目录

1. C++程序从写代码到显示出结果共包括哪四个步骤 4

2. 在main执行之前和之后执行的代码可能是什么？ 4

3. 结构体内存对齐问题？ 4

4. 指针和引用的区别 4

5. 基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间 6

6. new / delete 与 malloc / free的异同 6

7. free(void \*ptr)怎么正确释放指针 6

8. delete和delete[]区别？ 6

9. 宏定义和函数有何区别？ 6

10. 宏定义和typedef区别？ 7

11. 变量声明和定义区别？ 7

12. strlen和sizeof区别？ 7

13. 常量指针和指针常量区别？ 7

14. a和&a有什么区别？ 7

15. 数组名和指针（这里为指向数组首元素的指针）区别？ 8

16. 野指针和悬空指针 8

17. 迭代器失效的情况 8

18. C++内存模型 8

19. 缺省参数概念： 9

20. C和C++的区别 9

21. C++中struct和class的区别 9

22. define宏定义和const的区别 9

23. C++中const和static的作用及其所处程序区域 10

24. 类的对象存储空间？ 11

25. 拷贝初始化和直接初始化 11

26. 模板函数和模板类的特例化 12

27. C和C++的类型安全 13

28. 为什么析构函数一般写成虚函数 14

29. 虚函数的开销（表面上和底层CPU指令流水线、分支预测器） 15

30. 构造函数能否声明为虚函数或者纯虚函数，析构函数呢？ 16

31. C++中的重载、重写（覆盖）和隐藏的区别 18

32. C++的多态如何实现 19

33. C++有哪几种的构造函数 21

34. 浅拷贝和深拷贝的区别 22

35. 内联函数和宏定义的区别 22

36. 构造函数、析构函数、虚函数可否声明为内联函数 23

37. auto、decltype和decltype(auto)的用法 24

38. public，protected和private访问和继承权限/public/protected/private的区别？ 25

39. 如何用代码判断大小端存储 25

40. C++中NULL和nullptr区别 26

41. 简要说明C++的内存分区 27

42. C++的异常处理的方法 28

43. static的用法和作用？ 30

44. const关键字 31

45. static关键字 33

46. 继承条件下对象构造析构顺序是否可以改变 即先构造子类再构造父类 35

47. C++面向对象的三个基本特征：封装、继承、多态 35

48. 菱形继承的相关问题 35

49. 虚函数表（多态实现的原理） 37

50. 两个进程中，申请的指针，可能指向同一片内存吗 37

51. 代码中的变量是逻辑地址还是物理地址 37

52. 同一个进程下的两个线程a和b，在线程a中有一个static变量，线程b能访问得到吗？在线程a中new一个对象，线程b能访问的到吗？ 37

53. 函数指针和指针函数（函数指针的优势编译的过程怎么链接的 ） 38

54. lambda匿名函数原理（待完善） 38

55. sizeof和strlen的区别，sizeof计算结构体大小（结构体对齐！）（自己尝试一下）一个空类的大小，一个具有虚函数类的大小。 38

56. strcpy的缺点，与之相关的memcpy缺点（待完善） 39

57. 32位系统malloc最大申请内存多少(malloc最大申请内存) 39

58. 传输参数是数组，不是vector，如何判断数组的大小？ 39

59. 类型转换，有哪些，用过哪些？ 40

60. 4个智能指针，应用场景 40

61. C++深克隆（要实现深克隆要实现C++成员函数里的什么函数）memcpy 42

62. 内存对齐了解吗？给一个结构体，包括char、int指针，sizeof有多大？为什么？如果想紧凑对齐应该怎么做？32位LINUX是怎么内存对齐？64位呢？ 42

63. C++静态库和动态库 42

64. C++防止头文件被重复引入 42

65. #include<>和#include””的区别 43

66. Resize和reserve的区别 43

43

67. 左值右值，右值引用、移动语义和完美转发。 43

68. 空类大小为1，默认实现了什么？ 46

69. C++中类如果没有自定义构造函数，系统会默认为类提供什么构造函数 46

70. 析构函数为什么不能抛出异常 46

71. new失败了会返回什么？ 46

72. volatile关键字的作用 46

73. 用valgrind定位内存泄漏 47

75. 内存泄漏解决办法 50

76. C++中的静态绑定和动态绑定（多态） 50

77. std::pair 51

78. 结构体设计思想 52

79. 泛型编程（C++如何实现泛型编程、什么是泛型编程、优缺点） 52

80. typename 和 class的区别 53

81. 多态和泛型 53

82. 静态链接库和动态链接库（巩固） 54

83. C++ 类内存分布 56

84. C++聊强行访问私有成员，答按地址访问，反问能不能用优雅地强行访问 57

85. malloc 底层实现 57

malloc 底层原理 57

86. new、delete操作符的底层实现 new 61

87. c++如何定义一个只能在堆上（栈上）生成对象的类？ 62

88. 怎么用一个指向子类的基类指针调用基类的虚函数 63

89. C++ vector插入和删除为什么会导致迭代器实效 63

90. 什么是面向对象编程？ 63

91. Lambda表达式 64

92. c++匿名函数有没有地址 65

93. STL如何实现哈希表？（\*\*从哈希方法，到哈希冲突解决，到哈希表扩容，以及为什么要用质数\*\*） 65

94. std::move()的实现原理 65

95. 右值引用和move语义（巩固）、移动语义和完美转发 65

96. gdb调试的多线程调试（去自己的项目中尝试） 67

97. C++数据类型大小 67

98. 67

99. 67

1. C++程序从写代码到显示出结果共包括哪四个步骤

第一个阶段：**预处理阶段**，在正式的编译阶段之前进行。预处理阶段将根据已放置在文件中的预处理指令来修改源文件的内容。如#include指令就是一个预处理指令，它把**头文件的内容添加到.c或.cpp文件**中，**注释去除**，**头文件展开**，**宏定义替换**、**条件编译**（**去除重复头文件**）。

第二个阶段：**编译**、优化阶段，将其翻译成等价的中间代码表示或汇编代码，并执行优化。

第三个阶段：**汇编**，把汇编语言代码翻译成目标机器指令。

第四个阶段：**链接**，例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

1. 在main执行之前和之后执行的代码可能是什么？

**main函数执行之前**，主要就是初始化系统相关资源：

* 设置栈指针
* 初始化静态static变量和global全局变量，即.data段的内容
* 将未初始化部分的全局变量赋初值：数值型short，int，long等为0，bool为FALSE，指针为NULL等等，即.bss段的内容
* 全局对象初始化，在main之前调用构造函数，这是可能会执行前的一些代码
* 将main函数的参数argc，argv等传递给main函数，然后才真正运行main函数

**main函数执行之后**：

* **全局对象的析构函数会在main函数之后执行；**
* 可以用 **atexit**注册一个函数，它会在main 之后执行;

1. 结构体内存对齐问题？

* 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
* 未特殊说明时，按结构体中size最大的成员对齐（若有double成员，按8字节对齐。）

1. 指针和引用的区别

* 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
* 指针可以有多级，引用只有一级
* 指针可以为空，引用不能为NULL且在定义时必须初始化
* 指针在初始化后可以改变指向，而引用在初始化之后不可再改变
* sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小
* **当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。**
* 引用只是别名，不占用具体存储空间，只有声明没有定义；指针是具体变量，需要占用存储空间。
* 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
* 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
* 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

void test(int \*p)

{

　　int a=1;

　　p=&a;

　　cout<<p<<" "<<\*p<<endl;

}

int main(void)

{

    int \*p=NULL;

    test(p);

    if(p==NULL)

    cout<<"指针p为NULL"<<endl;

    return 0;

}

//运行结果为：

//0x22ff44 1

//指针p为NULL

void testPTR(int\* p) {

    int a = 12;

    p = &a;

}

void testREFF(int& p) {

    int a = 12;

    p = a;

}

void main()

{

    int a = 10;

    int\* b = &a;

    testPTR(b);//改变指针指向，但是没改变指针的所指的内容

    cout << a << endl;// 10

    cout << \*b << endl;// 10

    a = 10;

    testREFF(a);

    cout << a << endl;//12

}

1. 基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间

*虚函数表存放在全局数据区。*所有基类构造函数之后，但又在自身构造函数或初始化列表之前

1. new / delete 与 malloc / free的异同

**相同点**

* 都可用于内存的动态申请和释放

**不同点**

* new、delete则为C++的操作运算符，后者是C/C++语言标准库函数（stdlib.h）
* new自动计算要分配的空间大小，malloc需要手工计算
* new是类型安全的，malloc不是。例如：

int \*p = new float[2]; *//编译错误*  
int \*p = (int\*)malloc(2 \* sizeof(double));*//编译无错误*

* new调用名为operator new的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为operator delete的标准库函数释放该对象所用内存。后者均没有相关调用
* 后者需要库文件支持，前者不用
* new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象

new分配内存处于堆区，malloc处于自由存储区

在类或函数中，int\* ，char\* 这些只要不是new的，也同样不用释放，系统会自动把他们占的内存释放掉，只有new的才会手动的去delete

1. free(void \*ptr)怎么正确释放指针

https://blog.csdn.net/bdss58/article/details/94005337

1. delete和delete[]区别？

* delete只会调用一次析构函数。
* delete[]会调用数组中每个元素的析构函数。

But 析构一个int、char等数据类型的数组，调用delete也可以将数组析构。而用户自定义的却不行。

1. 宏定义和函数有何区别？

* 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时**不存在函数调用**，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
* 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
* 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
* 宏定义不要在最后加分号。

1. 宏定义和typedef区别？

* 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
* 宏替换发生在**编译阶段之前**，属于文本插入替换；**typedef是编译的一部分**。
* 宏不检查类型；typedef会检查数据类型。
* 宏不是语句，不在在最后加分号；typedef是语句，要加分号标识结束。
* 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

1. 变量声明和定义区别？

* extern int i；//声明但不定义
* int i;//声明也定义
* extern声明不是定义，也不分配存储空间。事实上，它只是说明变量定义在程序的其他地方。
* 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
* 相同变量可以在多处声明（外部变量extern），但只能在一处定义。

1. strlen和sizeof区别？

* sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
* sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
* 因为sizeof值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。

1. 常量指针和指针常量区别？

* **常量指针是一个指针，读成常量的指针，指向一个只读变量。如int const \*p或const int \*p。**
* **指针常量是一个不能给改变指向的指针。指针是个常量，不能中途改变指向，如int \*const p。**

1. a和&a有什么区别？

假设数组int a[10];  
int (\*p)[10] = &a;

a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。

* &a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。
* 若(int \*)p ，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。

1. 数组名和指针（这里为指向数组首元素的指针）区别？

* 二者均可通过增减偏移量来访问数组中的元素。
* 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
* 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

1. 野指针和悬空指针

都是是指向无效内存区域(这里的无效指的是"不安全不可控")的指针，访问行为将会导致未定义行为。

野指针，指的是没有被初始化过的指针

int main(void) {   
  int \* p;  
  std::cout<<\*p<<std::endl; return 0;  
}

因此，为了防止出错，对于指针初始化时都是赋值为 nullptr，这样在使用时编译器就会直接报错，产生非法内存访问。

悬空指针，指针最初指向的内存已经被释放了的一种指针。

int main(void) {   
  int \* p = nullptr;  
  int\* p2 = new int;   
  p = p2;  
  delete p2;  
}

此时 p和p2就是悬空指针，指向的内存已经被释放。继续使用这两个指针，行为不可预料。需要设置为p=p2=nullptr。此时再使用，编译器会直接保错。

避免野指针比较简单，但悬空指针比较麻烦。c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生。

产生原因及解决办法：

野指针：指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。

悬空指针：指针free或delete之后没有及时置空 => 释放操作后立即置空

1. 迭代器失效的情况

以vector为例：

**插入元素：**

1、尾后插入：size < capacity时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity时，所有迭代器均失效（需要重新分配空间）。

2、中间插入：中间插入：size < capacity时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity时，所有迭代器均失效。

**删除元素：**

尾后删除：只有尾迭代失效。

中间删除：删除位置之后所有迭代失效。

1. C++内存模型
2. 缺省参数概念：

 (1)缺省参数是声明或定义函数时为函数的参数指定一个默认值 ；(2)在调用该函数时，如果没有指定实参则采用该默认值，否则使用指定的实参。

1. C和C++的区别

* C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
* 标准C++中的字符串类取代了标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。
* C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
* C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
* 在C++中，允许有相同的函数名（与函数存储方式相关），不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。也就是C++可以重载，C语言不允许。
* C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而C语言中，必须要在函数开头部分。而且C++允许重复定义变量，C语言也是做不到这一点的
* 在C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
* C++相对与C增加了一些关键字，如：bool、using、dynamic\_cast、namespace等等

1. C++中struct和class的区别

**相同点**

* 两者都拥有成员函数、公有和私有部分
* 任何可以使用class完成的工作，同样可以使用struct完成

**不同点**

* 两者中如果不对成员不指定公私有，struct默认是公有的，class则默认是私有的
* class默认是private继承，而struct模式是public继承
* class可以作为模板类型，struct不行

