，v1.swap(v2)
3. vector头部插入效率低
4. 成倍增加扩容可以保证常量的时间复杂度，而指定大小的扩容是O(n)的时间复杂度
5. 在堆上分配空间

2. string

1. string是一个类，内部维护了一个char*
2. string每次扩容16B，sizeof(string) = 28B
3. int pos = str.find("xxx")　　find/rfind:从左向右/从右向左
4. string类初始化为空时，char*数组大小必须为1，放置'\0'
5.

3. deque

1. deque头部插入较快
2. 没有capacity，只有resize



4. stack：没有遍历行为

   pop push top empty

5. queue：只有队头、队尾可以访问

push pop back front empty

6. list
    1. STL的list是双向循环列表
    2. 采用动态存储，不会造成内存的浪费
    3. 不能随机访问，时间和空间开销大
    4. list插入和删除不会造成迭代器失效
7. set/multiset
    1. 插入后自动排序，set自动去重
    2. find返回迭代器，没找到返回.end()
    3. multiset和set的区别，multiset可以插入重复数据
    4. set的insert返回值：pair<iterator,bool>

       表示插入位置迭代器，是否成功插入

       multiset返回iterator
    5.
8. pair
    1. 创建：pair<string,int> p = make_pair("1",1);
9. map
    1. 所有元素都是pair，第一个为key，第二个为value
    2. find返回iterator,bool
10. *i是pair，i->first，*i.second
    3. 使用map[]，如果key不存在，自动调用insert
    4. 使用红黑树的优点
        1. 插入两次旋转、删除三次旋转。减少平衡的开销，可以进行频繁的插入和删除
        2. 红黑树的查找、插入、删除都是O(logn)
    5. 红黑树和哈希表的比较
        1. 红黑树的优点：有序性，操作时间复杂度稳定
        2. 红黑树的缺点：时间复杂度与n有关
        3. 哈希表的优点：查找添加删除时间复杂度常数级别
        4. 哈希表的缺点：不稳定，平均常数级，可能O(n)
11. resize

    改变容器内元素的数量
12. reserve

    改变capcity的值，需要进行赋值操作

4. 仿函数

在类中重载（），使用时像函数一样调用

```
class MyAdd
{
public:
    int operator()(int v1,int v2)
    {
        return v1 + v2;
    }

};

// 1、函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值
void test01()
{
    MyAdd myAdd;
```

5. 迭代器

提供对容器的访问接口，类模板表现得象指针，只是封装了原生指针，是指针概念的一种提升

可以在不暴露内部结构的前提下进行容器的遍历

1. 起始迭代器、尾后迭代器

   .begin() .end()

2. 迭代器*iterator *i表示<>里面的内容

3. 不支持随机访问的迭代器不能使用sort、reverse

   list等自带sort，reverse

4. 迭代器失效：

   序列式容器：插入/删除某个元素导致后面的迭代器均失效，这是因为vetor,deque使用了连续分配的内存，删除一个元素导致后面所有的元素会向前移动一个位置。所以不能使用erase(iter++)的方式

   erase会返回下一个迭代器

   关联性容器：插入不会导致迭代器失效，删除会使当前迭代器失效。这是因为map之类的容器，使用了红黑树来实现，插入、删除一个结点不会对其他结点造成影响。

   erase(iterator++)

   链表型容器：插入不会导致任何迭代器失效，删除指向位置失效

   既会返回下一迭代器，也可使用erase(iterator++)

5. set的迭代器是const的，map的迭代器可以修改val，但是不允许修改key。原因在于：红黑树修改不存在，必须摘下该节点，在重新插入，并进行平衡

5. 算法

   1. 使用for_each遍历

      for_each (nums.begin(), nums.end(), myPrint);

      myPrint(iterator)

6. allocator分配原始内存，并进行构造

   1. C++内存配置和释放的过程

      new：调用::operator new配置内存，调用对象构造函数构造对象内容，返回指针

      delete：调用对象析构函数，释放内存

   2. 使用allocator配置内存

      1. 内存配置：alloc::allocate()
      2. 内存释放：alloc::deallocate()
      3. 对象构造：::construct
      4. 对象析构：::destory

   3. 为了提高内存管理效率，STL会使用两级配置器，当超过128B时使用第一级空间配置器，小于128B时使用第二级空间配置器

      第一级空间配置器：>128B :使用malloc，realloc，free（）进行分配

      第二级空间配置器：<128B：使用内存池技术，通过空闲链表来管理

22. 智能指针

shared_ptr weak_ptr unique_ptr

将普通指针封装为栈对象，当栈的生明周期结束时，调用析构函数，释放掉已经申请的内存

引用计数：是线程安全的，计数加减是原子操作

指向对象的指针线程安全问题没有保障，引用对象安全无锁，但是对象的读写不是。在智能指针的拷贝过程中，先拷贝智能指针，再拷贝引用计数对象，这两个操作不是原子的。

mutex.lock()加锁

1. unique_ptr
    1. 保证同一时间只有一个智能指针指向该对象，允许临时变量赋值

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                          // #1 不允许
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You"));   // #2 允许
```

    2. 可以通过move、reset转移unique_ptr指向对象的所有权
    3. unique_ptr p2 = move(pointer);

       p2.reset(new xxx);

       auto ptr = make_unique();
    4. 无法进行复制构造和赋值操作，因为operator=被赋予了delete

```
int main()
{
    // 创建一个unique_ptr实例
    unique_ptr<int> pInt(new int(5));
    unique_ptr<int> pInt2(pInt);    // 报错
    unique_ptr<int> pInt3 = pInt;   // 报错
}
```

    5. 可以使用移动构造和移动赋值（move转移所有权）

```
unique_ptr<int> pInt(new int(5));
unique_ptr<int> pInt2 = std::move(pInt);    // 转移所有权
//cout << *pInt << endl; // 出错，pInt为空
cout << *pInt2 << endl;
unique_ptr<int> pInt3(std::move(pInt2));
```

    6. 但是可以从函数中返回一个unique_ptr
2. shared_ptr
    1. 多个智能指针可以指向同一个对象，最后一个引用被销毁时（引用计数为0）释放，使用.use_count()查看引用计数的值
    2. 使用shared_ptr也会存在内存泄露情况
        1. 使用指针分别赋值给两个智能指针，会导致重复释放问题

           int *p = new int(0);

shared_ptr point1(p);

shared_ptr point2(p);

两个指针分别开辟了自己的应用计数，p的应用计数为1，导致p会被释放两次，

修改：

1. 使用=重新构造，避免产生多个引用计数对象

2. 一个类要提供一个函数接口，返回一个指向当前对象的shared_ptr智能指针怎么办？方法就是**继承enable_shared_from_this类**，然后通过**调用从基类继承来的shared_from_this()方法**返回指向同一个资源对象的智能指针shared_ptr。

3. 两个对象的智能指针互相指向对方，造成循环引用，最终导致计数失效

3. shared_ptr的实现

