C++考点

1.面向对象编程的三大特性

**封装（安全性）**

封装就是将某些东西包装和隐藏起来，让外接无法直接使用，只能通过某些特定的方式才能访问。面向对象将万物都视为对象，任何对象都具有特性和行为，我们将特性称为成员变量，将其行为称为成员函数，被封装的特性只能通过特定的行为去访问。

**继承（复用）**

通过继承机制，可以利用已有的数据类型来定义新的数据类型。所定义的新的数据类型不仅仅拥有新定义的成员，而且还同时拥有旧的成员。我们称已存在的，用来派生新类的类为基类，又称为父类。由基类派生出来的新类称为派生类，又称为子类。

**多态（方便，语义一致性）**

多态将接口和实现分离，以共同的方法，但因个体的差异采取不同的策略。多态带来两个明显好处，一是不用大量的函数名了，二是会根据调用时的上下文来确定实现。

2.C++如何实现多态？

C++中共有3中实现多态的方式，函数重载，模板函数和虚函数。函数重载和模板函数是静态多态，编译时确定，动态多态通过继承和虚函数实现。

**函数重载**：相同函数名，参数类型或者参数数量不同，或者const关键字（常函数）。实现机制：交给编译器处理，通过编译时创建不同的名字来进行区分。

**模板函数：**函数体的内容有了，但是函数的参数类型待定，使用模板十分合适。实现机制：二次编译机制，实现编译器会在模板函数声明处进行一次编译，之后在函数模板调用的位置进行参数替换后进行第二次编译。

**虚函数：**父类中使用virtual关键字声明为虚函数，子类中对该虚函数进行覆写。使用父类指针指向不同的子类对象或者父类对象，通过指针调用虚函数可以调用相应的子类或者父类函数，而不是全部调用子类的函数。

实现机制：类中包含虚函数声明，则该类的所有对象共享一张虚表，该虚表每一行记录一个虚函数的入口地址。每个实例中包含自己的虚表指针，指向虚表。子类覆写父类虚函数时，子类虚表中对应中的虚函数入口地址被替换，但函数顺序不变。因此，当指向基类的指针ptr调用虚函数时，首先通过ptr找到虚表指针地址vptr，之后通过虚表指针访问虚表中的函数入口地址。由于前面虚函数入口地址替换原因，基类指针能找到正确的虚函数地址。

特点：父类=子类赋值时，并不会拷贝子类的虚表指针（默认赋值操作不会覆盖虚表指针）。对象赋值操作访问不了虚函数表（实现多态，必须要使用指针或者引用）。虚表指针一般占据类内存布局的第一个位置，来实现多层继承和多重继承时高效地访问虚表。另外，虚函数表的前面设置了一个指向type\_info的指针，用以支持RTTI（Run Time Type Identification，运行时类型识别）。RTTI是为多态而生成的信息，包括对象继承关系，对象本身的描述等，只有具有虚函数的对象在会生成。

3.为什么C++构造函数不能是虚函数，而析构函数都是虚函数？为什么绝不要在构造函数和析构函数中调用虚函数？

第一个问题：虚拟函数调用只需要“部分的”信息，即只需要知道函数接口，而不需要对象的具体类型（具体类型交给运行时）。但是构建一个对象，却必须知道具体的类型信息。如果你调用一个虚拟构造函数，你需要告知运行时对象类型，这个时候对象根本没被构造起来，而去问运行时类型，陷入了鸡生蛋蛋生鸡的悖论。

第二个问题：除非明确确定自己的类不会被继承，析构函数才可以使用普通函数，否则析构函数一定要使用虚函数。这是为了确保基类指针指向派生类时，派生类独有的资源能够被正确释放。

第三个问题：不是不能这样做，而是这样做的结果和预期效果不一样。在继承中：

a) 子类构造函数会先调用父类构造函数，而如果构造函数中有虚函数，此时子类还没有构造，所以此时的对象还是父类的，不会触发多态，调用的还是父类的虚函数。

b) 析构函数也是一样，子类先进行析构，这时如果有virtual函数的话，子类的内容已经被析构了，C++会视其父类，执行父类的虚函数。

4.指针和引用区别

指针是变量，指向内存存储单元；指针是别名，不是变量

指针在定义时可不初始化，引用定义时必须初始化

指针赋值之后可以在改变，引用不可以

指针可以指向NULL，引用不能为NULL

sizeof结果不同，引用是变量的大小，指针在32位机器上是4字节，64为机器上是8字节

指针可以有多级，引用只能是一级

5.虚函数，纯虚函数，抽象类

**纯虚函数**是在声明中使用=0来指定的，如virtual double getVolume() = 0;

类中包含至少一个纯虚函数，则该类为**抽象类**，抽象类不能实例化，只能用作接口。

6.成员函数=delete, =default, =0

=delete禁止使用编译器默认生成函数 =default 需要编译器生成一个默认的函数 =0一般和virtual关键字结合，表示纯虚函数，只声明不实现，留给子类去实现。

7.函数签名组成

函数签名（Signature）包含如下信息：

**函数的裸名**；

**class或者namespace**的作用域信息。如果该函数有内部链接属性的话，还应包含该函数所属的编译单元的相关信息；

**cv关键字**。如果函数是一个class的成员函数，且被const，volatile，或者const volatile修饰的话，还应包含这些信息；

**左值或右值**信息。如果函数是一个class的成员函数，且被&或者&&修饰的话，也应包含这些信息；

**模板参数类型**（如果函数是经由函数模板实例化得到的话，该类型指的是模板参数被实例化之前的类型，比如T或者T1）；

**返回类型**。如果函数是经由函数模板实例化得到，还应包含返回类型（非模板函数不包含此项内容）。

**实际参数类型**。如果函数是经由函数模板实例化得到，还应包含模板参数（比如T）以及用来实例化改模板的类型。

8. cv关键字const volatile

**volatile**关键字修饰的变量可能会意想不到的变化，提示编译器不去优化。例如两次读取变量i中间没有对i进行操作，这时第二次将会直接使用第一次存在寄存器中的i的值。这种编译器优化在多线程编程（如共享变量）和嵌入式程序中会发生错误，因此需要使用volatile告诉编译器每次读取该变量需要重新从内存读取寄存器。

**const**关键字作用比较多。

**修饰普通变量**：

const TYPE value or TYPE const value 含义相同

**修饰指针：**

const TYPE\* ptr, TYPE const \*ptr 指向对象不可变 TYPE\* const ptr 指针不可变

const TYPE\* const ptr指针不可变且指向对象不可变

**修饰引用：**

防止引用的对象被修改

**修饰函数参数**

防止函数内部修改传入参数，一般是结合引用使用

**修饰函数返回值**

返回值是底层const这样可以避免底层内容被外部接收之后修改，比如返回一个const char\*，必须使用const char\*来接收返回值，而不能用char\*;

返回值是顶层const则该const使用没有意义，比如返回char const\*，只是返回指针不能被修改没有没有意义，还是可以用char\*接收。

**修饰成员函数**

可以用来实现函数重载，const成员函数不能修改成员变量的值。（但是可以修改内容，比如包含成员char\*p，不能修改p指向的内存单元，但是能修改该内存单元内的存储数据）

**const与define区别**

处理阶段：#define由预处理器替换，const由编译器确定值

类型检查：#define无类型，不进行类型安全检查；const有类型，编译阶段进行类型检查

内存空间：#define直接进行替换，不会分配内存；const一般不分配内存除非必须分配不可，比如定义了const int a=12, int b=a;那么a可能只是被当做常数直接替换为12，但是如果添加了一条cout << &a，这个时候需要给a分配内存，才能打印地址。即使分配了内存后续的int b=12还是直接使用常数12替换a而不是去内存读取。基础类型（比如数值型）的const不能被修改，字符串还是可以被修改的，而不是是是用这样直接替换的方式。

一个例子

#include <iostream>

int main(){

int num\_e = 100;

const int num\_f = 100;

const int \*p\_e = &num\_e;

int \*p\_f = const\_cast<int\*>(p\_e);

\*p\_f = 1;

std::cout << p\_e << " " << p\_f << std::endl;

std::cout << num\_e << std::endl;

const int \*p\_m = &num\_f;

int \*p\_n = const\_cast<int\*>(p\_m);

\*p\_n = 2;

std::cout << &num\_f << " " << p\_m << " " << p\_n << std::endl;

std::cout << num\_f << " " << \*p\_n << std::endl;

return 0;

}

输出

0x7ffed18e4ed0 0x7ffed18e4ed0

1

0x7ffed18e4ed4 0x7ffed18e4ed4 0x7ffed18e4ed4

100 2

num\_f在最后打印的时候并没有从内存中去读取，而是编译时替换为100，因此不要尝试const\_cast一个本身不可变的对象，const\_cast一般用于底层const指针，但该指针指向的内存区域本身能被修改。

作用域不同：#define在任何局部内被定义，在之后的任何区域内都能被使用，const是变量属性，只在作用域内可见。

是否能被取消：#define定义后可使用#undef来取消定义，取消之后不可用。const定义后无法取消，在作用于内一直可用。

9.底层const和顶层const

底层const🡪指向常量的指针

底层const🡪指针常量

区分的作用：

**赋值（底层const不能赋值给非底层const）**

const int a=1;

const int\* b=&a; //正确 &a是底层const

int\* c=b;错误

**去除底层const时需要使用const\_cast**（至于指向对象如果不能被修改，那么转换了也没用，可能会出现出乎意料的事情，见上面的代码，num\_f打印仍然是100）

第二点需要结合下面的四种类型转换

10.C++11的4种类型转换

C语言强制类型转换的三个缺点：

没有从形式上体现转换功能和风险的不同；

将多态基类指针抓换成派生类指针时不检查安全性，无法判断转换后的指针是否确实指向一个派生类对象；

难以在程序中寻找到底什么地方使用了强制类型转换。

**static\_cast**比较自然和低风险的转换，例如整数浮点，字符之间转化。或者某个类实现了强制类型转换运算符T重载，则也能使用static\_cast<T>进行转换。static\_cast不能用于不同类型的指针间的转换，也不能用于整数和指针间的转换，也不能用于不同类型的引用之间的转换，这些转换风险比较高。