**引申**：C++和C的struct区别

* C语言中：struct是用户自定义数据类型（UDT）；C++中struct是抽象数据类型（ADT），支持成员函数的定义，（C++中的struct能继承，能实现多态）
* C中struct是没有权限的设置的，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数
* C++中，struct增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为public（为了与C兼容）
* struct作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在C中必须在结构标记前加上struct，才能做结构类型名（除：typedef struct class{};）;C++中结构体标记（结构体名）可以直接作为结构体类型名使用，此外结构体struct在C++中被当作类的一种特例

1. define宏定义和const的区别

**编译阶段**

* define是在编译的**预处理**阶段起作用，而const是在编译、运行的时候起作用

**安全性**

* define只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包含住全部的内容，要不然很容易出错
* const常量有数据类型，编译器可以对其进行类型安全检查

**内存占用**

* define只是将宏名称进行替换，在内存中会产生多分相同的备份。const在程序运行中只有一份备份，且可以执行常量折叠，能将复杂的的表达式计算出结果放入常量表
* 宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。
* 宏不检查类型；const会检查数据类型。
* 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

1. C++中const和static的作用及其所处程序区域

**static**

* 不考虑类的情况
* 隐藏。所有不加static的全局变量和函数具有全局可见性，可以在其他文件中使用，加了之后只能在该文件所在的编译模块中使用
* 默认初始化为0，包括未初始化的全局静态变量与局部静态变量，都存在全局未初始化区
* 静态变量在函数内定义，始终存在，且只进行一次初始化，具有记忆性，其作用范围与局部变量相同，函数退出后仍然存在，但不能使用
* 考虑类的情况
* static成员变量：只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始化，必须在类定义体外部初始化，初始化时不需要标示为static；可以被非static成员函数任意访问。
* static成员函数：不具有this指针，无法访问类对象的非static成员变量和非static成员函数；**不能被声明为const、虚函数和volatile**；可以被非static成员函数任意访问

**const**

* 不考虑类的情况
* const常量在定义时必须初始化，之后无法更改
* const形参可以接收const和非const类型的实参，例如

*// i 可以是 int 型或者 const int 型*  
void fun(const int& i){  
    *//...*  
}

* 考虑类的情况
* const成员变量：不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且必须有构造函数；不同类对其const数据成员的值可以不同，所以不能在类中声明时初始化
* const成员函数：const对象不可以调用非const成员函数；非const对象都可以调用；不可以改变非mutable（用该关键字声明的变量可以在const成员函数中被修改）数据的值

1. static无论是全局变量还是局部变量都存储在全局/静态区域，在编译期就为其分配内存，在程序结束时释放，例如：val\_a、val\_d、val\_h、val\_i。
2. const全局变量存储在只读数据段，编译期最初将其保存在符号表中，第一次使用时为其分配内存，在程序结束时释放，例如：val\_c；const局部变量存储在栈中，代码块结束时释放，例如：val\_j。
3. 全局变量存储在全局/静态区域，在编译期为其分配内存，在程序结束时释放，例如：val\_b、val\_e。
4. 局部变量存储在栈中，代码块结束时释放，例如：val\_h、val\_i。

注：当全局变量和静态局部变量未赋初值时，系统自动置为0。

const变量与define宏定义：

1、编译器处理方式不同。const变量在编译阶段处理，define宏在预处理阶段展开

2、类型和安全检查不同。

3、存储方式不同。define宏仅仅是展开，没有分配内存；const变量有内存地址

4、const 可以节省空间，避免不必要的内存分配。

5、 提高了效率。 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。

在C++ 程序中只使用const常量而不使用宏常量，即const常量完全取代宏常量。

1. 类的对象存储空间？

* 非静态成员的数据类型大小之和。
* 编译器加入的额外成员变量（如指向虚函数表的指针）。
* 为了边缘对齐优化加入的padding。

1. 拷贝初始化和直接初始化

* 当用于类类型对象时，初始化的拷贝形式和直接形式有所不同：直接初始化直接调用与实参匹配的构造函数，拷贝初始化总是调用拷贝构造函数。拷贝初始化首先使用指定构造函数创建一个临时对象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象。举例如下

string str1("I am a string");*//语句1 直接初始化*  
string str2(str1);*//语句2 直接初始化，str1是已经存在的对象，直接调用构造函数对str2进行初始化*  
string str3 = "I am a string";*//语句3 拷贝初始化，先为字符串”I am a string“创建临时对象，再把临时对象作为参数，使用拷贝构造函数构造str3*  
string str4 = str1;*//语句4 拷贝初始化，这里相当于隐式调用拷贝构造函数，而不是调用赋值运算符函数*

* **为了提高效率，允许编译器跳过创建临时对象这一步，**直接调用构造函数构造要创建的对象，这样就完全等价于**直接初始化了**（语句1和语句3等价）。但是需要辨别两种情况。
* 当拷贝构造函数为private时：语句3和语句4在编译时会报错
* 使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错

1. 模板函数和模板类的特例化

**引入原因**

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

**定义**

对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

**（1）模板函数特例化**

必须为原函数模板的每个模板参数都提供实参，且使用关键字template后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参，举例如下：

template<typename T> *//模板函数*  
int compare(const T &v1,const T &v2)  
{  
    if(v1 > v2) return -1;  
    if(v2 > v1) return 1;  
    return 0;  
}  
*//模板特例化,满足针对字符串特定的比较，要提供所有实参，这里只有一个T*  
template<>   
int compare(const char\* const &v1,const char\* const &v2)  
{  
    return strcmp(p1,p2);  
}

**本质**

特例化的本质是实例化一个模板，而非重载它。特例化不影响参数匹配。参数匹配都以最佳匹配为原则。例如，此处如果是compare(3,5)，则调用普通的模板，若为compare(“hi”,”haha”)则调用特例化版本（因为这个cosnt char\*相对于T，更匹配实参类型），注意二者函数体的语句不一样了，实现不同功能。

**注意**

模板及其特例化版本应该声明在同一个头文件中，且所有同名模板的声明应该放在前面，后面放特例化版本。

**（2）类模板特例化**

原理类似函数模板，不过在类中，我们可以对模板进行特例化，也可以对类进行部分特例化。对类进行特例化时，仍然用template<>表示是一个特例化版本，例如：

template<>  
class hash<sales\_data>  
{  
    size\_t operator()(sales\_data& s);  
    *//里面所有T都换成特例化类型版本sales\_data*  
    *//按照最佳匹配原则，若T != sales\_data，就用普通类模板，否则，就使用含有特定功能的特例化版本。*  
};

**类模板的部分特例化**

不必为所有模板参数提供实参，可以指定一部分而非所有模板参数，一个类模板的部分特例化本身仍是一个模板，使用它时还必须为其特例化版本中未指定的模板参数提供实参(特例化时类名一定要和原来的模板相同，只是参数类型不同，按最佳匹配原则，哪个最匹配，就用相应的模板)

**特例化类中的部分成员**

可以特例化类中的部分成员函数而不是整个类，举个例子：

template<typename T>  
class Foo  
{  
    void Bar();  
    void Barst(T a)();  
};  
  
template<>  
void Foo<int>::Bar()  
{  
    *//进行int类型的特例化处理*  
    cout << "我是int型特例化" << endl;  
}  
  
Foo<string> fs;  
Foo<int> fi;*//使用特例化*  
fs.Bar();*//使用的是普通模板，即Foo<string>::Bar()*  
fi.Bar();*//特例化版本，执行Foo<int>::Bar()*  
*//Foo<string>::Bar()和Foo<int>::Bar()功能不同*

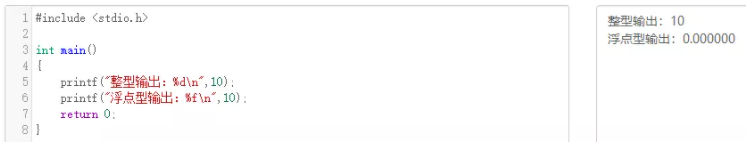
1. C和C++的类型安全

**什么是类型安全？**

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安全的程序。绝对类型安全的编程语言暂时还没有。

**（1）C的类型安全**

C只在局部上下文中表现出类型安全，比如试图从一种结构体的指针转换成另一种结构体的指针时，编译器将会报告错误，除非使用显式类型转换。然而，C中相当多的操作是不安全的。以下是两个十分常见的例子：



上述代码中，使用%d控制整型数字的输出，没有问题，但是改成%f时，明显输出错误，再改成%s时，运行直接报segmentation fault错误

* malloc函数的返回值

malloc是C中进行内存分配的函数，它的返回类型是void\*即空类型指针，常常有这样的用法char\* pStr=(char\*)malloc(100\*sizeof(char))，这里明显做了显式的类型转换。类型匹配尚且没有问题，但是一旦出现int\* pInt=(int\*)malloc(100\*sizeof(char))就很可能带来一些问题，而这样的转换C并不会提示错误。

**（2）C++的类型安全**

如果C++使用得当，它将远比C更有类型安全性。相比于C语言，C++提供了一些新的机制保障类型安全：

* 操作符new返回的指针类型严格与对象匹配，而不是void\*
* C中很多以void\*为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；
* 引入const关键字代替#define constants，它是有类型、有作用域的，而#define constants只是简单的文本替换
* 一些#define宏可被改写为inline函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全
* C++提供了**dynamic\_cast**关键字，使得转换过程更加安全，因为dynamic\_cast比static\_cast涉及更多具体的类型检查。

1. 为什么析构函数一般写成虚函数

**由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。举个例子：**



1. 虚函数的开销（表面上和底层CPU指令流水线、分支预测器）

虚函数的实现

虽然C++标准并没有规定编译器实现虚函数的方式，但是大部分编译器均是采用了虚函数表来实现虚函数，即对于每一个包含虚成员函数的类生成一个虚函数表，一个指向虚函数表的指针被放在对象的首地址（不考虑多继承等复杂情况），虚函数表中存储该类所有的虚函数地址。当使用引用或者指针调用虚函数时，首先通过虚函数表指针找到虚函数表，然后通过偏移量找到虚函数地址并调用。关于虚函数表的更多细节，建议阅读《深度探索C++对象模型》这本书。

虚函数表面上的开销

空间开销首先，由于需要为每一个包含虚函数的类生成一个虚函数表，所以程序的二进制文件大小会相应的增大；其次，对于包含虚函数的类的实例来说，每个实例都包含一个虚函数表指针用于指向对应的虚函数表，所以每个实例的空间占用都增加一个指针大小（32位系统4字节，64位系统8字节）。这些空间开销可能会造成缓存的不友好，在一定程度上影响程序性能。时间开销虚函数的时间开销主要是增加了一次内存寻址，通过虚函数表指针找到虚函数表，虽对程序性能有一些影响，但是影响并不大。

上述虚函数表面上的开销其实是微不足道的，真正影响虚函数性能的是隐藏在背后的，不被人轻易察觉的，只有对计算机体系结构有一定理解才能探寻出藏在背后的“性能杀手”。首先我们先看调用虚函数时，在汇编层生成了什么代码：

...

movq (%rax), %rax

movq (%rax), %rax

movq -24(%rbp), %rdx

movq %rdx, %rdi

call \*%rax

...