```
template<typename T>
class smart {
private:
    T* _ptr;
    int* _count; //reference counting
public:
    //构造函数
    smart(T* ptr = nullptr) :_ptr(ptr) {
        if (_ptr) {
            _count = new int(1);
        }
        else {
            _count = new int(0);
        }
    }

    //拷贝构造
    smart(const smart& ptr) {
        if (this != &ptr) {
            this->_ptr = ptr._ptr;
            this->_count = ptr._count;

            (*this->_count)++;
        }
    }

    //重载operator=
    smart operator=(const smart& ptr) {
        if (this->_ptr == ptr._ptr) {
            return *this;
        }
        if (this->_ptr) {
            (*this->_count)--;
            if (this->_count == 0) {
                delete this->_ptr;
                delete this->_count;
            }
        }
        this->_ptr = ptr._ptr;
        this->_count = ptr._count;
        (*this->_count)++;
        return *this;
    }
```

```
    //operator*重载
    T& operator*() {
        if (this->_ptr) {
            return *(this->_ptr);
        }
    }

    //operator->重载
    T& operator->() {
        if (this->_ptr) {
            return this->_ptr;
        }
    }
    //析构函数
    ~smart() {
        (*this->_count)--;
        if (*this->_count == 0) {
            delete this->_ptr;
            delete this->_count;
        }
    }
    //return reference counting
    int use_count() {
        return *this->_count;
    }
};
```

4. 可以使用make_shared传入普通指针，也可以使用get函数获取指针

3. weak_ptr
    1. 可以从shared_ptr和weak_ptr进行构造，构造、析构不会引起计数的增加、减少，上面的循环引用替换为weak_ptr，解决内存泄露问题
    2. weak_ptr没有重载*和->，如果需要访问内容，需要.lock()转化为共享指针再使用

```
shared_ptr<B> p = pa->pb_.lock();
p->print();
```

23. this指针

1. 调用函数的时候，含有默认参数this
2. 如果为空，调用成员变量引发异常
3. 成员参数在使用时默认调用隐含变量this

24. 堆的大小受寻址空间限制

32位系统的地址空间为4G，malloc超过时返回为0

25. 赋值构造函数

赋值构造函数禁止传值，形参传进来后会调用构造函数形成参数，会无休止的调用直到栈溢出

26. 右值引用

左值：可以取地址的变量，返回值为引用的函数

右值：没有变量名，不可修改（无名变量等等）

右值引用：

如果右值引用传入Data&& 的重载，那么会调用到const引用

std::move将一个对象强制转化为右值返回，不创建新的对象

27. C++的短路运算

if判断的时候如果前面的表达式已经满足/不满足new

后面的判断将不会执行

28. 类成员分布

1. static成员变量：不属于静态成员内存的部分，该类共享静态变量
2. static成员函数：没有this指针，在代码区

29. C++11的新特性

1. 右值引用

   避免无谓的复制，提高程序性能

   给不具名变量起了个别名

   1. 左值、右值

      左值：表达式结束后仍然存在的持久化对象--可对表达式取值

      右值：表达式结束时就不再存在的临时对象--不可对表达式取值

   2. 右值引用与左值引用

      1. 左值引用（引用）--C++98中出现：给变量取别名
      2. 右值引用：使用&&符号，给右值赋予生命周期（与右值引用符号生命周期相同）

   3. 左值引用只能绑定左值，右值引用只能绑定右值

   4. 常量左值引用：可以绑定非常量左值、常量左值、右值，只能读不能改

```
const int & a = 1; //常量左值引用绑定 右值， 不会报错

class A {
  public:
    int a;
};
A getTemp()
{
    return A();
}
const A & a = getTemp();    //不会报错 而 A& a 会报错
```

   5. 理论上，实际中一次构造函数都不会调用，存在编译器优化问题

```
class Copyable {
```

```
public:
    Copyable(){}
    Copyable(const Copyable &o) {
        cout << "Copied" << endl;
    }
};
Copyable ReturnRvalue() {
    return Copyable(); //返回一个临时对象
}
void AcceptVal(Copyable a) {

}
void AcceptRef(const Copyable& a) {

}

int main() {
    cout << "pass by value: " << endl;
    AcceptVal(ReturnRvalue()); // 应该调用两次拷贝构造函数1
    cout << "pass by reference: " << endl;
    AcceptRef(ReturnRvalue()); //应该只调用一次拷贝构造函数2
}
```

1：returnRvalue中，构造一个对象，然后返回的时候调用拷贝构造函数生成临时对象，调用acceptval时，将这个临时对象返回给局部变量a

2：acceptref（）：形参为左值引用，能够直接接受一个右值，不需要构建临时变量作为右值

6. 常量左值引用可以绑定一个右值，减少一次拷贝（非常量的左值引用可能会失败）
7. 左值引用， 使用 `T&`, 只能绑定**左值**

   右值引用， 使用 `T&&`， 只能绑定**右值**

   常量左值， 使用 `const T&`, 既可以绑定**左值**又可以绑定**右值**

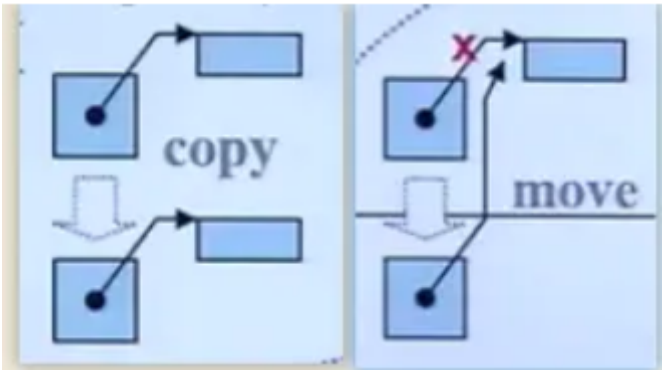
   已命名的**右值引用**，编译器会认为是个**左值**

   编译器有返回值优化，但不要过于依赖
8. 全局引用--通用引用：由初始化时绑定的值的类型确定

   使用T&&既可以传左值，又可以传右值

9. 移动构造与移动赋值

   使用临时变量进行移动构造函数时



10. 如果没有提供移动构造函数、只提供了拷贝构造函数，类传入std：：move()会失效但不会发生错误，因为拷贝构造函数的参数const T&也可以接受常量左值引用，但是会发生原始类销毁时，赋值类也无法构造

11. 如果是模板类

    即使定义了左值引用，也会发生一些问题，T&&可能是string&&，也可能是string&&&

    引用折叠规则：所有的右值引用叠加到右值引用上仍然为右值引用

    所有其它类型的叠加都变成左值引用

    最终变为：左值传入就是左值引用，右值传入就是右值引用

12. 完美转发

    函数参数交给另一个函数处理时，参数如果还能保持原有的参数特征，就是完美转发（左值右值类型不变）

    提供forward（）函数解决完美转发的问题

30. C++11的新特性

1. 增加了关键字auto、decltype
2. 容器的列表初始化
3. hash的unordered_map、unordered_set正式加入STL
4. default加入，生成默认的构造函数