**reinterpret\_cast**十分灵活，执行过程是逐个比特的复制。这种转换提供了很强的灵活性，但是 安全性也全部交付给程序员自己保证。可用于不同类型的指针之间，不同类型引用之间，指针和整数之间的转换。同时reinterpret\_cast也能将多态基类的指针强制转化为派生类的指针，但这种转换不检查安全性，因此针对基类指针/引用向下转换为派生类指针/引用的操作专门使用dynamic\_cast操作来进行。

**dynamic\_cast**专门用于将多态基类指针/引用转换为派生类指针/引用。对于不安全的指针转换，返回nullptr，对于不安全的引用转换，抛出异常。

**const\_cast**专门用于去掉const属性的转换。将const引用/指针转换为非const类型的引用/指针。不过不允许修改基础类型的const（比如const int i）const string s还是能修改。

11.模板的偏特化和全特化

小结论：

模板函数拥有不同的函数签名原则上能在程序中共存，但共存不代表一定能被正常使用，可能会导致重载歧义。

模板函数只能全特化，不能偏特化（会存在一些问题，还没得到C++标准认识）

模板类可以偏特化和全特化

**函数模板全特化**

**template<typename** T**>** *// 模板1*

**int** f(T)

{

**return** 1;

}

**template<>** *// 模板3， 对模板1进行了全特化，不需要写成 template<> int f<int>(int)*

**int** f(**int**)

{

**return** 1;

}

**类成员函数可以全特化、偏特化**

这个时候需要在函数前面添加<int>，不能省略

12.C++的公有继承（public），私有继承（private），保护继承（protected）

public继承原有的public, protected属性直接继承下来，权限不变，private不可见

private继承原有的public, protected属性继承下来，权限全部private，private还是不可见

protected继承原有的public, protected属性继承下来，权限全部protected, private还是不可见

public成员 自己、友元、子类、外人均能访问

protected成员 自己、友元、子类均能访问

private成员只能 自己、友元访问

13.虚继承和虚基类

虚继承是多继承中的一个概念。

下面的继承关系导致类D中包含两个A，一个是B继承来的，一个是C继承来的，这样导致了两个问题，首先是存储空间上产生了浪费，保存了两个A，另一个是访问A中成员时容易出现冲突，因为包含两个A，不能直接用变量名访问，而要添加是通过B还是C来的（B::m\_a形式）。为了解决这两个问题引入虚继承。

A A

↓ ↓

B C

↘ D ↙

1 class A{ public: int dataA; };

2 class B : public A{ public int dataB; };

3 class C : public A{ public int dataC; };

4 class D : public B, public C{ public int dataD; };

虚继承如下。B、C继承A时使用virtual关键字，表示**虚继承**A，承诺愿意共享它的基类A。类A被称为**虚基类**。这样派生类D中A的成员只包含一份，可以直接通过成员名访问。同时，也减少了内存消耗。

A

↙ ↘

B C

↘ D ↙

1 class A{ public: int dataA; };

2 class B : virtual public A{ public int dataB; };

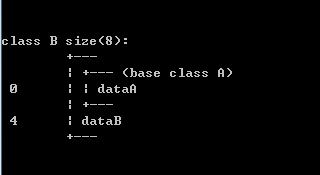
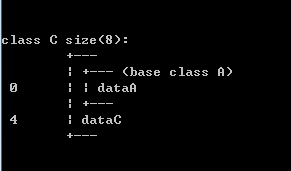
3 class C : virtual public A{ public int dataC; };

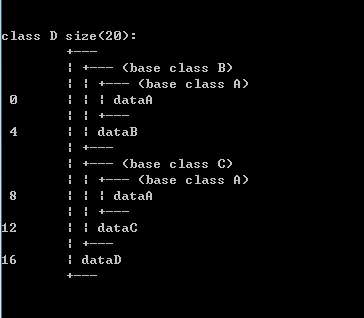
4 class D : public B, public C{ public int dataD; };

14.虚基类表和虚基类表指针

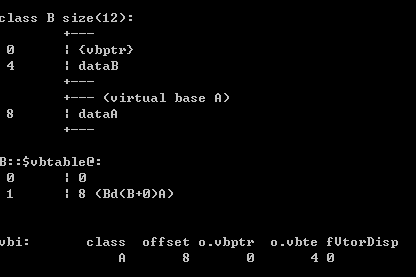
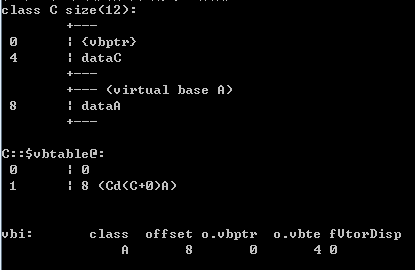
虚继承的实现方式是虚基类表指针和虚基类表。这两个概念和虚指针、虚表很像。和虚表一样，虚基类表也是由类成员，而不是对象成员，被所有对象共享，而虚基类表指针则是每个对象都有。虚表中记录的是每个虚函数的入口地址，而虚基类表记录的是每个虚基类成员到直接继承类地址的偏移量。