上述汇编代码最重要的就是第6行，在AT&T格式汇编中，这是一个间接调用，意义是从%rax指明的地址处读取跳转的目标位置。这也是虚函数调用与普通成员函数的区别所在，普通函数调用是一个直接调用。直接调用与间接调用的区别就是跳转地址是否确定，直接调用的跳转地址是编译器确定的，而间接调用是运行到该指令时从寄存器中取出地址然后跳转。

有了上面的基本认识，我们就可以分析虚函数的性能开销所在了，其实说到底，这个隐藏在背后的关键点就是分支预测器，如果看过我之前的博客，相信对分支预测器已经很熟悉了，如果感觉分支预测器还是很陌生，推荐阅读我以前的分支预测器的四篇文章：

有了分支预测器和CPU指令流水线的基本知识，我们可以发现对于直接调用而言，是不存在分支跳转的，因为跳转地址是编译器确定的，CPU直接去跳转地址取后面的指令即可，不存在分支预测，这样可以保证CPU流水线不被打断。而对于间接寻址，由于跳转地址不确定，所以此处会有多个分支可能，这个时候需要分支预测器进行预测，如果分支预测失败，则会导致流水线冲刷，重新进行取指、译码等操作，对程序性能有很大的影响。网上有部分文章中说对于虚函数这种间接跳转会直接导致流水线冲刷，这种说法明显是自相矛盾的，如果间接跳转必定会导致流水线冲刷，那把这些指令放进流水线的意义何在呢？其实查阅资料就可以知道，Intel和AMD的CPU中存在两级自适应预测器用于预测间接跳转，此预测器可以预测多分支跳转。

本文探究出影响到虚函数调用性能的背后原因是流水线和分支预测，由于虚函数调用需要间接跳转，所以会导致虚函数调用比普通函数调用多了分支预测的过程，产生性能差距的原因主要是分支预测失败导致的流水线冲刷性能开销。

本文的目的并不是为了说明虚函数调用有额外开销而让大家避免使用虚函数，使用不使用虚函数应该由自己程序的需要而定，如果程序逻辑需要使用动态绑定，如果不使用虚函数而是自己实现相应逻辑的话产生的性能损耗一般会比使用虚函数的性能损耗大得多。但对于一些性能敏感的程序，在虚函数可用可不用的时候，可以考虑不使用虚函数以提高性能。

1. 构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

**析构函数**：

* 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
* 只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
* 析构函数可以是纯虚函数，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。

**构造函数**：

* 构造函数不能定义为虚函数。在构造函数中可以调用虚函数，不过此时调用的是正在构造的类中的虚函数，而不是子类的虚函数，因为此时子类尚未构造好。
* 1、 从存储空间角度，虚函数相应一个指向vtable虚函数表的指针，这大家都知道，但是这个指向vtable的指针事实上是存储在对象的内存空间的。
* 问题出来了，假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。
* 2、 从使用角度，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。
* 构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。
* 所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
* 3、构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。
* 4、从实现上看，vbtl在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。
* 5、当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的VPTR。
* 因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。
* 并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE, 但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的 VTABLE，等.直到最后的构造函数结束。
* VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置VPTR指向它自己的VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的VTABLE的调用，而不是最后的VTABLE（全部构造函数被调用后才会有最后的VTABLE）。
* 因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。
* 直接的讲，**C++中基类采用virtual虚析构函数是为了防止内存泄漏**。
* 具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。
* 所以，为了防止这种情况的发生，**C++中基类的析构函数应采用virtual虚析构函数**。

1. C++中的重载、重写（覆盖）和隐藏的区别

（1）重载（overload）

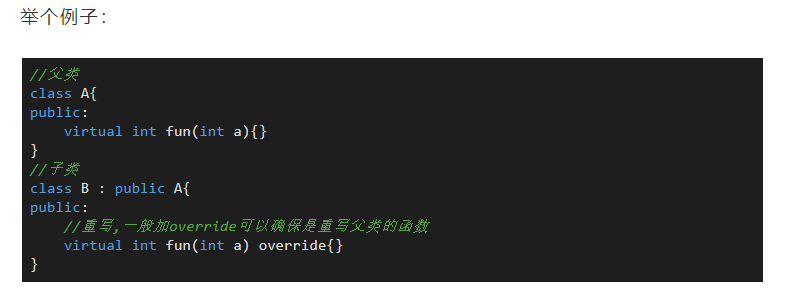
重载是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目有所不同，不能出现参数个数和类型均相同，仅仅依靠返回值不同来区分的函数。重载和函数成员是否是虚函数无关。举个例子：



（2）重写（覆盖）（override）

重写指的是在派生类中覆盖基类中的同名函数，重写就是重写函数体**，**要求基类函数必须是虚函数且：

* 与基类的虚函数有相同的参数个数
* 与基类的虚函数有相同的参数类型
* 与基类的虚函数有相同的返回值类型



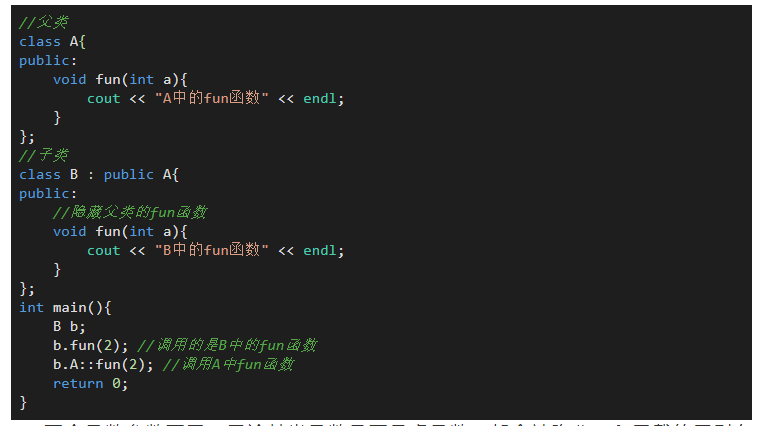
重载与重写的区别：

* 重写是父类和子类之间的垂直关系，重载是不同函数之间的水平关系
* 重写要求参数列表相同，重载则要求参数列表不同，返回值不要求
* 重写关系中，调用方法根据对象类型决定，重载根据调用时实参表与形参表的对应关系来选择函数体

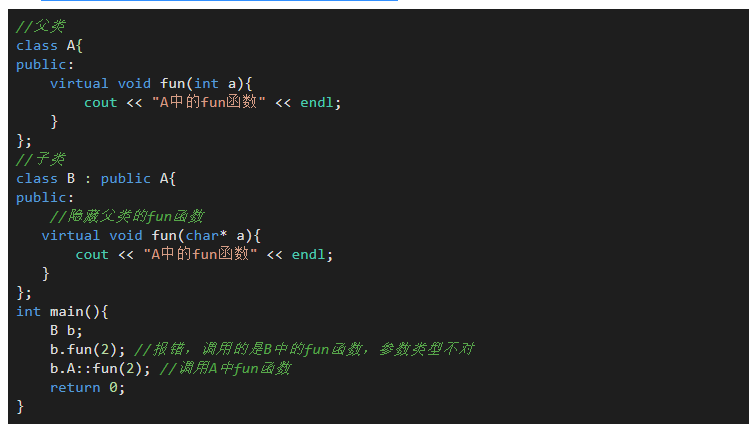
（3）隐藏（hide）

隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

* 两个函数参数相同，但是基类函数不是虚函数。**和重写的区别在于基类函数是否是虚函数。**举个例子：



两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。和重载的区别在于两个函数不在同一个类中。举个例子：

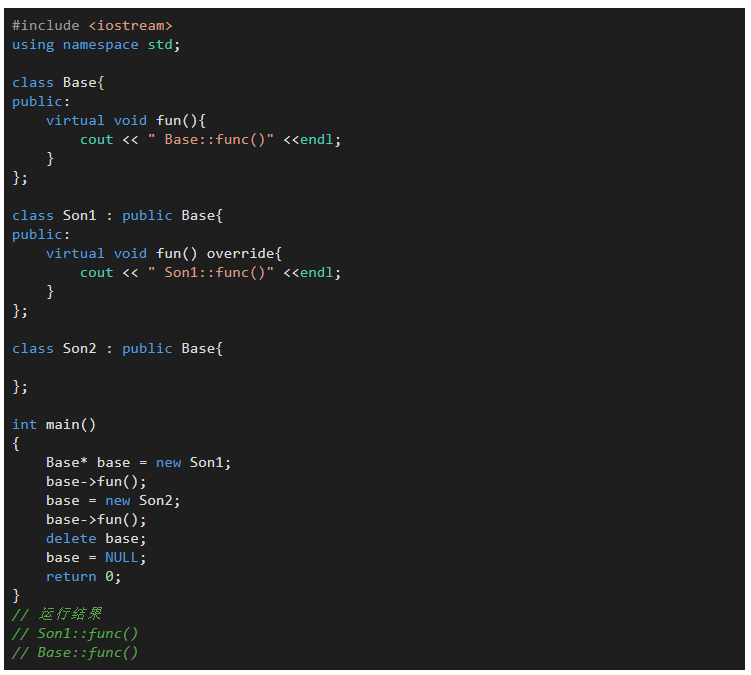


1. C++的多态如何实现

C++的多态性，**一言以蔽之**就是：

在基类的函数前加上**virtual**关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

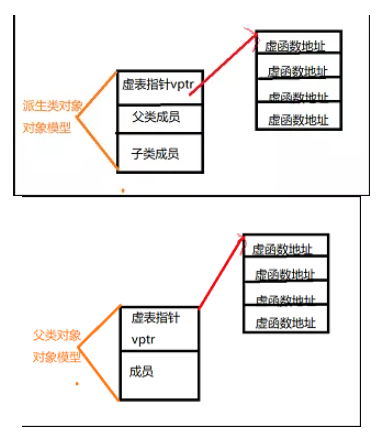
举个例子：



例子中，Base为基类，其中的函数为虚函数。子类1继承并重写了基类的函数，子类2继承基类但没有重写基类的函数，从结果分析子类体现了多态性，那么为什么会出现多态性，其底层的原理是什么？这里需要引出虚表和虚基表指针的概念。

虚表：虚函数表的缩写，类中含有virtual关键字修饰的方法时，编译器会自动生成虚表

虚表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针



上图中展示了虚表和虚表指针在基类对象和派生类对象中的模型**，**下面阐述**实现多态**的过程**：**

(1)编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维数组，虚表里保存了虚函数的入口地址

(2)编译器会在每个对象的前四个字节中保存一个虚表指针，即vptr，指向对象所属类的虚表。在构造时，根据对象的类型去初始化虚指针vptr，从而让vptr指向正确的虚表，从而在调用虚函数时，能找到正确的函数

(3)所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时，编译器只“看到了”父类，并为父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表

(4)当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚表中将此虚函数地址添加在后面

这样指向派生类的基类指针在运行时，就可以根据派生类对虚函数重写情况动态的进行调用，从而实现多态性。

1. C++有哪几种的构造函数

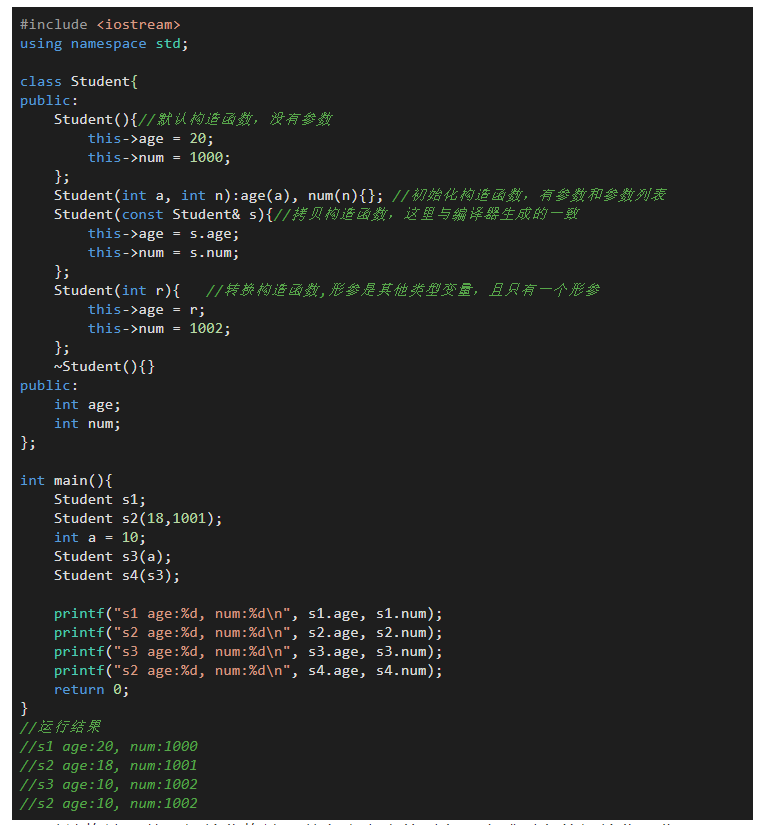
C++中的构造函数可以分为4类：

（1）默认构造函数。

（2）初始化构造函数

（3）复制（拷贝）构造函数

（4）转换构造函数



* 默认构造函数和初始化构造函数在定义类的对象，完成对象的初始化工作
* 复制构造函数用于复制本类的对象
* 转换构造函数用于将其他类型的变量，隐式转换为本类对象