```
class A11 {
    int data;
public:
    A11() = default;
     ~A11() = default;
    A11 (int _data):data(_data) {}
};
```

5. delete的加入，不需要再向以前一样需要放入private
6. 右值引用的加入

    c++11的std::move, 解决的问题是一个复制效率的问题: 必须保证以前左值不再需要了

    对临时变量(如函数中的参数)的复制，通过更改对象的所有者(move)，实现免内存搬迁或拷贝(去除深拷贝)，

    提高"复制"效率(其实不是复制，仅是更改了对象的所有者。

    1、减少内存复制成本

    2、将不再需要的变量，取消它对原先持有变量(内存)的持有(修改)权限

7. 智能指针
8. 多线程threadpool的库
9. lambda表达式
10. 可变参数模板

    template <typename T, typename... Types>

    void print(const T& firstArg, cosnt Typtes& ...args)

    递归使用args，循环调用自己

    最后需要重载一下空参数的函数

```
template <typename T, typename... Args>
void print(T head, Args... rest)
```

11. NULL、nullptr、nullptr_t
    1. NULL：宏定义0（C中为空指针）

       潜意识认为NULL为一个指针类型，参数中使用0更好
    2. nullptr（nullptr_t的对象）

       nullptr时nullptr_t类型的常量，nullptr_t定义了转到任意指针类型的转换操作符

       表示一个空指针
    3. 由于存在重载，NULL可能会产生调用函数的问题

       ```
       test(classA a,classB *pb);//函数1
       test(classA a,int i);//函数2
       test(a,NULL);//调用的是函数2
       ```

    4. nullptr_t表示nullptr的类型
12. 两个>>不再需要添加空格

    ```
    vector<list<int> >;   // OK in each C++ version
    vector<list<int>>;    // OK since C++11
    ```

13. auto自动判断参数类型

    变量类型过长/过复杂
14. 初始化使用成员列表/{}符号

    编译器看到成员列表，编译器做出initializer_list

    关联到array，然后逐一分配给函数

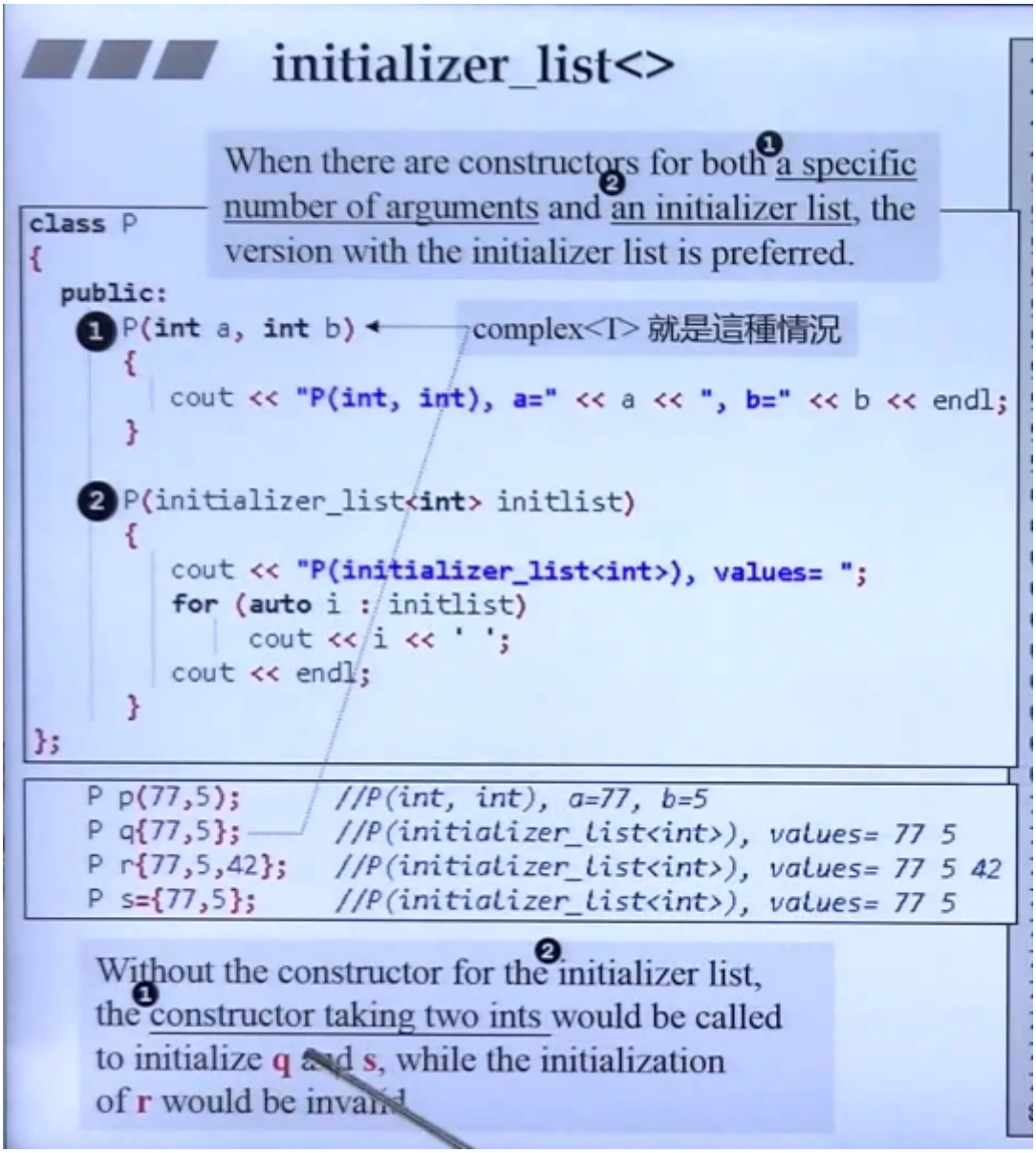
    如果函数参数存在initializer_list，则直接move过去

    （每一种容器都有initializer_list的传入参数）
15. 不能窄化初始化

    有的warning，有的error

    ```
    int x3{5.0};      // ERROR:
    int x4 = {5.3};   // ERROR:
    ```

16. initializer拆解问题



17. explicit

主要用于构造函数