上面代码中非虚继承中B,C,D的内存布局如下（vs编译器）

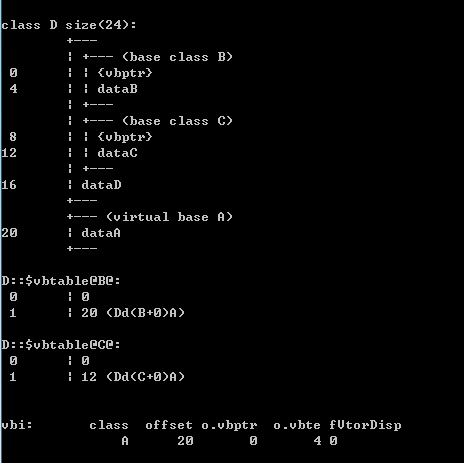
 



虚继承下B,C,D的内存布局（同样还是vs编译器）

B和C中由于继承成员dataA放在内存布局的最后，因此虚基类表中的偏移量均为8字节。

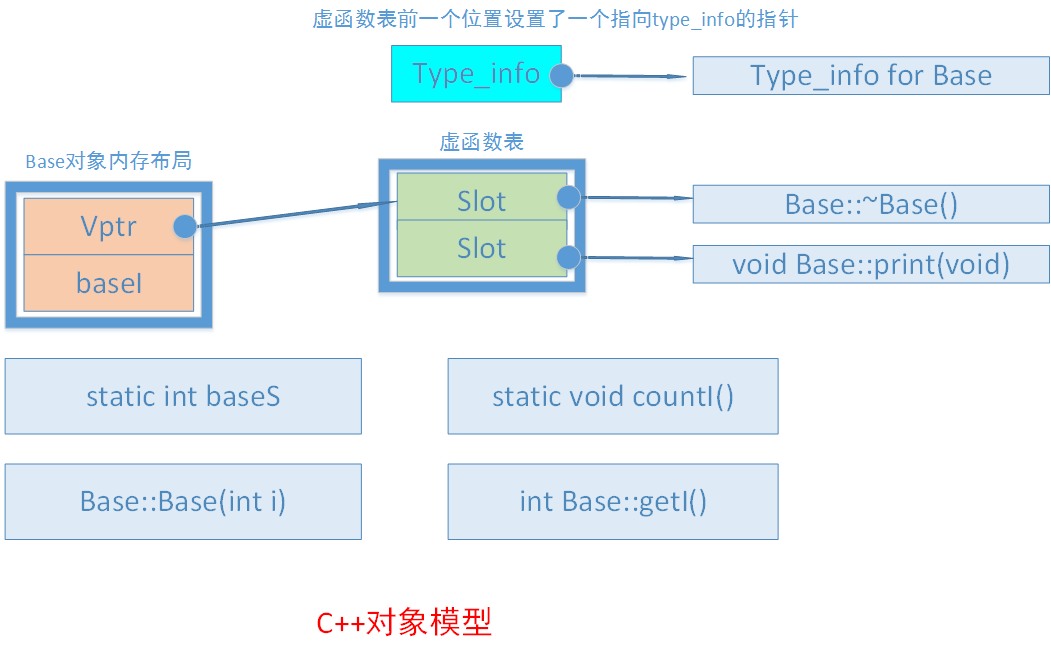


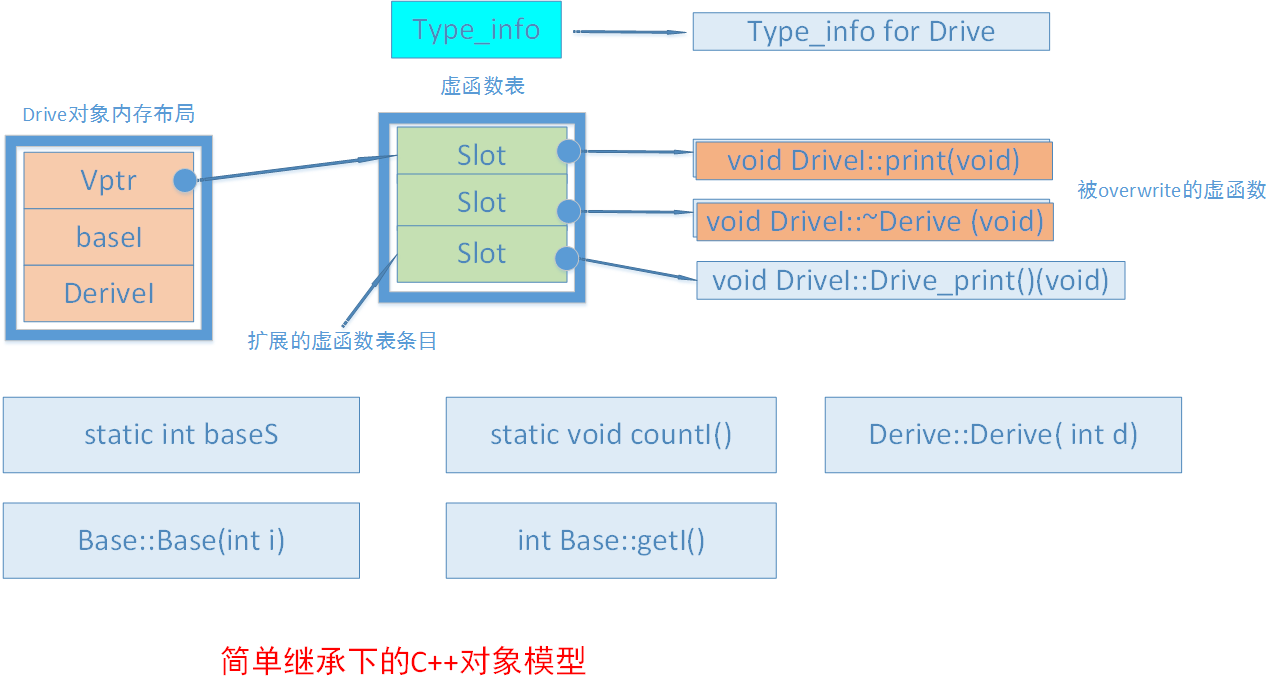
D中两个虚基类表中的偏移量分别是20和12，这样保证了在类D中无论是dataA还是B::dataA还是C::dataA，都是访问同一块内存单元。

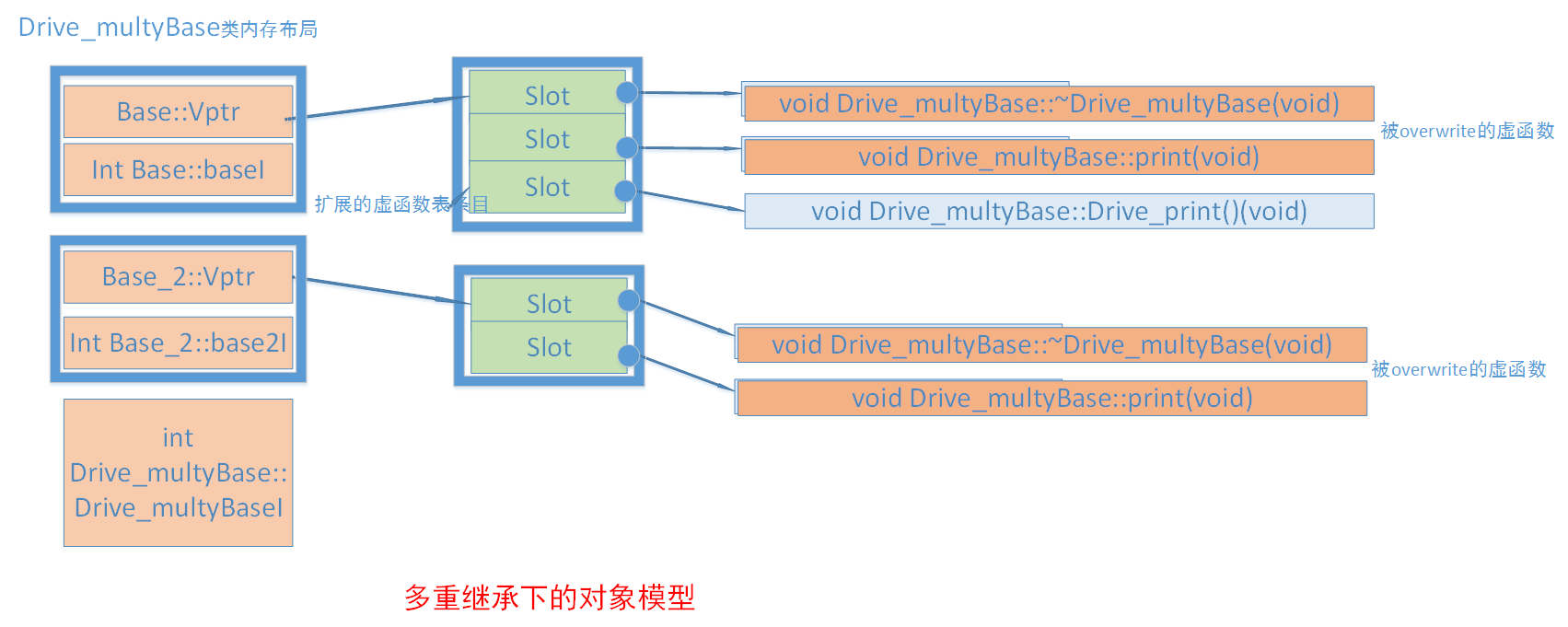
15.对象内存模型和继承中的内存布局 包含虚函数，静态成员，静态成员函数，虚基类等等

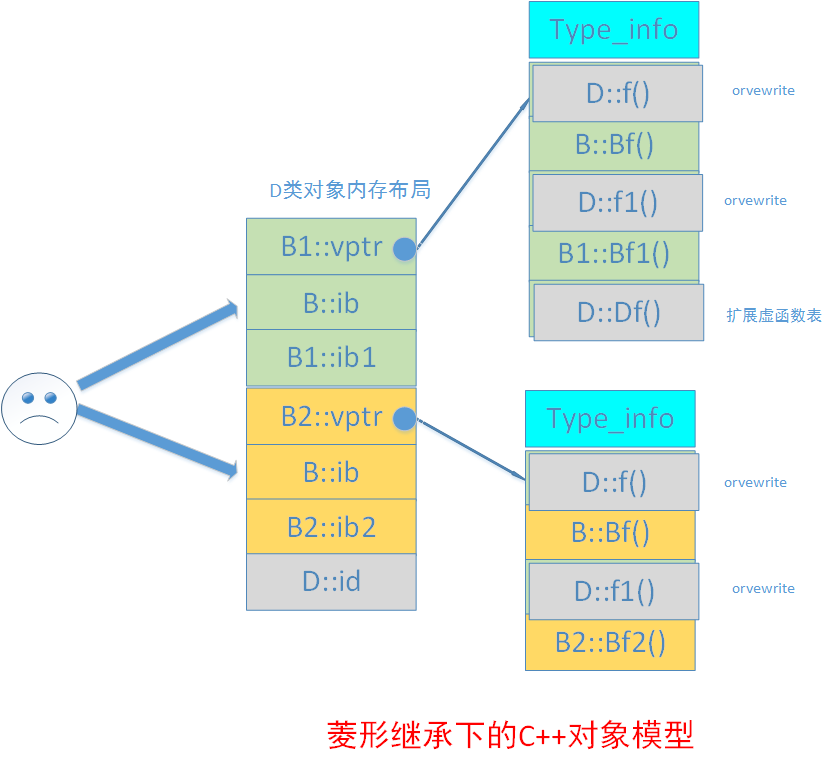
参考：<https://www.cnblogs.com/QG-whz/p/4909359.html>

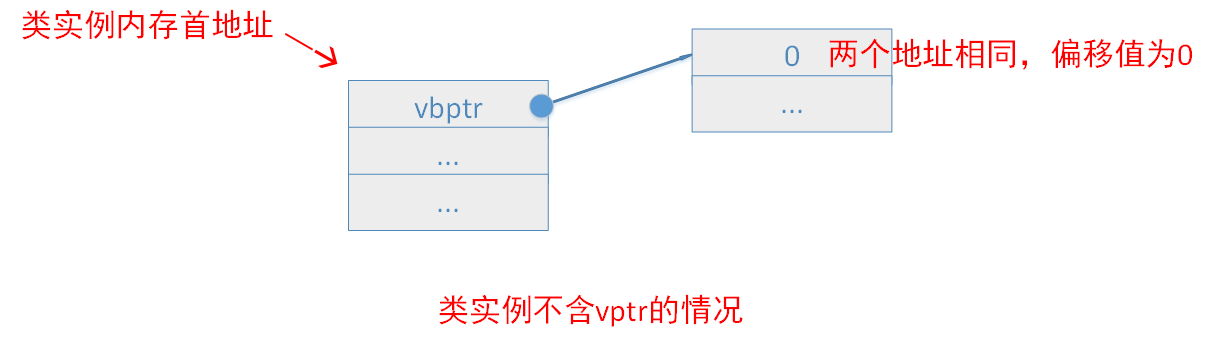
小结论：C++对象内存模型中静态成员和函数（不管static还是non-static）都不占据对象内存。静态成员放在数据区的静态区，成员函数放在代码区，其他普通成员变量，虚指针、虚表指针都放在栈区。