具体转换构造函数：

一个构造函数接收一个不同于其类类型的形参，可以视为将其形参转换成类的一个对象。像这样的构造函数称为转换构造函数。

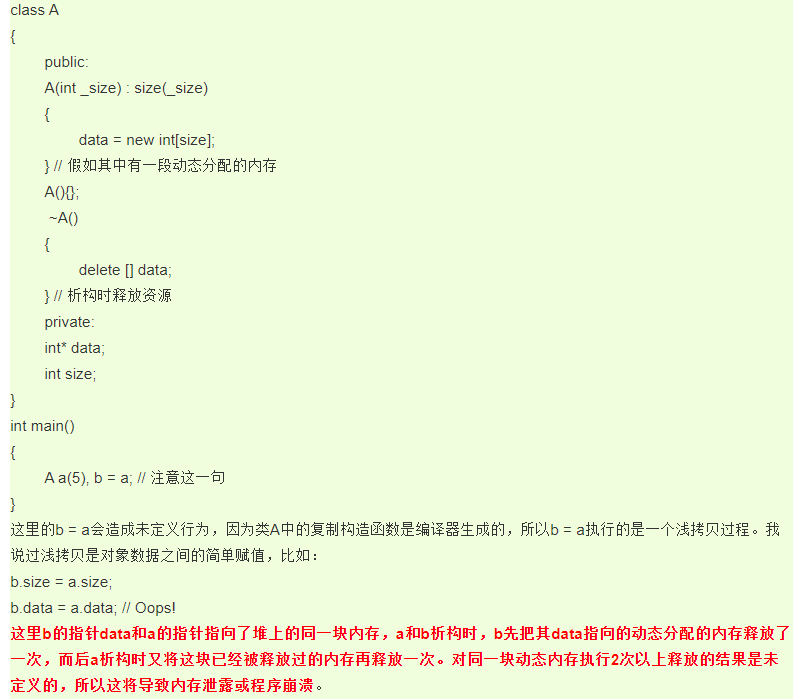
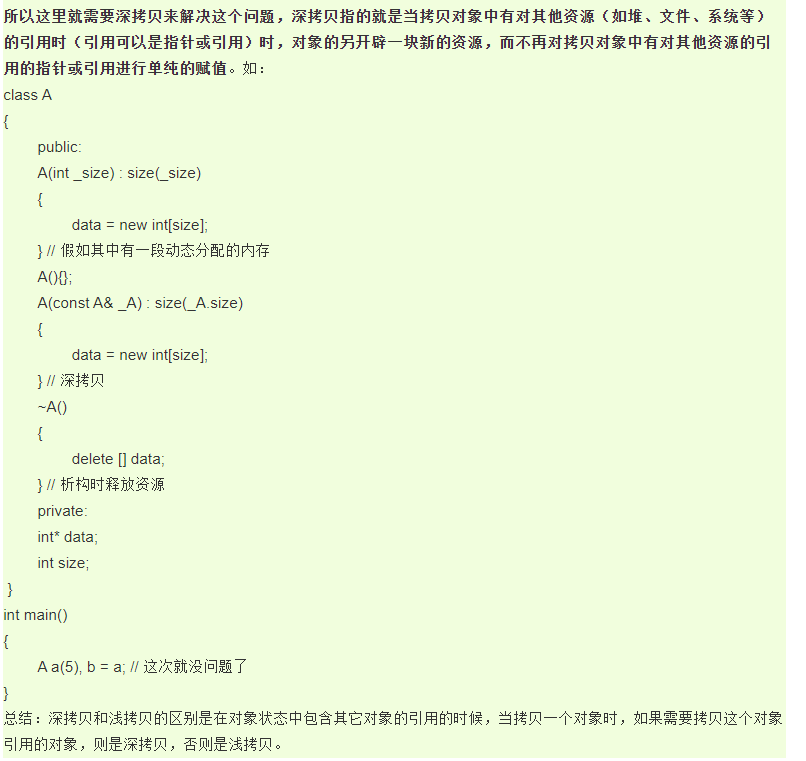
除了创建类对象之外，转换构造函数还为编译器提供了执行隐式类型转换的方法。只要在需要类的类型值的地方，给定构造函数的形参类型的值，就将由编译器执行这种类型的转换。

1. 浅拷贝和深拷贝的区别

一：两个的区别

1 在未定义显示拷贝构造函数的情况下，系统会调用默认的拷贝函数——即浅拷贝，它能够完成成员的一一复制。当数据成员中没有指针时，浅拷贝是可行的；但当数据成员中有指针时，如果采用简单的浅拷贝，则两类中的两个指针将指向同一个地址，当对象快结束时，会调用两次析构函数，而导致**指针悬挂现象**，所以，此时，必须采用深拷贝。

2 深拷贝与浅拷贝的区别就在于**深拷贝会在堆内存中另外申请空间**来储存数据，从而也就**解决了指针悬挂**的问题。简而言之，**当数据成员中有指针时，必须要用深拷贝**。

深拷贝和浅拷贝的区别是在对象状态中包含其它对象的引用的时候，当拷贝一个对象时，如果需要拷贝这个对象引用的对象，则是深拷贝，否则是浅拷贝。

1. 内联函数和宏定义的区别

内联(inline)函数和普通函数相比可以加快程序运行的速度，**因为不需要中断调用**，在编译的时候内联函数可以直接嵌入到目标代码中。

**内联函数适用场景**

* 使用宏定义的地方都可以使用inline函数
* 作为类成员接口函数来读写类的私有成员或者保护成员，会提高效率

**为什么不能把所有的函数写成内联函数**

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。所以一方面如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；另一方面每一处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数：

* 函数体内的代码比较长，将导致内存消耗代价
* 函数体内有循环，函数执行时间要比函数调用开销大

**主要区别**

* 内联函数在**编译**时展开，宏在**预编译**时展开
* 内联函数**直接嵌入到目标代码**中，**宏是简单的做文本替换**
* 内联函数有类型检测、语法判断等功能，而宏没有
* 内联函数是函数，宏不是
* 宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义
* 内联函数代码是被放到符号表中，使用时像宏一样展开，没有调用的开销，效率很高；

《inline函数和宏定义区别 整理》：https://blog.csdn.net/wangliang888888/article/details/77990650

* 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
* 内联函数本身是函数，强调函数特性，具有重载等功能。
* 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员，进而提升效率。**而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了**。

1. 构造函数、析构函数、虚函数可否声明为内联函数

首先，将这些函数声明为内联函数，在语法上没有错误。因为inline同register一样，只是个建议，编译器并不一定真正的内联。

register关键字：这个关键字请求编译器尽可能的将变量存在CPU内部寄存器中，而不是通过内存寻址访问，以提高效率

举个例子：



**构造函数和析构函数声明为内联函数是没有意义的**

《Effective C++》中所阐述的是：将构造函数和析构函数声明为inline是没有什么意义的，即编译器并不真正对声明为inline的构造和析构函数进行内联操作，因为编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简。其次，class中的函数默认是inline型的，编译器也只是有选择性的inline，将构造函数和析构函数声明为内联函数是没有什么意义的。

**将虚函数声明为inline，要分情况讨论（难点）**

有的人认为虚函数被声明为inline，但是编译器并没有对其内联，他们给出的理由是inline是编译期决定的，而虚函数是运行期决定的，即在不知道将要调用哪个函数的情况下，如何将函数内联呢？

上述观点看似正确，其实不然，如果虚函数在编译器就能够决定将要调用哪个函数时，就能够内联，那么什么情况下编译器可以确定要调用哪个函数呢，答案是当用对象调用虚函数（此时不具有多态性）时，就内联展开

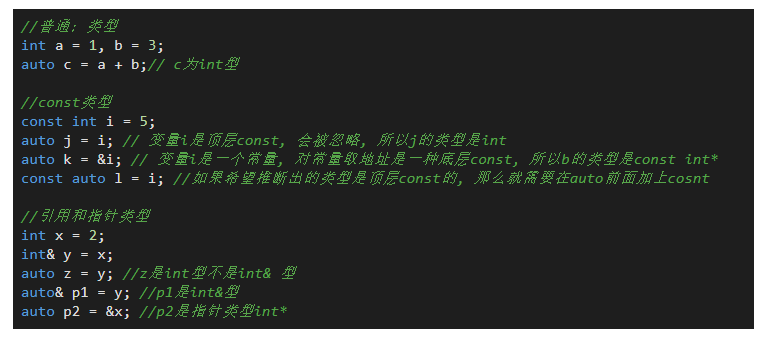
综上，当是指向派生类的指针（多态性）调用声明为inline的虚函数时，不会内联展开；当是对象本身调用虚函数时，会内联展开，当然前提依然是函数并不复杂的情况下

1. auto、decltype和decltype(auto)的用法

**（1）auto**

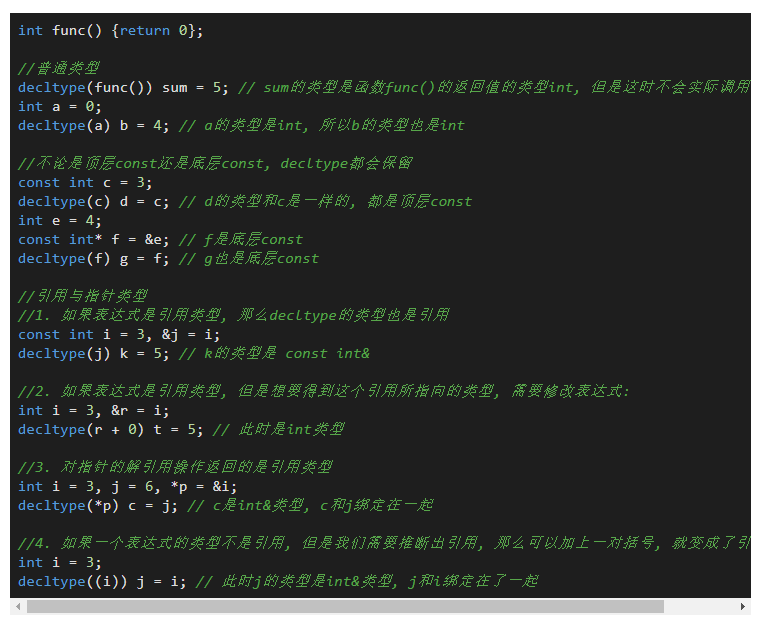
C++11新标准引入了auto类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应某种特定的类型说明符(例如 int)不同，

auto 让编译器通过初始值来进行类型推演。从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。举个例子：**相当于编译器调用了模板函数**



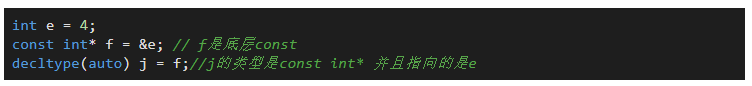
**（2）decltype**

有的时候我们还会遇到这种情况，**我们希望从表达式中推断出要定义变量的类型，但却不想用表达式的值去初始化变量。还有可能是函数的返回类型为某表达式的值类型。在这些时候auto显得就无力了，所以C++11又引入了第二种类型说明符decltype，它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。**



**（3）decltype(auto)**

decltype(auto)是C++14新增的类型指示符，可以用来声明变量以及指示函数返回类型。在使用时，会将“=”号左边的表达式替换掉auto，再根据decltype的语法规则来确定类型。举个例子：



1. public，protected和private访问和继承权限/public/protected/private的区别？

* public的变量和函数在类的内部外部都可以访问。
* protected的变量和函数只能在类的内部和其派生类中访问。
* private修饰的元素只能在类内访问。
* （一）访问权限
* 派生类可以继承基类中除了构造/析构、赋值运算符重载函数之外的成员，但是这些成员的访问属性在派生过程中也是可以调整的，三种派生方式的访问权限如下表所示：注意外部访问并不是真正的外部访问，而是在通过派生类的对象对基类成员的访问。



派生类对基类成员的访问形象有如下两种：

* 内部访问：由派生类中新增的成员函数对从基类继承来的成员的访问
* 外部访问：在派生类外部，通过派生类的对象对从基类继承来的成员的访问

（二）继承权限

**public继承**

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，都保持原有的状态，而基类的私有成员任然是私有的，不能被这个派生类的子类所访问

**protected继承**

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元函数访问，基类的私有成员仍然是私有的.