```
struct Complex
{
   int real, imag;

   explicit
   Complex(int re, int im=0) : real(re), imag(im)
   { }

   Complex operator+(const Complex& x)
   { return Complex((real + x.real),
                    (imag + x.imag)); }
};

Complex c1(12,5);
Complex c2 = c1 + 5;
```

```
class A {
    explicit A(int a, int b, int c)
    {
        ;
    }
    A( initializer_list<int> init )
    {
        ;
    }
};

int main()
{
    A a = { 1,2,3 };
}
```

18. for迭代的新用法

```
for(auto i : vec){;            auto&i : vec
}
```

19. =default =delete

    如果写了带参数的构造函数，将其他的=default就会提供默认的

    =delete会将其删除

31. 单例模式

1. 单例模式：全局只存在一个唯一的类实例，具有全局变量的特点

   应用场景：设备管理器、数据池

2. 全局只有一个实例，禁止用户自己定义实例（构造定义为private）
3. 有缺陷的懒汉式

```
class Singleton{
private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    Singleton(Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Singleton* m_instance_ptr;
public:
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    static Singleton* get_instance(){
        if(m_instance_ptr==nullptr){
              m_instance_ptr = new Singleton;
        }
        return m_instance_ptr;
```

```
    }
};

Singleton* Singleton::m_instance_ptr = nullptr;
```

线程安全内存安全的懒汉式--双重检查锁可能会失效

```
class Singleton{
public:
    typedef std::shared_ptr<Singleton> Ptr;
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    Singleton(Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Ptr get_instance(){

        // "double checked lock"
        if(m_instance_ptr==nullptr){
            std::lock_guard<std::mutex> lk(m_mutex);
            if(m_instance_ptr == nullptr){
              m_instance_ptr = std::shared_ptr<Singleton>(new Singleton);
            }
        }
        return m_instance_ptr;
    }


private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
    static Ptr m_instance_ptr;
    static std::mutex m_mutex;
};

// initialization static variables out of class
Singleton::Ptr Singleton::m_instance_ptr = nullptr;
std::mutex Singleton::m_mutex;
```

最终的懒汉式

```
class Singleton
{
public:
    ~Singleton(){
        std::cout<<"destructor called!"<<std::endl;
    }
    Singleton(const Singleton&)=delete;
    Singleton& operator=(const Singleton&)=delete;
    static Singleton& get_instance(){
        static Singleton instance;
        return instance;

    }
```

```
private:
    Singleton(){
        std::cout<<"constructor called!"<<std::endl;
    }
}
```

使用的是static magic特性：当变量初始化的时候，并发同时进入声明语句。并发线程会阻塞到初始化结束

在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。

（判断的标志，在该全局变量附近分配一块空间，表名该空间是否被初始化）

```
class single {
private:
    static int count;
    single() { count += 1; };
    single(const single&) = delete;
public:
    static single& getinstance()
    {
        static single sig;
        return sig;
    }
};
int single::count = 0;
int main()
{
    single& singleton = single::getinstance();
    single& singleton2 = single::getinstance();
    return 0;
}
```

32. 如何判断一段程序是由C编译还是C++编译

```
#ifdef __cplusplus
    cout<<"C++";
#else
    cout<<"C";
#endif
```

33. C语言字符串函数

1. strcpy遇到'\0'才结束
2.

```
void test2()
{
 char string[10], str1[10];
 int i;
 for(i=0; i<10; i++)
 {
 str1  = 'a';
 }
strcpy( string, str1 );
}
```

考答案

首先，代码根本不能通过编译。因为数组名str1为 char *const类型的右值类型，根本不能赋值。
再者，即使想对数组的第一个元素赋值，也要使用 *str1 = 'a';
其次，对字符数组赋值后，使用库函数strcpy进行拷贝操作，strcpy会从源地址一直往后拷贝，直到遇到'\0'为止。所以拷贝的长度是不定的。如果一直没有遇到'\0'导致越界访问非法内存，程序就崩了。

3. strlen不会计算末尾的'\0'
4. 完整的字符串拷贝函数

```
char* my_strcpy(char* dst, char* src)
{
    assert(dst!=NULL&&stc!=NULL);
    char* result = dst;       //为了保证链式操作
    while(*dst++ = *src++)!='\0');
    return result;
}
```

34. double、float型变量判断

不能直接相等！！！

if(a-b<1e-6)

35. define的副作用

当A为*p++时，其会加两次

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

36. 头文件重复引用

1. #ifndef xxx
   #define xxx
   #endif
2. #pragma once

37. C++的四种cast转换

向上转化：派生类向基类转化

向下转化：基类向派生类

1. const_cast：将const变量转化为非const
2. static_cast：用于隐式转换，非const转const，void*转指针
   可以用于多态向上转化，向下转化可以成功但是结果未知

3. dynamic_cast：用于动态类型转化，只能用于含有虚函数的类，

   用于类层次间的向上向下转化（只能指针/引用），向下转化非法返回NULL

4. reinterpret_cast

   什么都可以转，int转指针，但是可能会有问题

5. 为什么不用C的强转

   因为C的强转表面上看起来功能强大，但是转化不够明确，不能继续错误检查

38. C++指针

1. 指针和引用的区别
   1. 引用必须赋初值（左值引用的初始化且必须是永久变量，不能是临时变量），指针可以初始化为NULL
   2. 指针存在分配空间，而引用是变量的别名
   3. sizeof(指针) = 4

      sizeof(引用) = 引用体的原始大小
   4. 存在const指针，不存在const引用
   5. 指针可以更换指向的对象，引用是变量的别名不可更换
   6. 指针可以多级int**，但是引用只能一级
   7. 指针和引用的++运算符意义不同
   8. 返回动态内存分配对象必须用指针
2. C++数组和指针的区别
   1. 指针：保存数据的地址

      通过malloc分配，free释放
   2. 数组：保存数据

      存放在栈中，隐式分配与删除
3. 野指针：未初始化的指针，指向内容不确定

   指向一个被删除/无访问授权的地址

39. 函数指针

1. 指向函数的指针变量

   即指向函数的入口地址，可以使用该指针变量调用函数