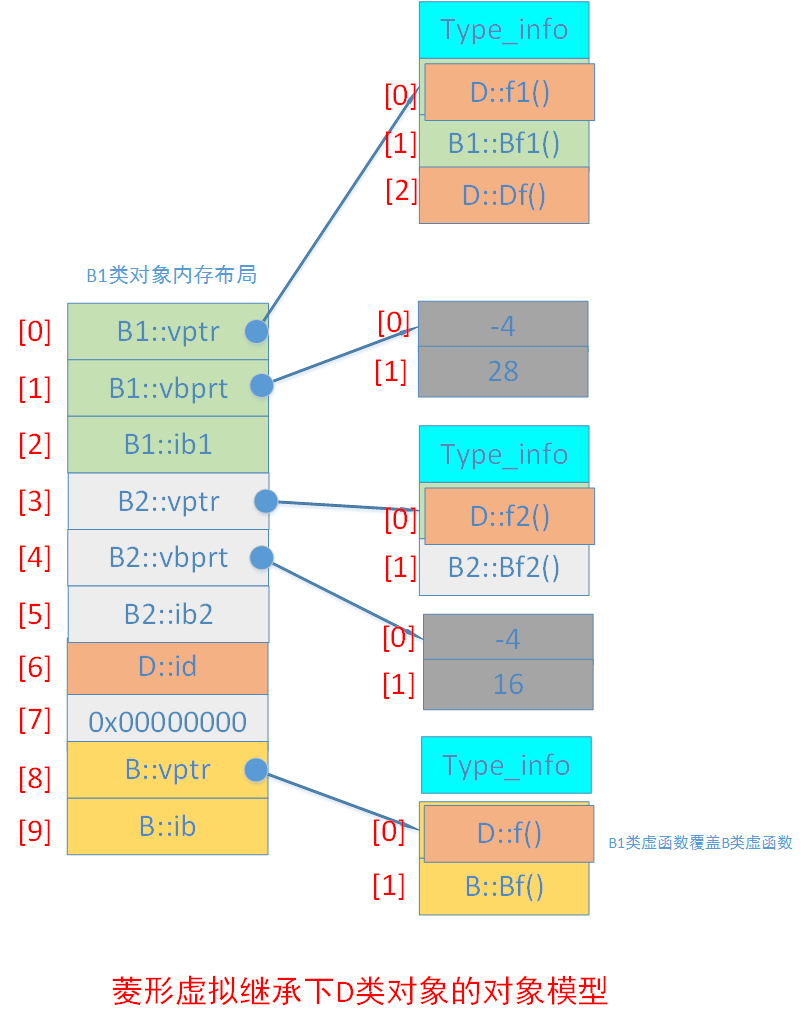
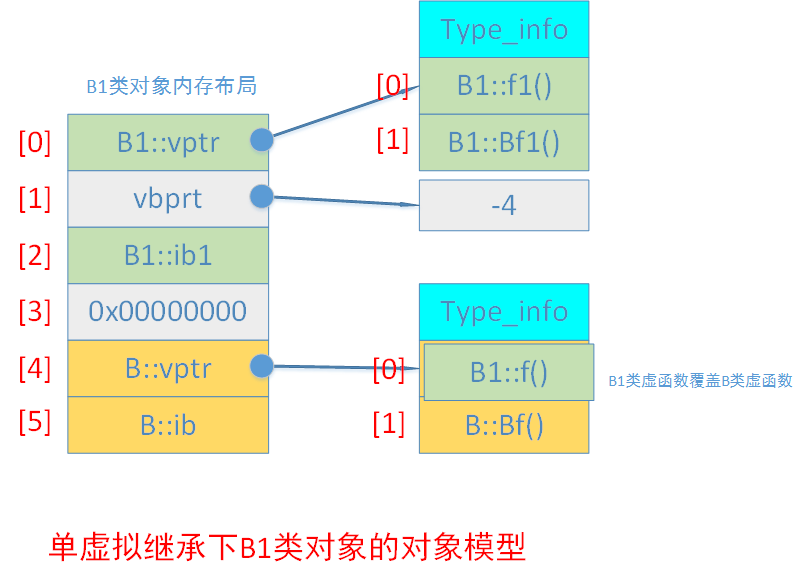






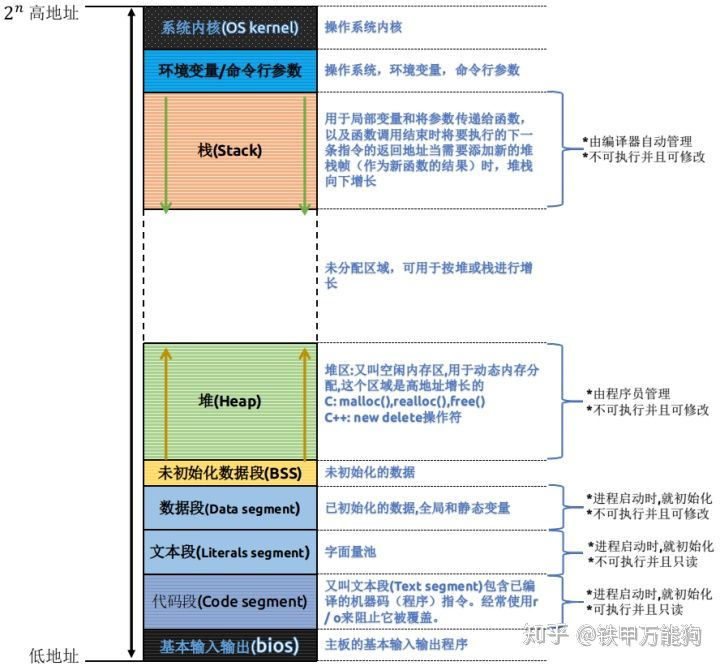






16.C++内存布局

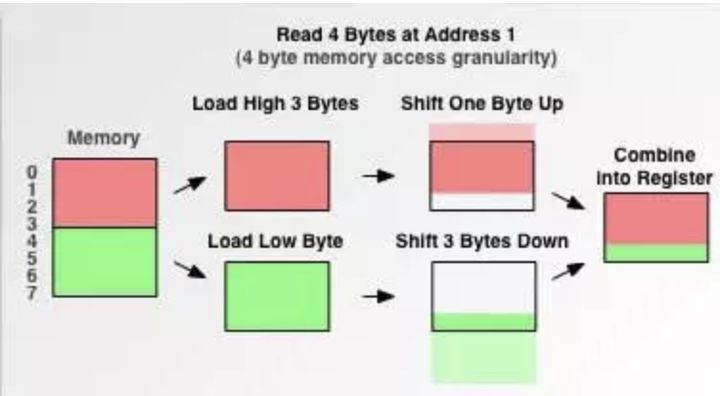
32位系统中，最高的1GB是内核空间，靠近内核内存的高地址内存还包含环境变量，命令行参数等。低地址的3GB是用户空间，包含最低地址的BIOS占据的内存。中间是应用程序的可用空间。自低地址开始，分为**代码区，全局区，堆区，栈区**。全局区自低地址向高地址分为**常量区**（字符串常量，const常量），**data区**（已初始化的全局变量，已初始化的静态变量和常量），**BBS区**（BBS区全部被初始化为0，包含未初始化的全局变量，未初始化的静态变量，初始化为0的全局变量和静态变量），堆区（程序员分配和释放，向高地址增长），栈区（普通变量，未初始化，向低地址增长）。C语言中区分初始化和未初始化的全局数据和静态数据，因此分了data区和BBS区，但是C++中这两个区域不在区分，合并为全局数据区。



17.C++内存对齐

现代计算机中内存空间都是按照 byte 划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但是实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k（通常它为4或8）的倍数，这就是所谓的内存对齐。一般处理器会以双字节,四字节,8字节,16字节甚至32字节为单位来存取内存，我们将上述这些存取单位称为**内存存取粒度**。

假如没有内存对齐机制，数据可以任意存放，现在一个int变量存放在从地址1开始的联系四个字节地址中，该处理器去取数据时，要先从0地址开始读取第一个4字节块,剔除不想要的字节（0地址）,然后从地址4开始读取下一个4字节块,同样剔除不要的数据（5，6，7地址）,最后留下的两块数据合并放入寄存器.这需要做很多工作。



每个特定平台上的编译器都有自己的默认“对齐系数”（也叫**对齐模数**）。**gcc**中默认**#pragma pack(4)**，可以通过预编译命令#pragma pack(n)，n = 1,2,4,8,16来改变这一系数。

**有效对其值**：是给定值#pragma pack(n)和结构体中最长数据类型长度中较小的那个。有效对齐值也叫对齐单位。

对齐规则：

(1)结构体第一个成员的偏移量（offset）为0，以后**每个成员相对于结构体首地址的 offset 都是该成员大小与有效对齐值中较小那个的整数倍**，如有需要编译器会在成员之间加上填充字节。(2)**结构体的总大小为有效对齐值的整数倍**，如有需要编译器会在最末一个成员之后加上填充字节。

**struct**

{

**int** i;

**char** c1;

**char** c2;

}x1;

**struct**{

**char** c1;

**int** i;

**char** c2;

}x2;

**struct**{

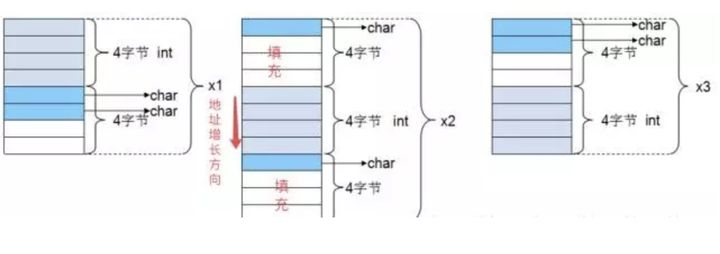
**char** c1;

**char** c2;

**int** i;

}x3;