**private继承**

私有继承的特点是基类的所有公有成员和保护成员都成为派生类的私有成员，并不被它的派生类的子类所访问，基类的成员只能由自己派生类访问，无法再往下继承，访问规则如下表



1. 如何用代码判断大小端存储

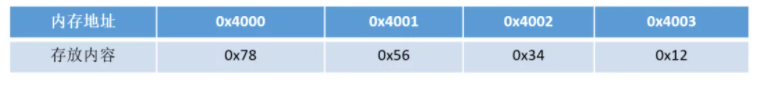
大端存储：字数据的高字节存储在低地址中

小端存储：字数据的低字节存储在低地址中

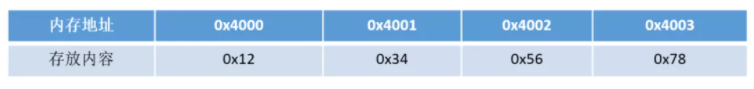
例如：32bit的数字0x12345678

**所以在Socket编程中，往往需要将操作系统所用的小端存储的IP地址转换为大端存储，这样才能进行网络传输**

小端模式中的存储方式为：



大端模式中的存储方式为：





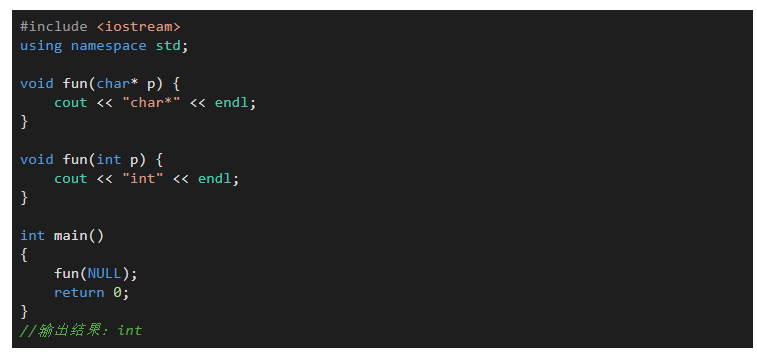
1. C++中NULL和nullptr区别

算是为了与C语言进行兼容而定义的一个问题吧

NULL来自C语言，一般由宏定义实现，而 nullptr 则是C++11的新增关键字。在C语言中，NULL被定义为(void\*)0,而在C++语言中，NULL则被定义为整数0。编译器一般对其实际定义如下：



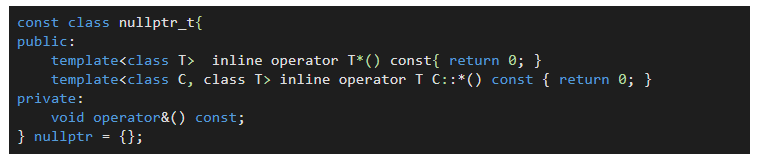
在C++中指针必须有明确的类型定义。但是将NULL定义为0带来的另一个问题是无法与整数的0区分。因为C++中允许有函数重载，所以可以试想如下函数定义情况：



**编译器在编译.cpp文件**中当前使用的作用域里的同名函数时，根据**函数形参的类型和顺序**会对**函数进行重命名**（不同的编译器在编译时对函数的重命名标准不一样）但是总的来说，他们都把文件中的**同一个函数名进行了重命名；**

那么在传入NULL参数时，会把NULL当做整数0来看，如果我们想调用参数是指针的函数，该怎么办呢?。nullptr在C++11被引入用于解决这一问题，nullptr可以明确区分整型和指针类型，能够根据环境自动转换成相应的指针类型，但不会被转换为任何整型，所以不会造成参数传递错误。

nullptr的一种实现方式如下：

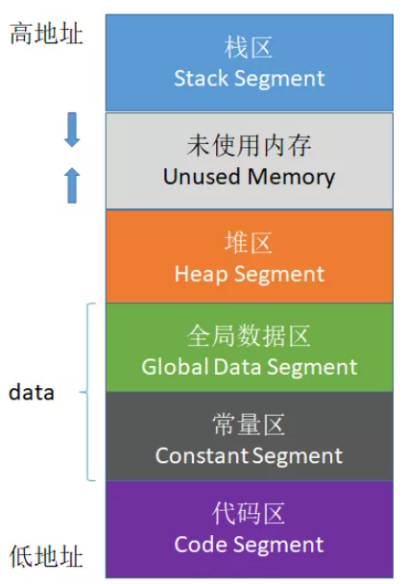


以上通过模板类和运算符重载的方式来对不同类型的指针进行实例化从而解决了(void\*)指针带来参数类型不明的问题，**另外由于nullptr是明确的指针类型，所以不会与整形变量相混淆。**但nullptr仍然存在一定问题，例如：



1. 简要说明C++的内存分区

C++中的内存分区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区和代码区。如下图所示



**栈**：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于**处理器的指令集**中，效率很高，但是分配的内存容量有限

**堆**：就是那些由malloc等分配的内存块，不过它是用free来结束自己的生命的。 new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收

**自由存储区**：就是那些由 new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个 delete。

**全局/静态存储区**：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

**常量存储区**：这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

**代码区**：存放函数体的二进制代码

1. C++的异常处理的方法

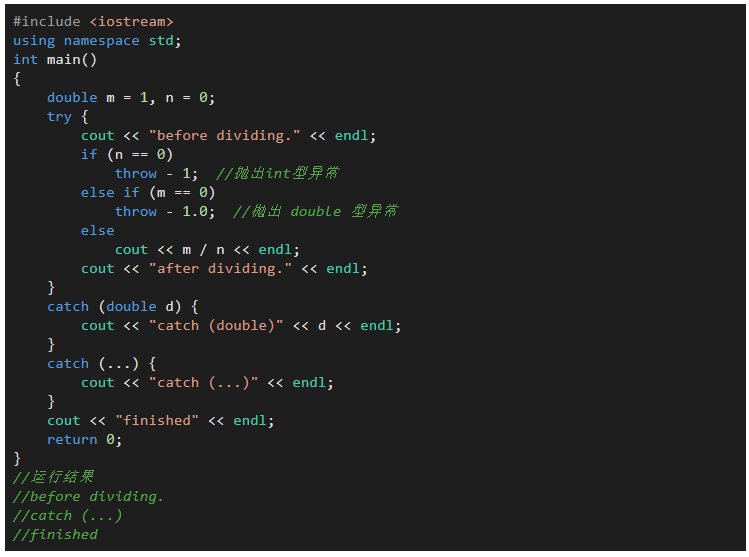
在程序执行过程中，由于程序员的疏忽或是系统资源紧张等因素都有可能导致异常，任何程序都无法保证绝对的稳定，常见的异常有：

* 数组下标越界
* 除法计算时除数为0
* 动态分配空间时空间不足
* …

如果不及时对这些异常进行处理，程序多数情况下都会崩溃。

**（1）try、throw和catch关键字**

C++中的异常处理机制主要使用**try**、**throw**和**catch**三个关键字，其在程序中的用法如下：



代码中，对两个数进行除法计算，其中除数为0。可以看到以上三个关键字，程序的执行流程是先执行try包裹的语句块，如果执行过程中没有异常发生，则不会进入任何catch包裹的语句块。如果发生异常，则使用throw进行异常抛出，再由catch进行捕获，throw可以抛出各种数据类型的信息，代码中使用的是数字，也可以自定义异常class。

catch根据throw抛出的数据类型进行精确捕获（不会出现类型转换），如果匹配不到就直接报错，可以使用catch(…)的方式捕获任何异常（不推荐）。

当然，如果catch了异常，当前函数如果不进行处理，或者已经处理了想通知上一层的调用者，可以在catch里面再throw异常。

**（2）函数的异常声明列表**

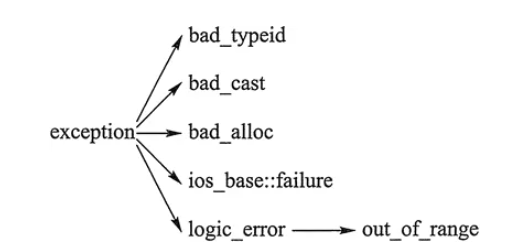
有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：



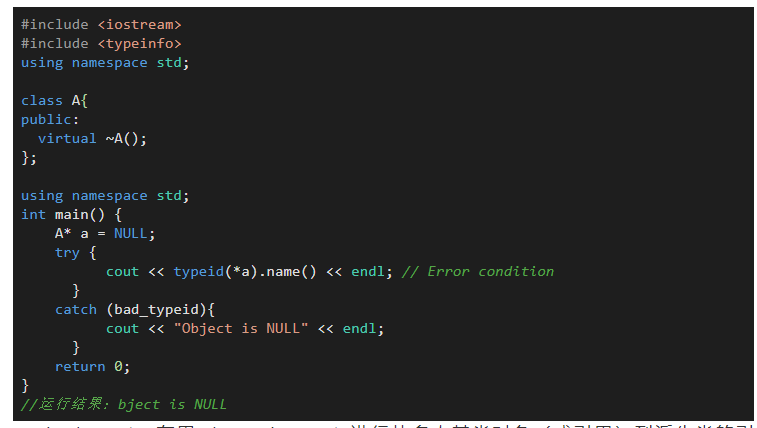
这种写法表名函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不会抛出任何异常，如果没有throw则可能抛出任何异常

**（3）C++标准异常类  exception**

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的，如下图所示



bad\_typeid：使用typeid运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会拋出此异常，例如：



* bad\_cast：在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会拋出此异常
* bad\_alloc：在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常
* out\_of\_range:用 vector 或 string的at 成员函数根据下标访问元素时，如果下标越界，则会拋出此异常

1. static的用法和作用？

1.先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可）

当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性。

2.static的第二个作用是保持变量内容的持久。（static变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围，说到底static还是用来隐藏的。

3.static的第三个作用是默认初始化为0（static变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节默认值都是0x00，某些时候这一特点可以减少程序员的工作量。

4.static的第四个作用：C++中的类成员声明static

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。

类内：

6)  static类对象必须要在类外进行初始化，static修饰的变量先于对象存在，所以static修饰的变量要在类外初始化；

7)  由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问 static修饰的类成员；

8)  static成员函数不能被virtual修饰，static成员不属于任何对象或实例，所以加上virtual没有任何实际意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual function

1. const关键字

const是C++种常用的类型修饰符，常类型是指使用类型修饰符const说明的类型，常类型的变量或对象的值是不能被更新的。

const的作用

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 作用 | 说明 | 参考代码 |
| 1 | 可以定义const常量 |  | const int max = 100 |
| 2 | 便于进行类型检查 | const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误 | void f(const int i) { .........}       //对传入的参数进行类型检查，不匹配进行提示 |
| 3 | 可以保护被修饰的东西 | 防止意外的修改，增强程序的健壮性 | void f(const int i) { i=10;//error! }       //如果在函数体内修改了i，编译器就会报错 |
| 4 | 可以很方便进行参数的调整和修改 | 同宏定义一样，可以做到不变则已，一变都变 |  |
| 5 | 为函数重载提供了一个参考 |  | class A {   void f(int i)    //一个函数   void f(int i) const {......} //上一个函数的重载 }; |
| 6 | 可以节省空间，避免不必要的内存分配 | const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝 |  |
| 7 | 提高了效率 | 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高 |  |

const的使用

* 定义常量

1. const修饰变量，一下两种定义在本质上是一样的。它的含义是const修饰的类型为type的变量value是不可变的

TYPE const Value\_Name = value;

const TYPE Value\_Name = value

1. 将const改为外部连接，作用域扩大至全局，编译时会分配内存，并且可以不进行初始化，仅仅作为声明，编译器认为在程序的其他地方进行了定义。

extend const int Value\_Name = value;

* 指针使用const

1. 指针本身是常量不可变

char\* const pContent;

1. 指针所指向的内容是常量不可变

const char\* pContent;

1. 两者都不可变

const char\* const pContent

还有其中的区分方法，沿着\*号划一条线：

如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向变量

如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

* 函数中使用const

const修饰函数参数

* 1. 传递过来的参数在函数内不可以改变（无意义，因为var本身就是形参）

void function(const int var);

* 1. 参数指针所指内容为常量不可变

void function(const char\* var);

* 1. 参数指针本身是常量不可变

void function(char\* const var);

* 1. 参数为引用，为了增加效率同时防止修改。修饰引用参数时：

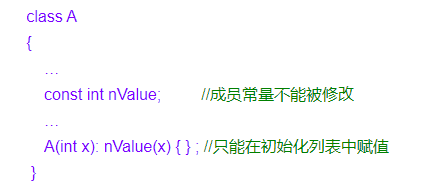
void function(const class& var); //引用参数在函数内不可以改变

void function(const TYPE& var); //引用参数在函数内为常量不可变

这样的一个const引用传递和最普通的函数按值传递的效果是一模一样的,他禁止对引用的对象的一切修改,唯一不同的是按值传递会先建立一个类对象的副本, 然后传递过去,而它直接传递地址,所以这种传递比按值传递更有效.另外只有引用的const传递可以传递一个临时对象,因为临时对象都是const属性, 且是不可见的,他短时间存在一个局部域中,所以不能使用指针,只有引用的const传递能够捕捉到这个家伙.