```
int(*p)(int, int);  //定义一个函数指针
int a, b, c;
p = Max;  //把函数Max赋给指针变量p, 使p指向Max函数
printf("please enter a and b:");
scanf("%d%d", &a, &b);
c = (*p)(a, b);  //通过函数指针调用Max函数
printf("a = %d\nb = %d\nmax = %d\n", a, b, c);
return 0;
```

40. C++析构函数与构造函数

1. 构造函数与析构函数相同，都不存在返回值
2. 和构造函数不同，析构函数不能有参数，只有一个且不能重载
3. 系统生成的析构函数：如果没有显示指定析构函数，系统生成一个析构函数，如果自定义了，编译器在执行时也会先调用自定义的析构函数再调用合成的析构函数，不执行任何操作
4. 如果存在动态分配内存，那么析构函数一定要进行释放

41. 内存池技术

1. 程序一次性分配调用一块内存作为内存池，分为单线程内存池和多线程内存池，多线程内存池在分配和释放时需要加锁
2. malloc的原理
    1. malloc用于动态分配内存，采用内存池的方式，先申请大块内存作为堆区，然后分配为小块
    2. 使用隐式链表管理块，使用显示链表管理空闲块
3.

42. C++的struct和class的区别

1. class默认private，struct默认public
2. 二者都可以继承
3. class可以定义模板类

43. 常见段错误

1. 使用了野指针
2. 试图修改字符串常量的内容
3. 动态内存分配时，写入超过了数组的范围，导致空闲链表的头被覆盖，引用空闲链表时出现错误

44. 常见内存泄漏

1. 堆内存泄露：使用malloc没有进行free
2. 系统资源泄露：使用了bitmap、handle、socket没有进行释放
3. 没有将基类的析构函数定义为虚函数，导致子类的内存泄漏
4. 使用了shared_ptr相互指向

45. 指针函数与函数指针

1. 指针函数：本质是一个函数，返回值是一个指针

   int * func_sum(int n)

   局部变量存放在栈区，函数结束自动释放，如果返回一个局部变量的地址

   那么第一次可以成功，第二次就会报错

2. 函数指针：本质是一个指针，指向了一个函数的入口地址

   int (*f) (int,int);

   f = func;

3. 函数指针多用于回调函数

   通过一个指针函数调用的函数，函数指针作为一个参数传给函数

   可以实现函数执行的多态化

46. lambda表达式--匿名函数

可能某个函数只会被调用一次

完成的是数学公式

```
int z =
    [] (int x,int y) ->int mutable {
        return x+y; //函数主体
    }(1,2);
[]:捕获列表，内部的表达式需要用到的外面的值
    1. 按值捕获（默认）--只复制值，而且在声明定义函数的时候进行值捕获
    2. 按引用捕获[&t]:匿名函数内部和外部使用的是相同的对象
    3. 按值捕获外界所有变量[=]
       按引用捕获外界所有变量[&]
    4. 值和引用分别捕获[x,y,&c,&d]
():参数列表
->:返回值
{}:函数主体
():最后的（1，2）表示当前输入

mutable:匿名函数无法修改捕获列表中的值，是const类型。但是增加mutable关键字，表示捕获列表中的值可以进行修改，但是无法影响原始数据（默认按值捕获，内部的变量捕获的时候就和外部独立了，地址会变化），只影响当前的匿名函数

//多次使用lambda表达式
auto f = [](int x,int y)->int{return x+y;};
f(1,2)
```

lambda表达式只与参数有关，一定的输入只对应相应的输出，称之为函数式编程，对多线程并发有较高的优势

嵌套auto计算f(1)(2)

47. C/C++的区别

1. C++是面对对象的编程语言，C是面向过程的编程语言
2. C++面向对象的特性：封装、继承、多态

   C++的封装相较于结构体：

   1. 增加了权限控制
   2. 使接口、函数调用更加突出，隐藏数据实现细节
   3. 数据和函数分离
3. C++相较于C，多了很多类型安全的功能--强制类型转换
4. C++支持泛型编程，比如模板类，函数模板

48. C++默认提供的类函数

1. 默认构造函数(无参构造函数)

   无参构造、有参构造

   如果没有显示定义，提供无参构造函数，一旦实现构造函数不在自动生成

   无参的构造函数和全缺省的构造函数都称为默认的构造函数，并且默认的构造函数只能有一个，（无参构造函数，全缺省构造函数，编译器生成的构造函数都认为是默认成员函数）

2. 析构函数，释放在堆区申请空间的数据成员的指针
3. 拷贝构造函数(浅拷贝)，对象中数据成员进行值赋值

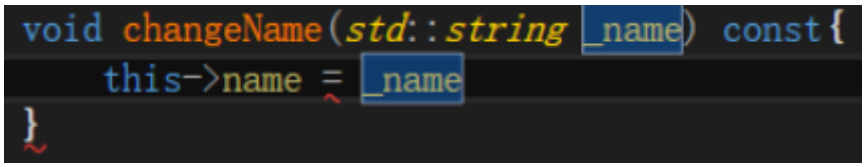
   特点1：当类中写了有参构造函数，编译器不会再提供默认构造函数，但会提供拷贝构造函数；
   特点2：当类中写了拷贝构造函数，编译器不会再提供其他构造函数(无参构造和有参构造函数，普通构造函数)*

   使用等号，接收者未初始化时调用拷贝构造函数
4. 赋值操作符的重载

   使用等号，接收者已初始化时调用赋值构造函数
5. 取地址操作符的重载

   取地址以及const取地址一般取出this指针
6. const修饰的取地址操作符的修改

   const修饰this指针，表示该成员函数不能对类成员进行修改



49. 使用初始化列表的好处

1. 类成员中存在常量，只能初始化不能赋值
2. 类成员中存在引用，只能初始化不能赋值
3. 提高效率