对应的内存布局如下。



18.堆和栈的区别，为什么栈要快？

**管理方式不同**：堆程序员使用new/malloc动态申请，使用delete/free释放，堆空间使用自由链表管理，栈，编译器在需要的时候分配，不需要的时候自动清除内存，有底层指令和寄存器支持。

**释放顺序不同**：栈上的内存符合先分配后释放的顺序，堆上的内存则是自由存储，分配和释放的次序没有限制。

**访问速度差异**：栈的访问速度更快，主要的原因是栈是当前带代码最常访问的内存，高速缓存命中的概率几乎是100%，而堆上面的数据显然访问没有栈频繁，因此堆上数据缓存命中的概率很低，速度自然就慢了下来。

**分配方式不同**：栈内存的分配具备底层支持，有专门的寄存器存储栈的地址，入栈和出栈有专门的指令，且栈分配连续，只要栈剩余空间足够就能分配，因此分配效率很高；堆内存由程序员管理，一般采用自由链表的方式管理堆内存，分配内存时按照一定的算法在自由链表中寻找最合适的内存块大小，因此分配效率较低。

**申请空间大小**：栈顶的地址和栈的最大容量是系统预先定制好的，通常为1M可使用ulimit -s查看和重新设置。堆受限于计算机系统中有效的虚拟内存，在32位系统中，堆内存可以达到2.9G大小，几乎占满全部用户空间。

19.new/delete和malloc/free区别

**1)**malloc/free为C的标准库，new/delete则为C++关键字，new/delete会由便器转为operator new和operator delete的调用，这两者的底层调用malloc和free；

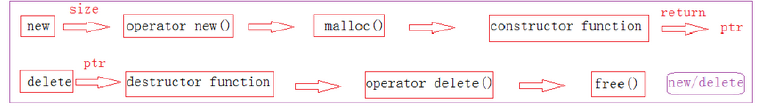
**2)**malloc开辟空间大小需要手动计算，new由编译器自己计算；

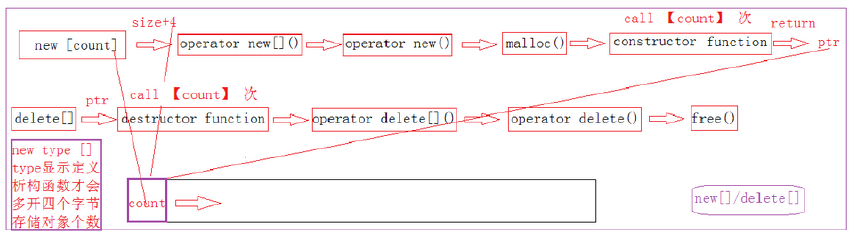
**3)**malloc返回指针类型是void\*，必须要使用强制类型转换转为对应类型的指针，new直接返回对应类型的指针;

**4)**malloc失败返回NULL，new失败抛出bad\_alloc异常；

**5)**malloc只负责开辟空间并且释放，new开辟空间，且调用构造函数，delete调用析构函数和释放空间；

**6)**malloc内存全部用free释放，new单个对象用delete，对象数组需要用delete [], 不加[]只调用一次对象的析构函数，极有可能造成内存泄露;





在开辟大小会多开辟四个字节，用于存放对象的个数，在返回地址时则会向后偏移4个字节，而在delete时则会查看内存上对象个数，从而根据个数count确定调用几次析构函数，从而完全清理所有对象占用内存。

**7)**申请内存位置，malloc一定是在堆上，new的内存位置取决于operator new的实现。

20.C++哪些情况出现内存泄漏？

1)malloc的没有free, new的没有delete, new []的没有delete[];

2)二级指针释放遗漏，二级指针需要两次释放；

3)基类析构函数未声明为虚函数，基类指针指向派生类对象时，调用delete不能释放派生类中的动态内存；

4)使用的第三方库存在内存泄露。

21.C++如何判断内存泄露方法？内存内存分析和动态内存分析？如何避免内存泄露？

1)自己实现的话，可以覆写operator new和operator delete函数，在这两个函数内部添加日志打印功能，然后通过分析日志，找到存在分配内存日志但是没有释放内存日志的地址，就发生了内存泄露。具体实现可以参考

<https://www.zhihu.com/question/29859828/answer/1798470821>

2)可以使用gcc（4.8以上）自带工具AddressSanitizer(ASan)，该工具可以用来检测内存泄漏、堆栈和全局内存越界访问、free后继续使用、局部内存被外层使用、Initialization order bugs

更加细分有静态内存分析和动态内存分析

静态内存分析直接对代码进行分析，不运行程序，不一定准确，但是基本正确，需要根据上下文代码来分析是否真的有问题。相关工具有

动态内存分析运行起来程序进行分析，分析非常准确，需要一处一处来分析，不能对全局代码进行分析。相关工具有

一般都是静态分析和动态分析结合。

避免内存泄露的方法

1)尽可能用智能指针管理动态内存；

2)使用string对象而不是char\*;

3)除非要使用旧的lib接口，否则不要使用原始指针；

4)避免new/delete使用，如果一定要使用，应该隐藏在一个RAII（资源获取即初始化）对象中，当变量离开当前范围时，资源可以自动被释放。

5)在new/delete之间写代码；

6)对涉及动态内存的程序进行内存分析。

C++不存在垃圾回收，自己实现垃圾回收的思路可以仿照其他高级语言，比如python引用计数，标记-清除机制和分代机制。

22.C++智能指针，什么时候修改引用计数？怎么解决循环引用问题的？

一共包含三个智能指针，**unique\_ptr**（独占资源所有权）, **shared\_ptr**（共享资源所有权）, **weak\_ptr**（共享资源的观察者，和shared\_ptr一起使用，不影响引用计数）

**1)std::unique\_ptr**

当我们独占资源的所有权的时候，可以使用 std::unique\_ptr 对资源进行管理——离开 unique\_ptr 对象的作用域时，会自动释放资源。这是很基本的 RAII 思想。std::unique\_ptr 是 move-only 的。

std::unique\_ptr<int> uptr = std::make\_unique<int>(200);

std::unique\_ptr<int> uptr1 = uptr; // 错误

std::unique\_ptr<int> uptr2 = std::move(uptr); // 正确，uptr=nullptr

std::unique\_ptr<int[]> uptr = std::make\_unique<int[]>(10); //指向数组

**自定义deleter 是仿函数（实现了（）运算符重载的类）**

struct FileCloser {

void operator()(FILE\* fp) const {

if (fp != nullptr) {

fclose(fp);

}

}

};

std::unique\_ptr<FILE, FileCloser> uptr(fopen("test\_file.txt", "w"));

// 或者使用lambda表达式

std::unique\_ptr<FILE, std::function<void(FILE\*)>> uptr(

fopen("test\_file.txt", "w"), [](FILE\* fp) {

fclose(fp);

});

**2)std::shared\_ptr**

std::shared\_ptr 其实就是对资源做引用计数——当引用计数为 0 的时候，自动释放资源。

std::shared\_ptr<int> sptr = std::make\_shared<int>(200);

同样也支持自定义deleter

引用计数原理

可以采用指向相同动态内存的指针实现

什么时候引用计数增加，什么时候引用计数减少？

引用计数增加：使用shared\_ptr初始化另一个shared\_ptr，传递给函数，作为函数返回值

引用计数减少：给shared\_ptr赋予一个新值，shared\_ptr被销毁，shared\_ptr离开作用域

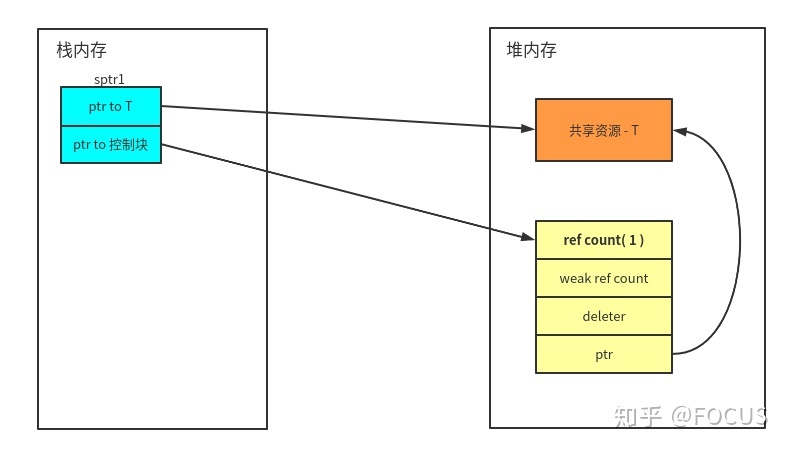
auto r = make\_shared<int>(42);

r = q; // 递增q指向对象的引用计数，递减r原来指向对象的引用计数

引用计数具体实现方式由标准库自己决定

**shared\_ptr和unique\_ptr对比**

无自定义 deleter 的 unique\_ptr 只需要将裸指针用 RAII 的手法封装好就行，无需保存其它信息，所以它的开销和裸指针是一样的。如果有自定义 deleter，还需要保存 deleter 的信息。shared\_ptr 需要维护的信息有两部分：指向共享资源的指针；引用计数等共享资源的控制信息——实现上是维护一个指向控制信息的指针。shared\_ptr 的 的 deleter 是保存在控制信息中，所以，是否有自定义 deleter 不影响 shared\_ptr 对象的大小。shared\_ptr如下



**shared\_ptr存在循环引用问题，因此引入weak\_ptr**

**3)std::weak\_ptr**

利用weak\_ptr就可以解决循环引用的问题，因为weak\_ptr是弱引用型指针，是用来监视shared\_ptr的，不会使引用计数增加。它不管理shared\_ptr内部的指针，也就是数据指针，它是用来监视shared\_ptr生命周期的，通过它的构造不增加引用计数，析构不减少引用计数这一点，从而解决了循环引用的问题。

struct A;

struct B;

struct A{

shared\_ptr<B>bptr; // ①weak\_ptr<B>bptr;