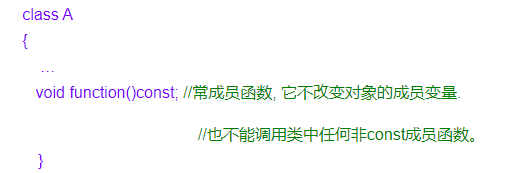
* 类相关使用const
  + - 1. const修饰成员变量

const修饰类的成员函数，表示成员常量，不能被修改，同时它只能在初始化列表种赋值



* + - 1. const修饰成员函数

const修饰类的成员函数，**则该成员函数不能修改类中任何非const成员变量**。一般写在函数的最后来修饰

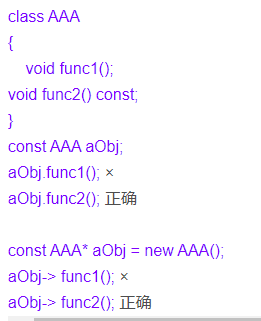


对于const类对象/指针/引用，只能调用类的const成员函数，因此const修饰成员函数的最重要作用就是限制对于const对象的使用。

* + 1. const成员函数不被允许修改它所在对象的任何一个数据成员
    2. const成员函数能够访问对象的const成员，而其他成员函数不可以
       1. const修饰类对象/对象指针/对象引用

const修饰类对象表示该对象时常量对象，其中的二年和成员不能被修改，对于对象指针和对象引用也一样。

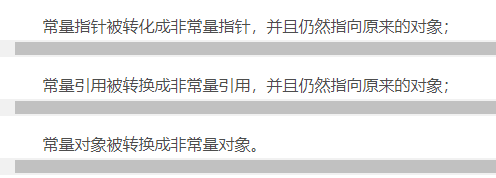
const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。



将const类型转化为非const类型的方法

采用：const\_cast<type\_id> (expression)进行转换

该运算符用来修改类型的const或volatile属性。除了const或volatile修饰之外，type和expression类型是一样的



使用const的一些建议

* 大胆使用const，这将给你无尽的益处，但前提是你必须搞清楚
* 要避免最一般的赋值操作错误，如将const变量赋值，具体可见思考题；
* 在参数中使用const应该使用引用或指针，而不是一般的对象实例，原因同上；
* const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；
* 不要轻易的将函数的返回值类型定为const;
* 除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;
* 任何不会修改数据成员的函数都应该声明为const 类型。

补充重要说明：

## 在C++中只有被声明为const的成员函数才能被一个const类对象调用。

## const成员函数可以被对应的具有相同形参列表的非const成员函数重载

非const对象既可以调用const成员函数，也可以调用非const成员函数。

const对象只能调用const成员函数，不能调用非const成员函数。因为非const成员函数的this指针类型为非const，而const对象的指针类型为const，不可以将const指针作为参数传入非const成员函数。

C++默认指向const对象的指针必须也具有const属性。

1. static关键字

**在程序中使用static**

**变量**

* 1. **局部变量**

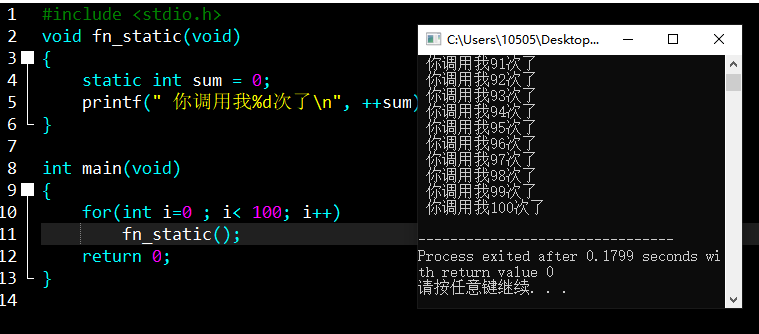
**普通局部变量是再熟悉不过的变量了，在任何一个函数内部的变量（不如static修饰符）都属于这个范畴。编译器一般不对普通局部变量进行初始化，也就是说它的初值是不确定的，除非对其显示赋值。**

**即普通局部变量存储于进程栈空间，使用完毕后立即释放。**

**静态局部变量使用static修饰符定义，即时在声明时未赋初值，编译器也会把它初始化为0.且**静态局部变量存储于进程的全局数据区**，即时函数返回，它的值也会保持不变**

**变量在全局数据区分配内存空间；编译器自动对其初始化，其作用域为局部作用域，当定义它的函数结束时，其作用域随之结束。**

**作用域结束、但数据未被回收，因为放在全局数据区。**



**可见，静态局部变量的效果跟全局变量有一拼，但是位于函数体内部，就极有利于程序的模块化了。**

* 1. **全局变量**

**全局变量定义在函数体外部，在全局数据区分配存储空间，且编译器会自动对其初始化。**

**普通全局变量对整个工程可见，其他文件可以使用extern外部声明后直接使用。也就是说其他文件不能再定义一个与其相同名字的变量了（否则编译器会认为它们是同一个变量）**

**静态全局变量仅对当前文件可见，其他文件不可访问，其他文件可以定义与其同名的变量，两者互不影响。**

在定义不需要与其他文件共享的全局变量时，加上static关键字能够有效地降低程序模块之间的耦合，避免不同文件同名变量的冲突，且不会误使用。

**函数：**

**函数的使用方式与全局变量类似，再函数的返回类型前加上static，就是静态函数。其特性如下：**

**静态函数只能声明它的文件中可见，其他文件不能使用该函数**

**不同的文件可以使用相同名字的静态函数，互不影响**

**非静态函数可以在另一个文件中直接引用，甚至不必extern声明**

**静态数据成员：**

**在类内数据成员的声明前加上static关键字，该数据成员就是类内的静态数据成员。其特点如下：**

**1. 静态数据成员存储在全局数据区，静态数据成员在定义时分配存储空间，所以不能在类声明中定义**

**2.静态数据成员是类的成员，无论定义了多少个类的对象，静态数据成员的拷贝只有一个，且对该类的所有对象可见。也就是说任一对象都可以对静态数据成员进行操作。而对于非静态数据成员，每个对象都有自己的一份拷贝。**

**3.由于上面的原因，静态数据成员不属于任何对象，在没有类的实例时其作用域就可见，在没有任何对象时，就可以进行操作**

**4.和普通数据成员一样，静态数据成员也遵从public, protected, private访问规则**

**5.静态数据成员的初始化格式：<数据类型><类名>::<静态数据成员名>=<值>**

* **类的静态数据成员有两种访问方式：<类对象名>.<静态数据成员名> 或 <类类型名>::<静态数据成员名>**

**6.同全局变量相比，使用静态数据成员有两个优势：**

**（1）静态数据成员**没有进入程序的全局名字空间**，因此不存在与程序的其他全局名字冲突的可能**

**（2）可以实现信息隐藏。静态数据成员可以是private成员，而全局变量不能**

**静态成员函数：**

**与静态数据成员类似，静态成员函数属于整个类，而不是某一个对象，其特性如下**

**（1）静态成员函数没有this指针，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数**

**（2）出现在类体外的函数不能指定关键字static**

**（3）非静态函数可以任意的访问静态成员函数和静态数据成员（这点与const相同）**

**总结：**

**static是一个很有用的关键字，使用得当可以使程序锦上添花。当然，有的公司编码规范明确规定只用于本文件的函数要全部使用static关键字声明，这是一个良好的编码风格。**

**无论如何，要在实际编码时注意自己的编码习惯，尽量体现出语言本身的优雅和编码者的编码素质。**

1. 继承条件下对象构造析构顺序是否可以改变 即先构造子类再构造父类

不可以

1. C++面向对象的三个基本特征：封装、继承、多态

封装可以隐藏实现细节，使得代码模块化；

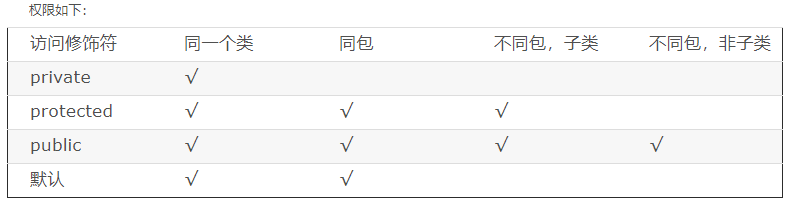
继承可以扩展已存在代码模块；它们的目的都是为了——代码重用

而多态则是为了实现另一个目的——接口重用

封装的意义：

保护或者防止代码被无意破坏。

* 1. 保护数据成员
  2. 方法的细节对用户是隐藏的，只要接口不变，内容的修改不会影响到外部调用者
  3. 对象含有完整属性与之对应的方法时称为封装
  4. 从对象外面不能直接访问对象的属性，只能通过该属性对应的方法访问
  5. 对象的方法可以接收对象外面的消息



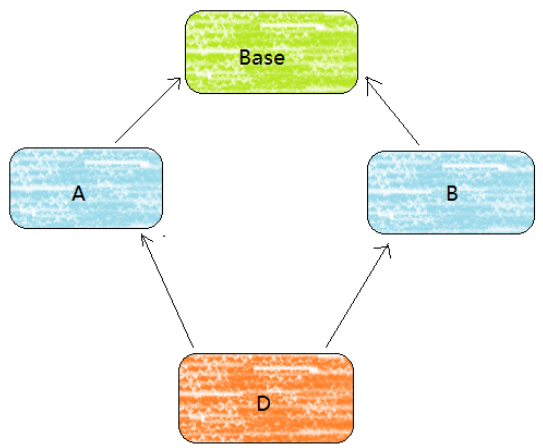
继承：

主要实现重用代码，节省开发时间：

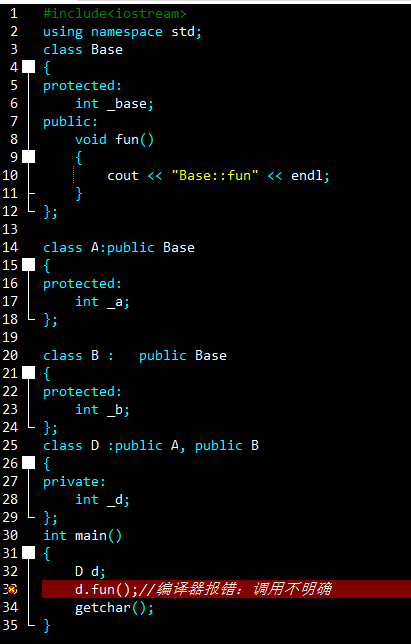


保护成员扩大的访问范围表现在：基类的保护成员可以在派生类的成员函数中被访问。

1. 菱形继承的相关问题

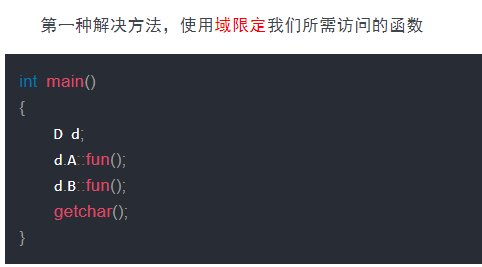


 如上图，菱形继承即多个类继承了同一个公共基类，而这些派生类又同时被一个类继承。这么做会引发什么问题呢，让我们来看一段代码吧！



 我们可以看见D的对象模型里面保存了两份Base,当我们想要调用我们从Base里继承的fun时就会出现调用不明确问题，并且会造成数据冗余的问题，明明可以只要一份就好，而我们却保存了两份。

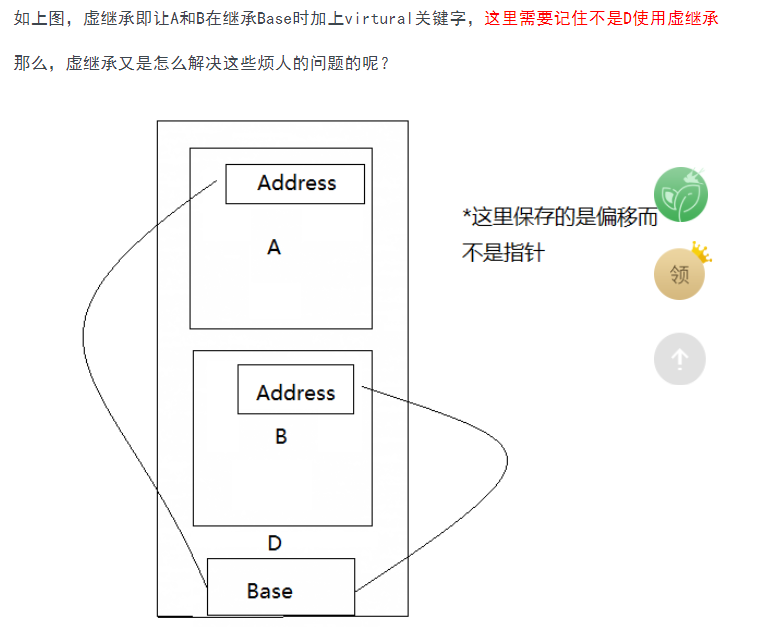
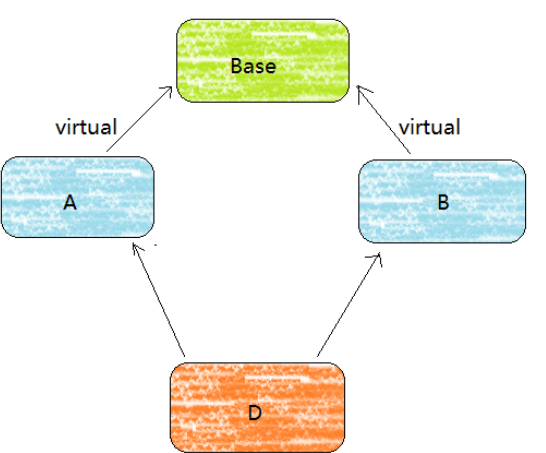
解决方法一：域限定



这样做法没有问题但是，这样做非常的不方便，并且当程序十分大的时候会造成我们思维混乱

    于是，C++给了我们一个别的解决方案——虚继承

解决方法二：虚继承



我们可看见在A和B中不再保存Base中的内容，**保存了一份偏移地址**，然后将Base的数据保存在一个公共位置处这样保证了数据冗余性的降低同时，我们也能直接的使用d.fun（）来调用Base里的fun函数。