```
template<class T>
class NamedPtr {
public:
    NamedPtr(const string& initName, T *initPtr);
    ...
private:
    const string& name; // 必须通过成员初始化列表进行初始化
    T * const ptr; // 必须通过成员初始化列表进行初始化
};
```

50. 传值和传引用的区别

51. 构造函数的const、static

构造函数不能用const修饰，因为const修饰的是this指针，加上const之后无法对对象的值进行赋值

构造函数也不能是static，因为静态函数没有this指针

顶层const不能重载，底层const可以重载

52. GCC额外项使得先于main执行

`__attribute((constructor))`是gcc扩展，标记这个函数应当在main函数之前执行。同样有一个`__attribute((destructor))`，标记函数应当在程序结束之前（main结束之后，或者调用了exit后）执行

53. 函数传入传出

1. 压栈顺序-从右向左
2. 传出：生成一个临时变量，将引用作为函数参数传入函数内

54. C++那些事

1. const

   const会进行安全检查，#define只是简单的字符串替换

   1. 可以防止修改、起到保护作用
   2. 节省空间，避免不必要的空间分配：const常量给出地址，define需要拷贝多份
   3. const对象默认文件局部变量

      如果需要将const常量引用，需要显示声明extern

      ```
      //extern_file1.cpp
      extern const int ext=12;
      //extern_file2.cpp
      #include<iostream>
      /**
       * by 光城
       * compile: g++ -o file const_file2.cpp const_file1.cpp
       * execute: ./file
       */
      extern const int ext;
      int main(){
          std::cout<<ext<<std::endl;
      }
      ```

   4. const指针必须进行初始化，因为常量不可更改
   5. const *代表：const修饰指针指向的变量，* const代表修饰指针本身
   6. 必须使用const void* 保存const对象的地址

2. static

   1. 变量声明为static，会在程序的生命周期内分配，多次调用也只分配一次
   2. 类的静态成员不能使用构造函数初始化

3. this指针（解析自引用的函数参数）

   1. 不是对象的一部分，不能使用sizeof定位
   2. 当非静态成员函数访问非静态成员变量时，添加隐含的this指针
   3. 非静态成员函数返回类对象本身时，使用return *this
   4. this指针在非const解释为A* const this

      常函数解释为const A* const
   5. 在成员函数执行前构造，执行结束后清楚
   6. 指向第一个成员变量的地址

      例如int x
   7. 静态成员没有this指针，不能定义为虚函数

4. inline（与define的区别，define预编译，inline编译）

   1. 编译器对inline的处理步骤

    1. 将inline函数体复制到inline掉点处
    2. 为inline分配栈空间（变量）
    3. 映射输入参数、返回值
2. 不宜内联（.o文件过大可能导致编译器不进行内联操作）
    1. 函数体内代码过长，内联导致内存消耗代价大
    2. 函数体内出现循环，函数体内执行时间比调用高
3. 虚函数可以是内联函数吗
    1. 虚函数可以为内联函数，但是表现多态的时候不能内联
    2. 内联为编译器建议，虚函数多态性在运行时表现，编译器无法知道运行期调用哪个代码，因此不可以内联（编译器会忽略inline关键字）
    3. 虚函数唯一可以内联：编译器直到所调用的对象的类：Base：：xxx（），只有在具有实际对象时才会发生

5. sizeof大小
    1. 空类的大小为1
    2. 虚函数、成员函数、**静态数据成员**（static int）不占用类对象的存储空间
    3. 包含虚函数的类，无论多少，都只包含一个vtpr虚函数指针的大小
    4. 普通继承，派生类继承所有的对象，按照字节对齐计算大小
    5. 虚函数继承，继承vtpr，32位4字节，64位8字节
    6. 虚继承：继承基类的vptr
6. 纯虚函数和抽象类

   声明赋值为0即可实现纯虚函数
    1. 实现抽象类：成员函数内可以调用纯虚函数，构造、析构内部不能使用纯虚函数
    2. 如果一个类从抽象类派生而来，它必须实现了基类中的所有纯虚函数，才能成为非抽象类。
7. 虚函数
    1. 虚函数体是动态绑定的，但是虚函数参数是静态绑定的。默认参数的使用需要看指针或者引用本身的类型，而不是对象的类型。
    2. 静态成员函数：不可以为虚函数、const、volatile

       static没有this，定义virtual没有意义
    3. 构造函数不可以为虚函数，除了inline和explicit，构造函数也不允许有其他关键字
    4. 尽管虚函数表vtable是在编译阶段就已经建立的，但指向虚函数表的指针vptr是在运行阶段实例化对象时才产生的。 如果类含有虚函数，编译器会在构造函数中添加代码来创建vptr。 问题来了，如果构造函数是虚的，那么它需要vptr来访问vtable，可这个时候vptr还没产生。 因此，构造函数不可以为虚函数。
8. volatile
    1. 对volatile修饰的变量进行读写操作时，会引发副作用
    2. 编译器可能对无用循环进行缩减，但是如果发生对端口的读写，就会发生错误

       中断函数处理内容，可能编译器判断没有事件发生

```
static int i=0;

int main()
{
    while(1)
    {
    if(i) dosomething();
    }
}
/* Interrupt service routine */
```

```
void IRS()
{
    i=1;
}
```

3. volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
4. volatile 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 volatile 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
5. const 可以是 volatile （如只读的状态寄存器）
6. 指针可以是 volatile

9. assert：宏而非函数

   可以动过#define NDEBUG关闭assert

10. 位域

    1. 不能使用&作用位域，指针无法指向类的位域
    2. unsigned xxx xxx：x： 一个xbit的位域
    3. 一般在结构体中使用

```
struct _PRCODE
{
    unsigned int code1: 2;
    unsigned int cdde2: 2;
    unsigned int code3: 8;
};
struct _PRCODE prcode

prcode.code1 = 0;
prcode.code2 = 3;
procde.code3 = 102;
```

    4. 可以命名空名称位域使其作为填充作用
    5. 位域的对齐规则（结构体的总大小）

       如果当前位域成员超过了unsigned int的边界（32bits），那么将其放置到下一个

       使用宽度为0的未命名位于成员令下一位域成员与下一整数对齐