~A(){ cout << "A has been destory!" << endl;

}

};

struct B{

shared\_ptr<A> aptr; // ②weak\_ptr<A>bptr;

~B() { cout << "B has been destory!" << endl;

}

};

int main(){

shared\_ptr<A> pa(new A);

shared\_ptr<B> pb(new B);

pa->bptr=pb;

pb->aptr=pa;

return 0;

}

原程序对象A和B引用计数离开main函数均为1，因此无法调用析构函数。在1或者2处之一修改一处即可消除循环引用问题，比如修改①处，离开main时，开始A的引用计数为0，B的引用计数是1，因此先调用A的析构函数，接着调用shared\_ptr<B>对象的析构函数，使得B的引用计数减为0，这个时候调用B的析构函数。由此可见循环引用中只要有一处破除掉shared\_ptr的引用计数即可，不需要每个地方都进行替换。

23. auto关键字, decltype关键字

了解一点基本的用法算了，auto是进行自动类型推导，通过初始化值来完成；decltype根据表达式推导类型，无需初始化。auto常用于迭代器，lambda表达式。其他黑魔法不去关心了。

**auto**

auto i =100; // i 是 int

auto p = new A(); // p 是 A\*

auto k = 34343LL; // k 是 long long

map <string, int, greater <string> >mp;

for( auto i = mp.begin(); i != mp.end(); ++i)

cout << i -> first << ", " << i -> second;

编译器会自动判断出 i 的类型是 map<string, int, greater >::iterator。

**decltype**

int i;

double t;

struct A { double x; };

const A\* a = new A();

decltype(a) x1; //x1 是 A\*

decltype(i) x2; //x2 是 int

decltype(a -> x) x3; // x3 是 double

联合使用

struct A {

int i;

A(int ii) : i(ii) {}

};

A operator + (int n, const A & a)

{

return A(a.i + n);

}

template <class T1, class T2>

// C++14中可以直接去掉用auto了，不需要decltype配合

auto add(T1 x, T2 y) -> decltype(x + y) {

return x + y;

}

24. lambda表达式

主要是捕获列表

可进行**传值捕获**或者**引用捕获**

int value = 0;

auto a1 = [](int x) {/\*仅能访问全局外部变量\*/};

auto a2 = [value](int x) {/\*值传递局部变量value\*/};

auto a3 = [this](int x) {/\*值传递this指针\*/};

auto a4 = [&value](int x) {/\*引用传递局部变量value\*/};

auto a5 = [=](int x) {/\*值传递所有可访问的外部变量\*/};

auto a6 = [&](int x) {/\*引用传递所有可访问的外部变量\*/};

auto a7 = [=, &value](int x) {/\*引用传递局部变量value,值传递所有其他可访问的外部变量\*/};

auto a8 = [&, value](int x) {/\*值传递局部变量value，引用传递所有其他可访问的外部变量\*/};

25. nullptr关键字

C++11新标准引入nullptr原因

#define NULL (void\*)0 // C语言中NULL定义

#ifndef NULL

#ifdef \_\_cplusplus

#define NULL 0 // C++中NULL定义

#else

#define NULL ((void \*)0)

#endif

#endif

下面程序中，把NULL当做指针编译出错，void\*不能自动转为char\*，把NULL当做0不符合我们的预期，因此C++11引入nullptr明确区分两种情况。空指针一律采用nullptr表示。

void foo(int a)

{

cout<<"This is int"<<endl;

}

void foo(char \* ptra)

{

cout<<"This is pointer"<<endl;

}

int main(void)

{

foo(NULL); //

return 0;

}

26.新的for循环用法

基于范围的for循环

27. explict关键字的作用

explicit关键字的作用就是**防止类构造函数的隐式自动转换**.

class CxString

{

public:

char \*\_pstr;

int \_size;

CxString(int size) // explicit CxString(int size)

{

\_size = size; // string的预设大小

\_pstr = malloc(size + 1); // 分配string的内存

memset(\_pstr, 0, size + 1);

}

};

// 下面是调用:

CxString string1(24); // 这样是OK的, 为CxString预分配24字节的大小的内存

CxString string2 = 10; // 这样是OK的, 为CxString预分配10字节的大小的内存

CxString string6 = 'c'; // 这样也是OK的, 其实调用的是CxString(int size), 且size等于'c'的ascii码

第二条第三条相当于隐式构建CxString temp(10), CxString temp2(‘c’)再调用默认的赋值运算符。添加explict后能避免第二条和第三条的隐式转换

28. extern C作用

被extern "C"修饰的变量和函数是按照C语言方式编译和链接的。

这个问题主要出现C++项目在依赖第三方的C库（提供了动态库和头文件）。这种情况下对于第三方库中的函数声明需要添加extern C包含起来。这是因为C++支持函数重载，C不支持导致二者的函数签名有所区别。如果不添加extern C，那么链接时将按照C++的函数签名方式去动态库中寻找，就找不到出现undefined reference链接错误。

举例：

void foo( int x, int y ); 该函数被C编译器编译后在符号库中的名字为\_foo，而C++编译器则会产生像\_foo\_int\_int之类的名字

extern修饰变量，函数

修饰符extern用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用。能够被其他模块以extern修饰符引用到的变量通常是全局变量，函数也是全局函数。

29. 什么是野指针？

指向非法的内存地址指针叫作野指针（Wild Pointer），也叫悬挂指针（Dangling Pointer），意为无法正常使用的指针。出现野指针的情形：使用未初始化的指针、指针所指的对象已经消亡、指针释放后之后未置空。

野指针有时比较隐蔽，编译器不能发现，为了防止野指针带来的危害，开发人员应该注意以下几点。

（1）C++引入了引用机制，如果使用引用可以达到编程目的，就可以不必使用指针。因为引用在定义的时候，必须初始化，所以可以避免野指针的出现。

（2）如果一定要使用指针，那么需要在定义指针变量的同时对它进行初始化操作。定义时将其置位NULL或者指向一个有名变量。

（3）对指针进行free或者delete操作后，将其设置为NULL。对于使用 free 的情况，常常定义一个宏或者函数 xfree 来代替 free 置空指针：

#define xfree(x) free(x); x = NULL;

30.C++引用是如何实现的？

引用底层实现就是变量地址或者说是指针，可以通过汇编代码发现，引用其实也是占据内存的，保存着变量的地址。int a=1; int& b=a; 之所以&b不能得到引用地址是因为编译器把&b变成了&（\*b）。

31.指针和数组的区别？

1)赋值上有区别，指针可以互相赋值，数组只能逐个元素的拷贝赋值；

2)数组代表一块连续对象的内存空间，指针只是指向一个对象；

3)数组名同时也是数组第一个元素的首地址，这是个常量，无法更改，指针指向则可以随便更改；

4)sizeof不同，在32位系统中，sizeof(指针)始终是4，sizeof(数组)=sizeof(元素类型)\*数组长度。

5)数组名赋值给指针是可以的，失去数组的内涵，也失去常量特性。

32.数组指针（行指针）和指针数组的区别

int (\*arr)[8]; 数组指针，本质是指针，指向连续8int的内存空间；

int \*arr[8]; 指针数组，本质是数组，每个元素类型是int\*。

数组名可以看做一个const的数组指针

33.指针函数和函数指针区别

int \*fun(int x,int y); 指针函数，本质上是函数，返回值类型是指针；

int (\*fun)(int x,int y); 函数指针，本质是指针，可以指向同类型的函数名

34.sizeof和strlen区别与联系

1)sizeof**接收参数类型**包含数组，指针，类型，对象，函数（）

数组—编译时确定分配的数组空间大小

指针—指针存储空间的大小，32位机器4字节，64位机器8字节

类型—该类型占据的内存空间大小

对象—包含非静态成员，虚函数指针，虚基类指针等

函数—&函数名返回地址大小占据内存，同指针，函数名()返回函数返回类型的大小，返回类型不能是void，不存在sizeof(函数名)

strlen接收的类型只能是字符串首地址，包含常量字符串和带’\0’结尾的字符串数组

**2)计算时间不同**，sizeof编译时确定，因此像sizeof（函数名（））不需要真的去调用函数，同时sizeof也无法获取动态数组的字节数。strlen只能运行时确定值的大小。

**3)类型不同**，sizeof是操作符，语言层面支持，strlen是函数，由字符串库支持。

35. this指针

通常在class定义时要用到类型变量自身时，因为这时候还不知道变量名（为了通用也不可能固定实际的变量名），就用this这样的指针来使用变量自身。

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。

this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。例如，调用date.SetMonth(9) <===> SetMonth(&date, 9)，this帮助完成了这一转换。

**使用场景**

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return \*this；另外一种情况是当参数与成员变量名相同时，如this->n = n （不能写成n = n）。

36.C++引入内联函数的原因

内联函数主要是为了用来取代C语言中的宏。使用C语言中的宏也能实现和内联函数一样的功能，但是使用宏的话，预处理器直接进行替换操作，不能利用编译器严格的类型检查，使用存在一定隐患。inline函数给编译器建议，在调用处内联地展开开始，避免函数调用的开销，同时也利用了编译器的类型检查功能，更加安全可靠。

37. main函数执行前还会执行什么代码？

程序在执行时会调用各种各样的运行时库函数，因此执行前main函数前必须要**初始化好运行时库**，**mainCRTStartup函数**会负责相应的初始化工作，他会完成一些**C全局变量以及C内存分配等初始化工作**，在C++里，还要执行**全局类对象的构造函数**。最后，mainCRTStartup才调用main函数。