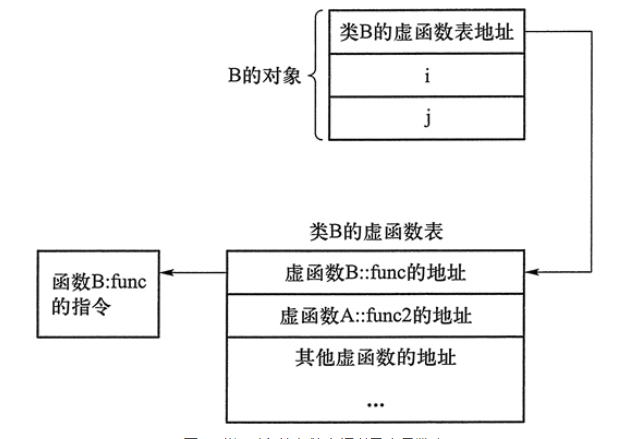
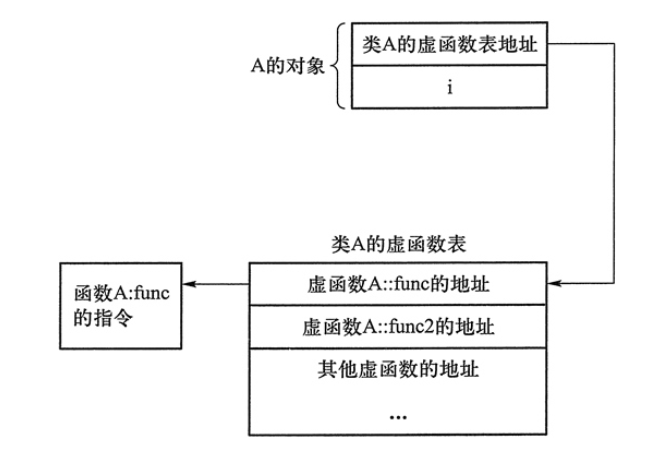
菱形继承的虚函数表：

菱形继承会有两份虚函数表指针。调用父类虚函数会有二义性的存在。域限定可以解决。

1. 虚函数表（多态实现的原理）

有了虚函数以后，对象所占的存储空间比没有虚函数多了4个字节。实际上，任何有虚函数的类及其派生类的对象都包含多出来的四个字节，这四个字节就是位于对象存储空间的最前端，实现动态多态的关键，其中存储的就是虚函数表的地址。

每一个有虚函数的类（或有虚函数的类的派生类）都有一个虚函数表，该类的任何对象都放着虚函数表的指针（可以认为是由编译器自动添加到构造函数中的指令完成的）

虚函数表是编译器生成的，程序运行时被载入内存。一个类的虚函数表中列出该类的全部虚函数地址。例如。

多态的函数调用语句被编译成根据基类指针所指向的（或基类引用所引用的）对象中存放的虚函数表的地址，在虚函数表中查找虚函数地址，并调用虚函数的一系列命令。

程序所占有的分区5种：1.代码区 2.堆区 3.栈区 4.全局数据区（静态区） 5.文字常量区

虚函数表存储在虚函数表存放在全局数据区

1.不是函数 2.不是代码段3.不是局部变量 4. 虚函数的个数在编译时期可以确定,即虚函数表的大小可以确定,即大小是在编译时期确定的

1. 两个进程中，申请的指针，可能指向同一片内存吗

可能，指向指针离散分布。

1. 代码中的变量是逻辑地址还是物理地址

逻辑地址

首先

1.不然换个电脑就无法跑这个程序了

2.逻辑地址是操作系统可执行的物理地址空间映射出来的逻辑地址

3.一般地逻辑地址和物理地址没有直接的关系，这个转换由操作系统完成并维护，与你的程序无关。

1. 同一个进程下的两个线程a和b，在线程a中有一个static变量，线程b能访问得到吗？在线程a中new一个对象，线程b能访问的到吗？
2. 函数指针和指针函数（函数指针的优势编译的过程怎么链接的 ）

指针函数：返回指针类型的函数

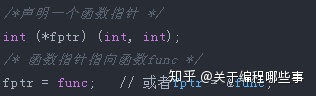
函数指针：其本质是一个指针变量，该指针指向这个函数。总结来说，函数指针就是指向函数的指针。

指针函数本质是一个函数，其返回值为指针。  
 函数指针本质是一个指针，其指向一个函数。

指针函数的本质是一个函数，其返回值是一个指针。示例如下：

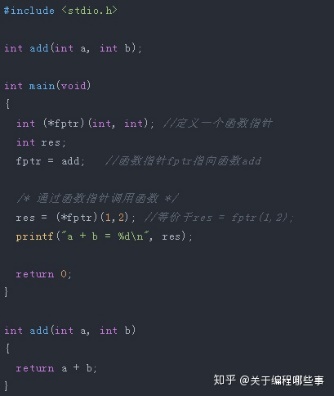
* int \*pfun(int, int);

函数指针其本质是一个指针变量，该指针变量指向一个函数。C程序在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。函数指针示例：



func 是一个函数名，那么func与&func都表示的是函数的入口地址。同样的，在函数的调用中可以使用：方式一：func()，也可以使用方式二：(\*fun)()。这两种调用方式是等价的，只是我们平时大多都习惯用方式一的调用方法。至于为什么func与&func的含义相同：

对于函数func来说，函数的名称就是函数代码区的常量，对它取地址(&func)可以得到函数代码区的地址，同时，fun本身也可以视为函数代码区的地址。因此，函数名称和对其取地址其含义是相同的。函数指针示例程序如下：



函数指针广泛应用于嵌入式软件开发中，其常用的两个用途：调用函数和做函数的参数

1. lambda匿名函数原理（待完善）
2. sizeof和strlen的区别，sizeof计算结构体大小（结构体对齐！）（自己尝试一下）一个空类的大小，一个具有虚函数类的大小。

一个空类的大小为1个字节不允许一个对象（当然包括类对象）的大小为0，不同的对象不能具有相同的地址

首先，类大小的计算遵循结构体的对齐原则

类的大小与普通数据成员有关，与成员函数和静态成员无关。即普通成员函数，静态成员函数，静态数据成员，静态常量数据成员均对类的大小无影响

虚函数对类的大小有影响，是因为虚函数表指针带来的影响

虚继承对类的大小有影响，是因为虚基表指针带来的影响

空类的大小是一个特殊情况,空类的大小为1，注意内存对齐，可省字节

 含虚函数的的单一继承与多继承

https://blog.csdn.net/longjialin93528/article/details/80160467

1. strcpy的缺点，与之相关的memcpy缺点（待完善）
2. 32位系统malloc最大申请内存多少(malloc最大申请内存)

地址空间限制是有的，

但是malloc通常情况下申请到的空间达不到地址空间上限。

内存碎片会影响到你“一次”申请到的最大内存空间。比如你有10M空间，申请两次2M，一次1M，一次5M没有问题。

但如果你申请两次2M，一次4M，一次1M，释放4M，那么剩下的空间虽然够5M，但是由于已经不是连续的内存区域，malloc也会失败。系统也会限制你的程序使用malloc申请到的最大内存。

Windows下32位程序如果单纯看地址空间能有4G左右的内存可用，不过实际上系统会把其中2G的地址留给内核使用，所以你的程序最大能用2G的内存。

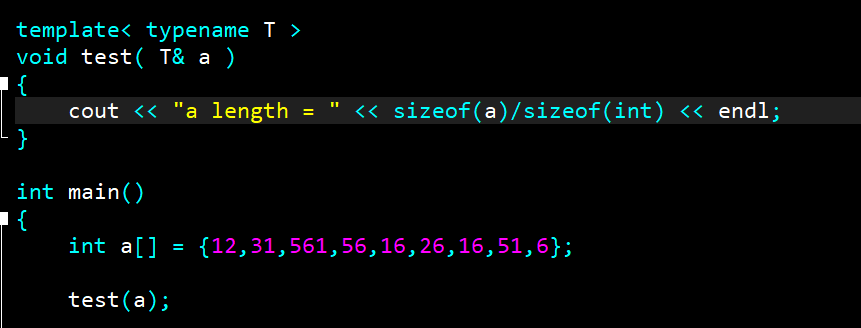
除去其他开销，你能用malloc申请到的内存只有1.9G左右。

影响因素：

操作系统  
硬件  
当前内存的使用状况。

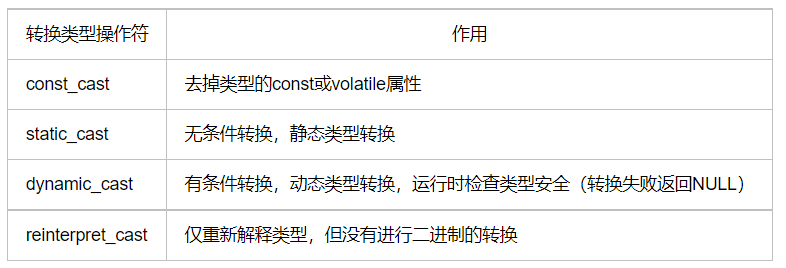
1. 传输参数是数组，不是vector，如何判断数组的大小？

* 循环到NULL指针
* 使用template<typename T>定义函数，定义模板就可以把数组引入到自定函数当中。



1. 类型转换，有哪些，用过哪些？

<https://www.cnblogs.com/evenleee/p/10382335.html>

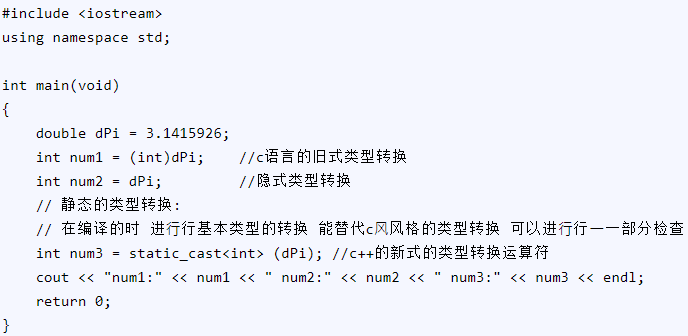


* + - 1. const\_cast，去掉类型的const类型或volatile类型

const\_cast<目标类型>（标识符）:目标类型只能是**指针或者引用**

* + - 1. **static\_cast，无条件转换，静态类型转换**

static\_cast<目标类型>（标识符）所谓静态，即在编译器内即可决定其类型的转换，用的也是最多的一种



* + - 1. dynamic\_cast，有条件转换，动态类型转换，运行时检查类型安全（失败返回NULL）

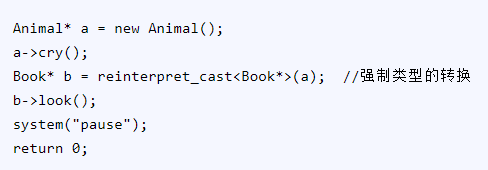
子类与父类之间的多态类型准换

用于多态中父子类之间的多态转换

dynamic\_cast<目标类型>（标识符）

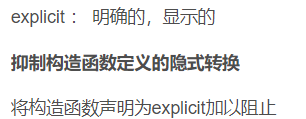
* + - 1. reinterpret\_cast，仅重新解释类型，但没有进行二进制的转换

reinterpret\_cast<目标类型>（标识符）数据的二进制重新解释，但是不改变其值。



1. 4个智能指针，应用场景

C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。



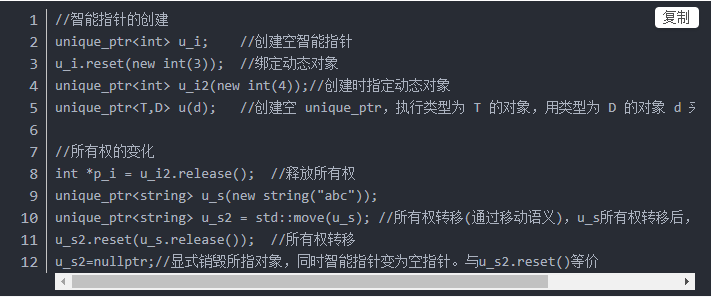
关键字explicit只对一个实参的构造函数有效（因此在智能指针当中，不需要自动将指针转换为智能指针对象）

unique\_ptr（替代auto\_ptr）:

持有对对象的独有权——两个unique\_ptr 不能指向一个对象，即 unique\_ptr 不共享它所管理的对象。它无法复制到其他 unique\_ptr，无法通过值传递到函数，也无法用于需要副本的任何标准模板库 （STL）算法。只能移动 unique\_ptr，即对资源管理权限可以实现转移。

这意味着，内存资源所有权可以转移到另一个 unique\_ptr，并且原始 unique\_ptr 不再拥有此资源。

unique\_ptr 与原始指针一样有效，并可用于 STL 容器。将 unique\_ptr 实例添加到 STL 容器运行效率很高，因为通过 unique\_ptr 的移动构造函数，不再需要进行复制操作。