```
struct stuff
{
    unsigned int field1: 30;
    unsigned int       : 2;
    unsigned int field2: 4;
    unsigned int       : 0;
    unsigned int field3: 3;
};
```

    6. 位域初始化
        1. 成员初始化列表
        2. 成员定位符
        3. 将其映射到整性地址，然后进行赋值

```
1. struct stuff s1= {20,8,6};

2. struct stuff s1;
    s1.field1 = 20;
    s1.field2 = 8;
    s1.field3 = 4;

3. int* p = (int *) &b1;  // 将 "位域结构体的地址" 映射至 "整形（int*) 的地址"
*p = 0;                   // 清除 s1，将各成员归零
```

11. extern

    extern “C”用于C++链接在C语言模块中定义的函数

    1. C++与C的函数编译后产生的符号并不相同（C++支持重载，函数名包含参数）
    2. 让C++语句去找_add这类的C函数符号
    3. 在C中调用C++不支持extern "C"需要包含在cpp的头文件中

    （1）C++调用C函数：

```
//xx.h
extern int add(...)

//xx.c
int add(){

}

//xx.cpp
extern "C" {
    #include "xx.h"
}
```

    （2）C调用C++函数

```
//xx.h
extern "C"{
    int add();
}
//xx.cpp
int add(){

}
//xx.c
extern int add();
```

12. struct

    1. C语言中的struct

       C中struct只能将数据成员放在里面，不能放置函数

       C中struct不能防止访问修饰符

       C中struct名字可以和函数同名

       C中struct不能继承

       C中定义struct必须加struct

          stuct stuk stk;

    2. C++中的struct

C++结构体中不仅可以定义数据，还可以定义函数。

C++结构体中可以使用访问修饰符，如：public、protected、private 。

C++结构体使用可以直接使用不带struct。

C++继承

若结构体的名字与函数名相同，可以正常运行且正常的调用！但是定义结构体变量时候只用用带struct的

```
struct Student {

};
Student(){}
Struct Student s; //ok
Student s;  //error
```

C	C++
不能将函数放在结构体声明	能将函数放在结构体声明
在C结构体声明中不能使用C++访问修饰符。	public、protected、private 在C++中可以使用。
在C中定义结构体变量，如果使用了下面定义必须加struct。	可以不加struct
结构体不能继承（没有这一概念）。	可以继承
若结构体的名字与函数名相同，可以正常运行且正常的调用！	若结构体的名字与函数名相同，使用结构体，只能使用带struct定义！

13. struct和class的区别

    两者默认访问权限不同：struct是public访问权限，class是private访问权限

14. union

    联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

    - 默认访问控制符为 public
    - 可以含有构造函数、析构函数
    - 不能含有引用类型的成员
    - 不能继承自其他类，不能作为基类
    - 不能含有虚函数
    - 匿名 union 在定义所在作用域可直接访问 union 成员
    - 匿名 union 不能包含 protected 成员或 private 成员
    - 全局匿名联合必须是静态（static）的

15. C++多态

    C++中维护一张虚函数表，父类的指针指向子类的对象，则该指针会调用子类重写过的虚函数表

    C实现多态

    1. 封装struct
    2. 继承：结构体嵌套
    3. 多态：类与子类的函数指针不同

16. explicit

    - explicit 修饰构造函数时，可以防止隐式转换和复制初始化
    - explicit 修饰转换函数时，可以防止隐式转换，但按语境转换除外

17. friend

    普通函数/成员函数访问类私有/保护成员的方式

    1. 友元函数

       普通函数对一个访问某个类中的私有/保护成员

       友元类

       类A中的成员函数访问B中的私有/保护成员

       优点：提高了程序运行效率

       缺点：破坏了封装性

    2. 友元函数无继承性

       类B是类A的友元，类C继承于类A，那么友元类B是没办法直接访问类C的私有或保护成员。

    3. 友元函数五传递性

       类B是类A的友元，类C是类B的友元，那么友元类C是没办法直接访问类A的私有或保护成员，也就是不存在“友元的友元”这种关系。

18. using

    1. 改变访问性

       using Base::n可以改变访问权限

    2. 将所有重载实例加载到派生类的作用域中

```
class Base{
    public:
        void f(){ cout<<"f()"<<endl;
        }
        void f(int n){
            cout<<"Base::f(int)"<<endl;
        }
};

class Derived : private Base {
    public:
        using Base::f;
        void f(int n){
            cout<<"Derived::f(int)"<<endl;
        }
};
```

    3. using B = A取代typedef

19. ::

    1. 全局作用域符::name：表示作用域为全局命名空间
    2. 类作用域符class::name：表示指定类型作用域的范围
    3. 命名空间作用域符namespace::name：表示特定作用域命名空间

20. enum

    作用域不受限，容易引起命名冲突

```
enum expression{excited,blue};
enum weather{sun,cloud,rain,blue};
```

    1. 加入命名空间进行限制

但由于命名空间可以被扩充内容，所以还是无法避免重复现象

```
namespace expression {
    enum expression { excited, blue };
}

enum weather{sun,cloud,rain,blue};
```

2. 使用类/结构体来限定作用域

```
struct Color1
{
    enum Type
    {
        RED=102,
        YELLOW,
        BLUE
    };
};
```

3. 枚举类

   取出枚举类内容必须使用static_cast进行强转

4. 枚举常量不会占用对象的存储空间，编译时被全部求值

   枚举常量的缺点：隐含数据类型为整数、最大值有限

21. decltype

    decltype的作用为：查询表达式的类型

    1. 可以使用匿名类型

```
struct
{
    int d ;
    doubel b;
}anon_s;
decltype(anon_s) temp;
```

    2. 泛型编程中结合auto，追踪函数的返回值类型
    3. decltype(e)的返回结果
        1. 如果e为没带括号的标记符表达式、类成员访问表达式，e表示实体类型
        2. e为被重载的函数，编译错误
        3. 若e类型为T，且为将亡值，decltype(e) 为T&&
        4. e的类型是T，如果e是一个左值，那么decltype（e）为T&