在main函数返回之后，他会记录main函数的返回值，调用atexit注册的函数，然后结束进程。

**全局对象的构造函数**在main函数之前调用，析构函数在main函数之后调用。

**全局变量的初始化**操作；

局部栈对象在定义的时候调用构造函数，出了可见范围的时候调用析构函数。

堆对象在new的时候调用构造函数，delete的时候调用析构。

**全局静态对象**和全局对象一样。

**局部静态对象**在定义的时候调用构造，main函数之后调用析构

38.类什么时候会被析构？

对象生命周期结束，被销毁时；

delete指向对象的指针，或delete指向对象的基类指针，而其基类析构函数是虚函数；

对象i是对象o的成员，o的析构函数的最后，对象i的析构函数也被调用。

39.重载和覆盖的区别

重载：函数重载是指在同一作用域内（名字空间），可以有一组具有相同函数名，不同参数列表的函数；

覆盖（也叫重写）：指在派生类中重新对基类中的虚函数（注意是虚函数）重新实现。即函数名和参数都一样，只是函数的实现体不一样；

40.static用法

（1）在**修饰变量**的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。

（2）static **修饰全局变量**的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是 extern 外部声明也不可以。

（3）static **修饰一个函数**，则这个函数的只能在本文件中调用，不能被其他文件调用。static 修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为 0。

（4）**不想被释放**的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用 static 修饰。

（5）考虑到**数据安全性**（当程序想要使用全局变量的时候应该先考虑使用 static）

全局变量和全局静态变量的区别

1）**全局变量是不显式用 static 修饰的全局变量，全局变量默认是有外部链接性的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 extern 全局变量名的声明，就可以使用全局变量。**

2）**全局静态变量是显式用 static 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 extern 声明也不能使用**。

**静态局部变量**有以下特点：

（1）该变量在全局数据区分配内存；

（2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；

（3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；

（4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束。

41.单例模式

**1）懒汉模式：标准的 “双检锁” + “自动回收” 实现**

class Singleton

{

public:

static Singleton\* GetInstance()

{

if (m\_pInstance == NULL )

{

Lock(); // 加锁

if (m\_pInstance == NULL )

{

m\_pInstance = new Singleton ();

}

UnLock(); // 解锁

}

return m\_pInstance;

}

// 实现一个内嵌垃圾回收类

class CGarbo

{

public:

~CGarbo()

{

if(Singleton::m\_pInstance)

delete Singleton::m\_pInstance;

}

};

static CGarbo Garbo; // 定义一个静态成员变量，程序结束时，系统会自动调用它的析构函数从而释放单例对象

private:

Singleton(){};

Singleton(Singleton const&);

Singleton& operator=(Singleton const&);

static Singleton\* m\_pInstance;

};

Singleton\* Singleton::m\_pInstance = NULL;

Singleton::CGarbo Garbo;

**静态局部变量的懒汉模式 ，而不是new在堆上创建对象，避免自己回收资源。C++11中要求编译器保证静态变量初始化的线程安全性，可以不加锁。但C++ 0X以前，仍需要加。**

class Singleton

{

public:

static Singleton\* GetInstance()

{

Lock(); // not needed after C++0x

static Singleton instance;

UnLock(); // not needed after C++0x

return &instance;

}

private:

Singleton() {};

Singleton(const Singleton &);

Singleton & operator = (const Singleton &);

};

**2）饿汉模式**

**基础版本：程序一开始完成单例对象的初始化，后续不再需要考虑多线程安全性问题，就可以避免懒汉模式里频繁加锁解锁带来的开销。**

class Singleton

{

public:

static Singleton\* GetInstance()

{

return &m\_instance;

}

private:

Singleton(){};

Singleton(Singleton const&);

Singleton& operator=(Singleton const&);

static Singleton m\_instance;

};

Singleton Singleton::m\_instance; // 在程序入口之前就完成单例对象的初始化

虽然这种实现在一定程度下能良好工作，但是在某些情况下会带来问题 --- 就是在C++中非局部静态对象的 “初始化”顺序的不确定性， 参见Effective c++ 条款47。

考虑： 如果有两个这样的单例类，将分别生成单例对象A, 单例对象B. 它们分别定义在不同的编译单元（cpp中）， 而A的初始化依赖于B 【 即A的构造函数中要调用B::GetInstance() ，而此时B::m\_instance 可能还未初始化，显然调用结果就是非法的 】， 所以说只有B在A之前完成初始化程序才能正确运行，而这种跨编译单元的初始化顺序编译器是无法保证的。

**增强版饿汉模式（boost实现）不懂这个实现**

boost 的实现方式是：单例对象作为静态局部变量，然后增加一个辅助类，并声明一个该辅助类的类静态成员变量，在该辅助类的构造函数中，初始化单例对象。以下为代码

class Singleton

{

public:

static Singleton\* GetInstance()

{

static Singleton instance;

return &instance;

}

protected:

// 辅助代理类

struct Object\_Creator

{

Object\_Creator()

{

Singleton::GetInstance();

}

};

static Object\_Creator \_object\_creator;

Singleton() {}

~Singleton() {}

};

Singleton::Object\_Creator Singleton::\_object\_creator;

首先，代理类这个外部变量初始化时，在其构造函数内部调用Singleton::GetInstance();从而间接完成单例对象的初始化，这就通过该代理类实现了饿汉模式的特性。

其次，仍然考虑第三种模式的缺陷。 当A的初始化依赖于B，【 即A的构造函数中要调用B::GetInstance() ，而此时B::m\_instance 可能还未初始化，显然调用结果就是非法的 】 现在就变为【在A的构造函数中要调用B::GetInstance() ，如果B尚未初始化，就会引发B的初始化】，所以在不同编译单元内全局变量的初始化顺序不定的问题就随之解决。

42.new创建对象和不new的区别

1）内存位置不同，new在自由存储区（取决于operator new实现）,不new在栈区；

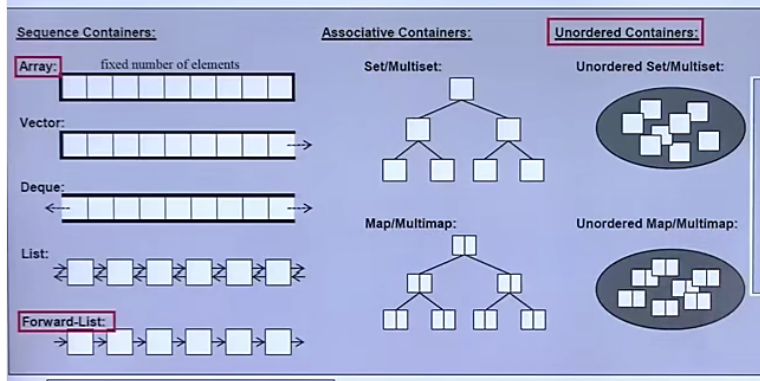
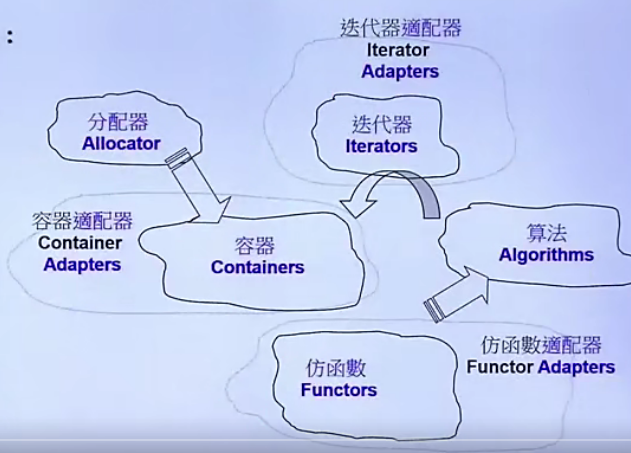
2）生存周期不同，new的对象除非手动释放，否则一直不会销毁，不new的离开作用域被销毁。

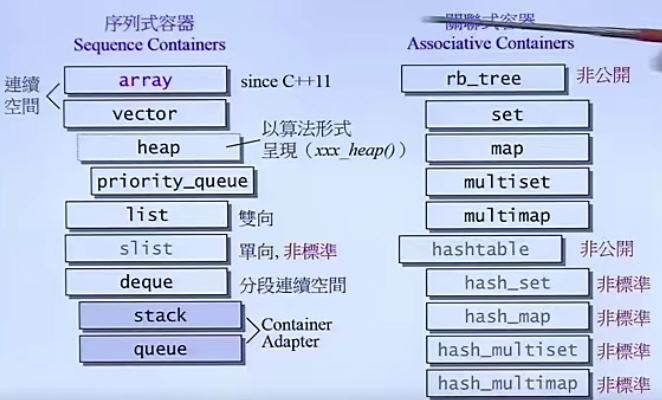
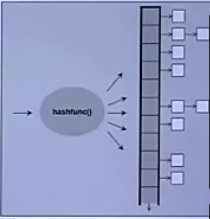
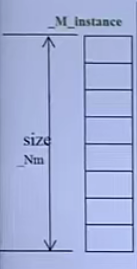
43.C++深拷贝与浅拷贝

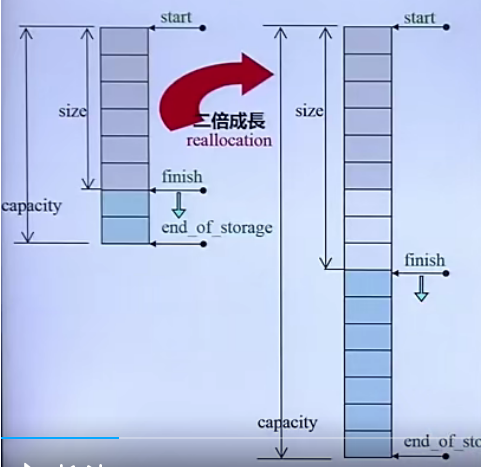
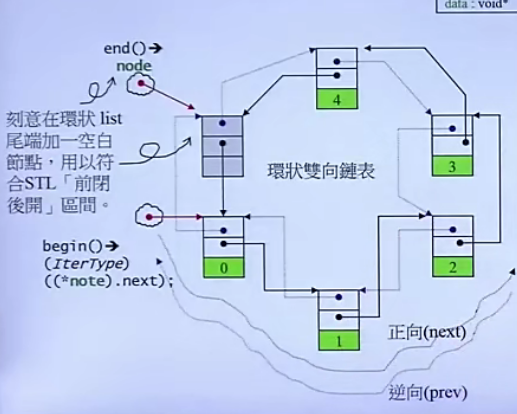
浅拷贝只拷贝指针，没有拷贝指针对应的动态成员内容。如果是存在实体共享的需求，应该用shared\_ptr代替普通指针；

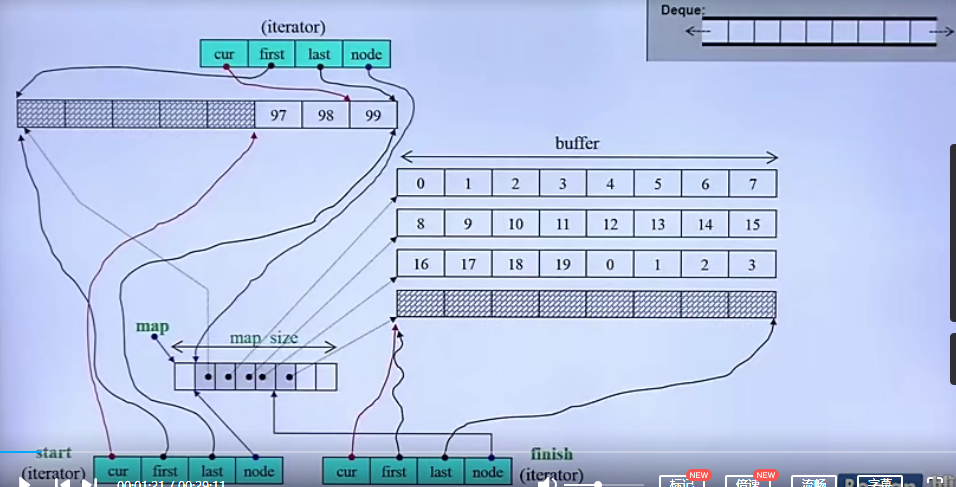
深拷贝则不仅仅拷贝普通成员，对应动态成员则重新生成一份拷贝，新的对象中地址指向新的内存单元，与原对象彻底解绑。

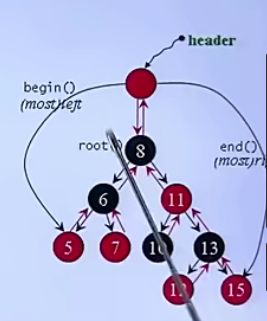
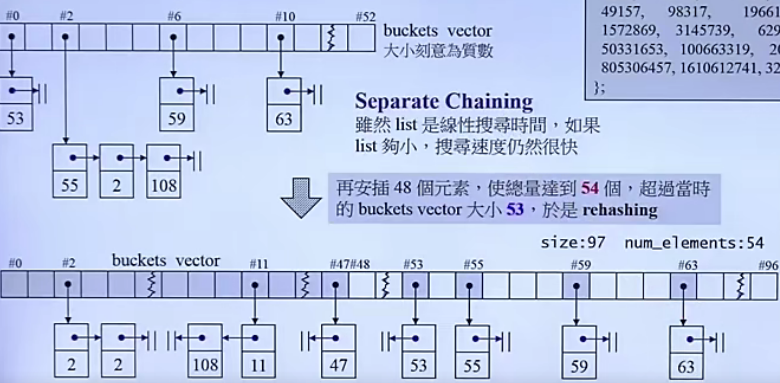
44. STL容器实现原理





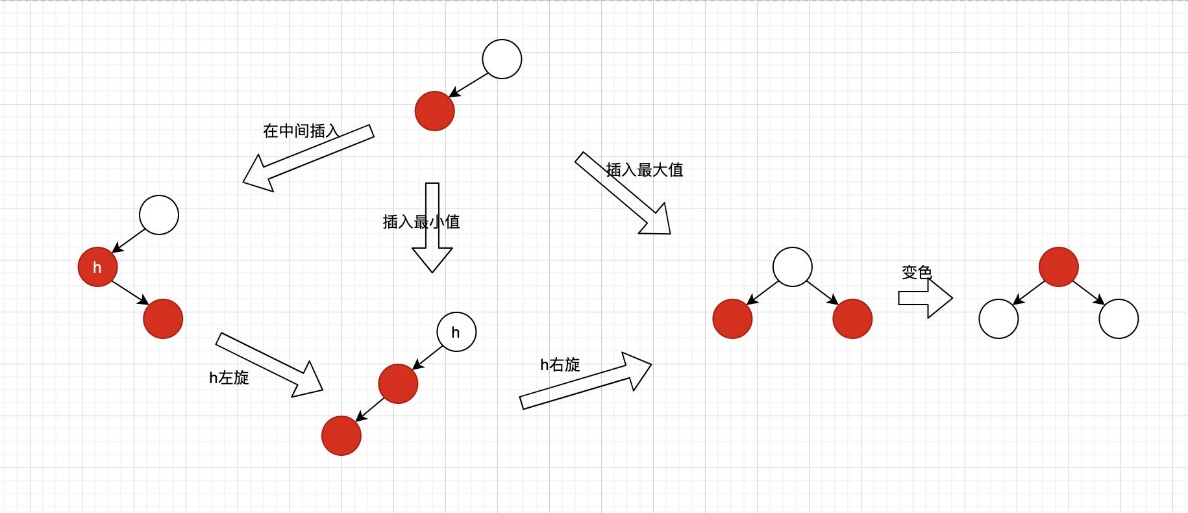


45.介绍下R-B tree实现

<https://segmentfault.com/a/1190000039774810>

插入操作如下，删除操作稍微复杂些，见链接介绍 基于2-3树实现，先要理解2-3树的理论



46.迭代器++iter和iter++区别

前者返回引用，效率比较高，后者需要先拷贝一个临时对象并返回，效率比较低（拷贝赋值，拷贝构造）

\_Myiter& operator++()

{ // preincrement

++\*(\_Mybase \*)this;

return (\*this);

}

\_Myiter operator++(int)

{ // postincrement

\_Myiter \_Tmp = \*this;

++\*this;

return (\_Tmp);

}

47.萃取机制

，iterator 是建立在 container 和 algorithm 间的桥梁，成为 container 和 algorithm 互相之间沟通交流的媒介，比方说，algorithm 希望知道更多关于所传入变量的信息，比方说数据类型，迭代器类型等等，那么 萃取 traits 就显得尤为重要了。

template<typename T>

void myFunction(T& coll) {

for (int i = 0; i < coll.size(); ++i) {

cout<<coll[i]<<endl;

}

这个函数很简单，就是希望传入一个 container ，然后把他们的每个值都打印出来，那么我传入 vector<int> , vector<string> 等等，这些都是可以的，这没问题。但是，如果我希望的到 T 这个 container 中的每个数据的数据类型，怎么办呢？ 这就需要 萃取机制了，也就是 iterator\_traits 了。我们在上面这个例子中加入一句：

typedef typename iterator\_traits<typename T::iterator>::value\_type vt;

首先看这句话，把模板的 container 类型的iterator传入iterator\_traits，问 iterator\_traits 索要value tyep，并定义为 vt ，以供后面使用，这就完成了萃取。

gcc2.9中iterator\_traits源码 每个迭代器要自己去定义这些typedef从而交给萃取机使用

template <class \_Iterator>

struct iterator\_traits {

typedef typename \_Iterator::iterator\_category iterator\_category;

typedef typename \_Iterator::value\_type value\_type;

typedef typename \_Iterator::difference\_type difference\_type;

typedef typename \_Iterator::pointer pointer;

typedef typename \_Iterator::reference reference;

};

当然，iterator\_traits 不仅可以萃取 这个iterator 的 value type，还可以萃取出 category，difference type，pointer，reference的信息，看最上面的定义就知道了。这里再说明一下，为什么要用 category 对不同 container 的 iterator进行分类呢？

using \_\_STD::input\_iterator\_tag;

using \_\_STD::output\_iterator\_tag;

using \_\_STD::forward\_iterator\_tag;

using \_\_STD::bidirectional\_iterator\_tag;

using \_\_STD::random\_access\_iterator\_tag;

原因很简单，这都是为 algorithm 服务的，因为不同的算法，比方说 move 类型的算法，对于 forward\_iterator\_tag 类型的container，只能向前move，例如forward\_list，而对于 bidirectional\_iterator\_tag 类型的container，比方说 deque，就可以两头移动，所以需要对此进行分类。

最后，除了 iterator\_traits 以外，还可以对很多东西进行萃取，比方说 type traits，char traits，allocator traits，pointer traits，array traits。

后续和操作系统，计算机网络再来一起讲

48.grpc的优点

49.语言线程支持std::thread

50.操作系统方面进程和线程的区别？