```
//规则三：左值，推导为类型的引用。

decltype ((i))var6 = i;      //int&

decltype (true ? i : i) var7 = i; //int&  条件表达式返回左值。

decltype (++i) var8 = i; //int&  ++i返回i的左值。

decltype(arr[5]) var9 = i;//int&. []操作返回左值

decltype(*ptr)var10 = i;//int& *操作返回左值

decltype("hello")var11 = "hello"; //const char(&)[9]  字符串字面常量为左值，且为const左值。
```

5. int arr[4]; decltype(arr) -> int

22. 引用与指针

1.

引用	指针
必须初始化	可以不初始化
不能为空	可以为空
不能更换目标	可以更换目标

2. 引用

   1. 左值引用：表示对象的身份
   2. 右值引用：必须绑定到右值的引用（临时对象、将亡值）

      右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

      1. 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
      2. 能够更简洁明确地定义泛型函数。
   3. 引用折叠

      - `X& &`、`X& &&`、`X&& &` 可折叠成 `X&`
      - `X&& &&` 可折叠成 `X&&`

3. 使用const 引用作为只读形参，可以避免参数拷贝、获得与传值相同的调用方式
4. 引用型返回值

   重载某些操作符时，使用引用型返回值可以进行连续调用

```
vector<int> v(10);
v[5] = 10;     //[]操作符返回引用，然后vector对应元素才能被修改
               //如果[]操作符不返回引用而是指针的话，赋值语句则需要这样写
*v[5] = 10;    //这种书写方式，完全不符合我们对[]调用的认知，容易产生误解
```

5. 引用只是指针操作的语法糖，在底层操作方法完全相同

23. 宏

    1. 字符串化操作符#

       预处理器将该参数转化为字符串数组

       忽略传入参数名前面和后面的空格

       ```
       #define exp(s) printf("test s is:%s\n",s)
       string str = exp2(   bac );
       cout<<str<<" "<<str.size()<<endl;

       输出：
       bac 3
       ```

       传入参数名存在空格时，会自动连接子字符串，并添加一个空格链接，忽略其他空格

       exp2(asda bac); size = 8

    2. 符号链接符##

       分隔链接方式，先分隔，然后强制链接，将多个形参转化为一个实际参数名

    3. 续行操作符

       定义的宏不能用一行表达完整，使用\表示下一行继续、

    4. 宏展开：一定要加（） / {}

       ```
       #define fun() {f1();f2();}
       if(a>0)
           fun();
       // 宏展开
       if(a>0)
       {
           f1();
           f2();
       };
       ```

    5. 使用do while避免goto

    6. 避免空宏的警告

       #define EMPTYMICRO do{}while(0)

55. 查看C++版本

存在名字叫__cplusplus的宏，

直接cout输出查看即可

有201103L和199711L两种

56. 三五法则

1. 需要析构函数的类也需要拷贝构造函数和拷贝赋值函数

   合成析构函数不足以释放资源，需要额外的

2. 需要拷贝操作的类也需要赋值操作

3. 析构函数是不能删除的

4. 如果一个类有删除的/不可访问的析构函数，默认/拷贝构造函数未删除的

5. 如果一个类有const或引用成员，不能使用合成的拷贝赋值操作

57. C++不能被继承的类

1. 构造函数和析构函数都是private

   这样派生类无法构造父类（编译器会报错）

2. A中声明B的友元类

   那么B可以调用A的private里面的构造和析构函数

   如果此时C继承B，但是C不是A的友元类，那么不能构造基类

   其中B与A必须是虚继承：这样C才必须担负起构造A的责任，才会出现不能构造的情况

```
class A
{
public:
    int c;
private:
    A() { c = 0; };
    friend B;
    ~A() {};
};

class B : public A
{
public:
    void print() {
        cout << c;
    }

};
class C :public B
{

};
```

58. 继承的对象

1. 静态成员函数、非静态成员函数

   可以被继承

2. 构造函数、赋值操作函数不能被继承

59. 仿函数

函数指针不能满足STL对抽象性的要求

1. 仿函数是重载类的()函数，创建一个行为类似函数的对象
2. 可以实现类似函数调用的过程所以叫做仿函数
3. 由于只有一个重载函数，sizeof(A) = 1B

4.
```
template<typename T>
class function {
public:
    T operator()(const T& a, const T& b) const
    {
        return a + b;
    }

};

function<int> a;
cout << a(1, 2);
```

5. 满足STL算法中的抽象性

   例如count_if

   先考虑一个简单的例子：假设有一个vector<string>，你的任务是统计长度小于5的string的个数，如果使用count_if函数的话，你的代码可能长成这样：

```
1 bool LengthIsLessThanFive(const string& str) {
2     return str.length()<5;
3 }
4 int res=count_if(vec.begin(), vec.end(), LengthIsLessThanFive);
```

   其中count_if函数的第三个参数是一个函数指针，返回一个bool类型的值。一般的，如果需要将特定的阈值长度也传入的话，我们可能将函数写成这样：

```
1 bool LenthIsLessThan(const string& str, int len) {
2     return str.length()<len;
3 }
```

   但是count_if只能够接受带有一个参数的函数

   要改进：

   1. 使用全局变量length

      没有可扩展性，容易出错，代码很复杂

   2. 使用仿函数+成员变量

```
class ShorterThan {
    public:
        explicit ShorterThan(int maxLength) : length(maxLength) {}
        bool operator() (const string& str) const {
            return str.length() < length;
        }
    private:
        const int length;
};
```

   count_if(myVector.begin(), myVector.end(), ShorterThan(length));//直接调用即可

60. 命名空间

1. 命名空间的作用：大型的工程文件，不同头文件使用了相同的变量命名。可能造成变量命名冲突的问题，引入命名空间--一个由用户自己定义的作用域，在不同作用域中可以使用相同名字的变量，互不干扰，使用域标识符进行区分
2. 命名空间中的变量只会在自己的作用域中有效

3. 命名空间嵌套命名空间

```
namespace ns1{

   int a;

   namespace ns2{
        int b;}
}

   使用b：cout<<ns1::ns2::b<<endl
```

4. 命名空间别名

   namespace TV = ns1

5. 使用using命名空间成员名

   using ns1::a代表using作用域中的a默认为ns1::a，不能进行两个名字的命名，否则会造成重定义

6. 无名的命名空间：其他文件无法引用，只能在本文件的作用域中有效。和static相同作用

   而且本文件使用无名命名空间中的成员不需要添加命名空间限定符

7. 标准命名空间：库的标识符定义在std的命名空间之中