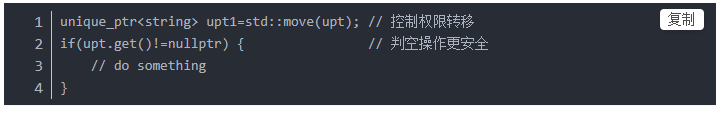


auto\_ptr（C++11已经摒弃）:

auto\_ptr采取所有权模型，但是在运行的时候才会程序奔溃。即转换后，原先指针也已经是空指针了，程序会在运行的时候崩溃。而unique\_ptr不会在运行阶段崩溃，而是在编译器因下述代码就会出现错误。避免因潜在内存问题导致程序崩溃。

因为 auto\_ptr 有拷贝语义，拷贝后原对象变得无效，再次访问原对象时会导致程序崩溃；unique\_ptr 则**禁止了拷贝语义，但提供了移动语义**，即可以使用 std::move()

进行控制权限的这里要注意，在使用 std::move 将 unique\_ptr 的控制权限转移后，不能够再通过 unique\_ptr 来访问和控制资源了，否则同样会出现程序崩溃。我们可以在使用 unique\_ptr 访问资源前，使用成员函数 get() 进行判空操作。转移



shared\_ptr:（大小是普通指针的两倍）

shared\_ptr 是一个标准的共享所有权的智能指针，允许多个指针指向同一个对象. 利用引用计数的方式实现了对所管理的对象的所有权的分享，即允许多个 shared\_ptr 共同管理同一个对象。像 shared\_ptr 这种智能指针.

weak\_ptr:

解决shared\_ptr的死锁

智能指针的缺点：

* + - 1. shared\_ptr循环引用容易死锁
      2. 基于引用计数的一些性能损耗（多线程会有问题）

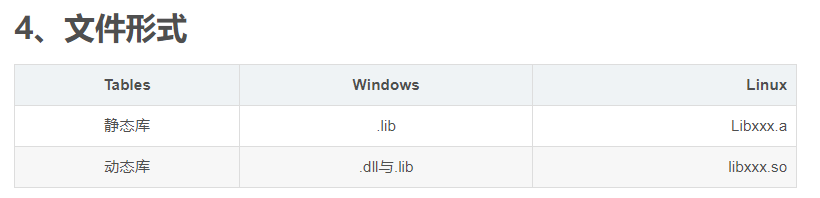
1. C++深克隆（要实现深克隆要实现C++成员函数里的什么函数）memcpy
2. 内存对齐了解吗？给一个结构体，包括char、int指针，sizeof有多大？为什么？如果想紧凑对齐应该怎么做？32位LINUX是怎么内存对齐？64位呢？
3. C++静态库和动态库

静态库和动态库从字面意思来看，区别就是静态和动态。而这里的静态和动态指的是库的链接阶段。可以看如下的编译过程。

静态库：在链接阶段库将会与目标汇编后的目标文件.o一起打包生成可执行文件。成为可执行文件的一部分，后续此库就可以消失了。也就是说在编译的最后一步（链接阶段），如果程序需要使用静态库，在这一步都会一起打包到可执行文件中。

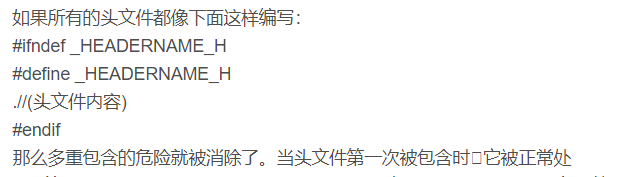
动态库：而动态库在编译阶段都不会有什么动作，只有在程序运行时才被加载，也就是动态库的链接是发生在程序运行时期的，它和可执行文件是分开的，只是可执行文件在运行的某个时期调用了它。



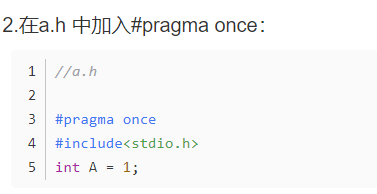


1. C++防止头文件被重复引入

条件编译



program once



1. #include<>和#include””的区别

引用的头文件不同：

#include <>引用的是编译器的类库路径里面的头文件

#include “”引用的是程序目录的相对路径下的头文件

用法不同：

#include <>用来包含标准头文件

#include “”用来包含非标准头文件

调用文件的 顺序不同：

#include<>编译程序会先到标准函数库中调用文件

#include””编译程序会先到当前目录中调用文件

预处理程序的指示不同：

#include<>指示预处理程序到预处理的缺省路径下寻找文件

#include””指示预处理程序先到当前目录下寻找文件

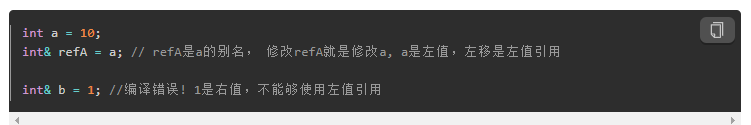
“”找得到标准头文件、而<>找不到程序下的头文件

1. Resize和reserve的区别



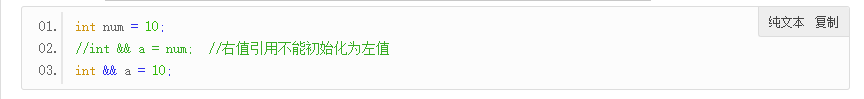
1. 左值右值，右值引用、移动语义和完美转发。

左值引用：



右值引用：

跟左值引用一样，都需要进行初始化，但是左值引用对象是变量，而右值引用和声明左值引用一样，右值引用也必须立即进行初始化操作，且只能使用右值进行初始化



c++中引入了右值引用和移动语义，可以避免无谓的复制，提高程序性能。

核心：

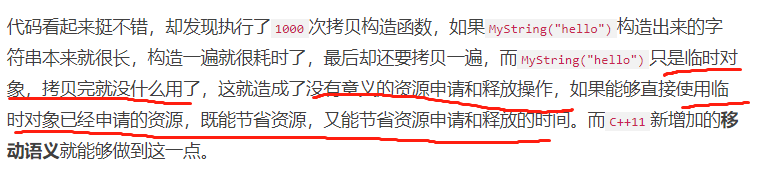
**C++中所有的值都必然属于左值、右值二者之一。左值是指表达式结束后依然存在的持久化对象，右值是指表达式结束时就不再存在的临时对象。所有的具名变量或者对象都是左值，而右值不具名。很难得到左值和右值的真正定义，但是有一个可以区分左值和右值的便捷方法：看能不能对表达式取地址，如果能，则为左值，否则为右值。**

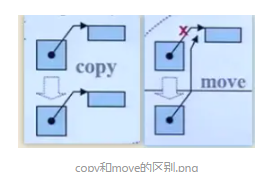
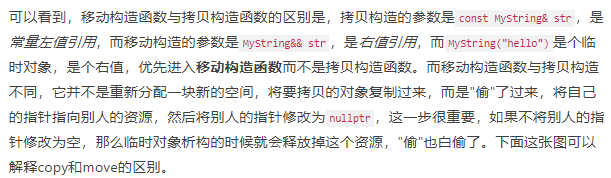
右值引用的特点：

* 特点1：通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样长，只要该变量还活着，该右值临时量将会一直存活下去
* 特点2：右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值
* 特点3：T&& t在发生自动类型推断的时候，它是左值还是右值取决于它的初始化。

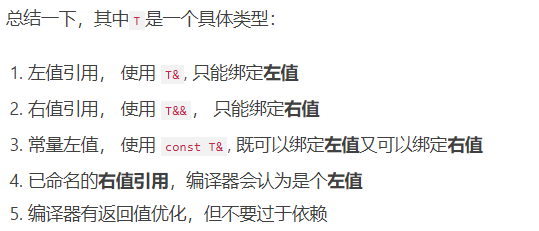
右值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要std::move()将左值强制转换为右值。

移动语义：要实现移动语义就必须增加两个函数：移动构造函数和移动赋值构造函数。









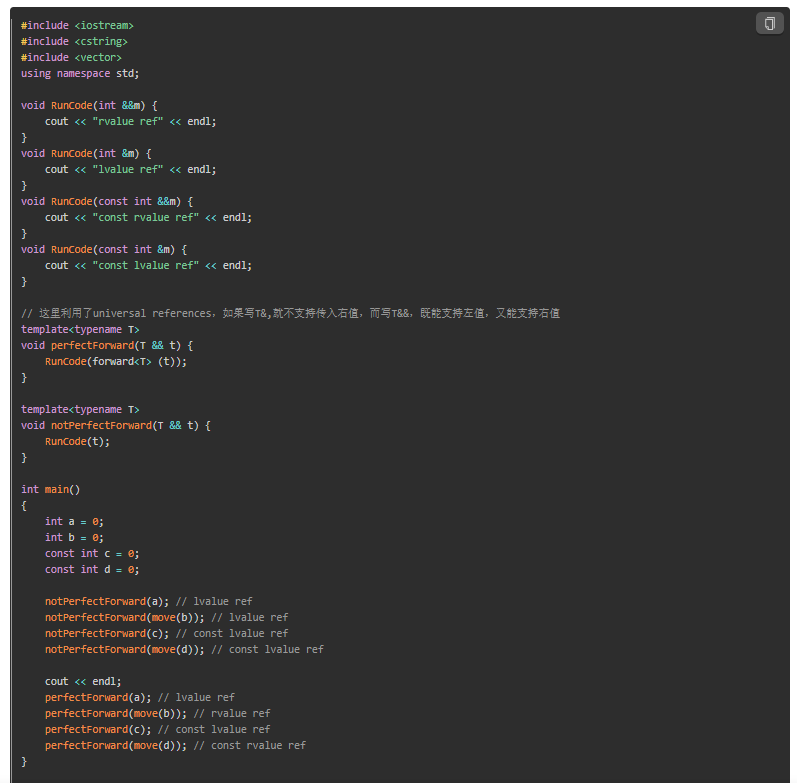
Universal references(通用引用)



只有当**发生自动类型推断**时（如函数模板的类型自动推导，或auto关键字），&&才是一个universal references。

完美转发：





使用universal references通用引用类型 + std::forward模板函数共同实现完美转发。

所谓转发，就是通过一个函数将参数继续转交给另一个函数进行处理，原参数可能是右值，可能是左值，如果还能继续保持参数的原有特征，那么它就是完美的。

1. 空类大小为1，默认实现了什么？

如果空类是0产生两个问题

* + - 1. sizeof T 为0会导致非法除0操作
      2. 不能通过指针区分不同数组对象

默认实现，**默认构造函数、默认拷贝构造函数、析构函数、赋值运算符**

1. C++中类如果没有自定义构造函数，系统会默认为类提供什么构造函数

共两个：默认构造函数和默认拷贝构造函数

1. 析构函数为什么不能抛出异常

more effective c++提出两点理由：

1. 如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。

2. 通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。

1. new失败了会返回什么？

new失败返回bad\_alloc并抛出异常、有些编译器对C++标准支持不是很好，new失败就不会抛出异常，而返回0。底层封装malloc，malloc失败返回0（不够内存了）

1. volatile关键字的作用

volatile提醒编译器它后面所定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

声明时语法： volatile int vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

volatile的本意是“易变的”,由于**访问寄存器的速度要快过RAM**，所以编译器一般都会作减少存取外部RAM的优化。



volatile的使用场景：

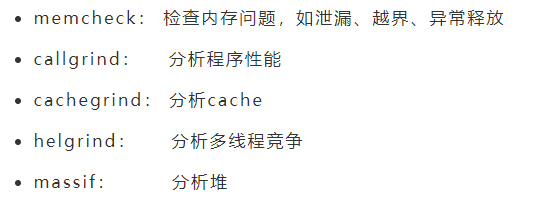
* + - 1. 中断服务程序中修改的供其它程序检测的变量需要加volatile；
      2. 多任务环境下各任务间共享的标志应该加volatile；
      3. 存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；

volatile不能保证线程安全。因为多线程对数据的修改主要是先取值、修改、赋值，而volatile只能保证在各个线程串行运行访问同一个变量的时候才会线程安全。比如：多核计算机中i == 0,两个线程同时操作i++，最终结果也可能出现i == 1的错误结果。

1. 用valgrind定位内存泄漏

valgrind是一款程序调试和分析的工具，其最重要的功能是内存检查。除了检查内存泄露外，还能检查内存越界、内存异常释放等问题。

具体来说，就是可以通过valgrind的toolname参数，来定制使用valgrind功能。下面，一起来看下toolname参数的部分选项。



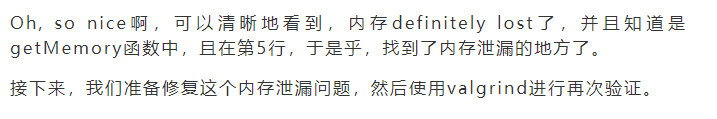
接下来，我们介绍使用valgrind的toolname参数中的memcheck, 并用实际例子来看看如何定位内存泄漏。

先写一段内存泄漏的代码



valgrind *--tool=memcheck --leak-check=yes --show-reachable=yes ./a.out*



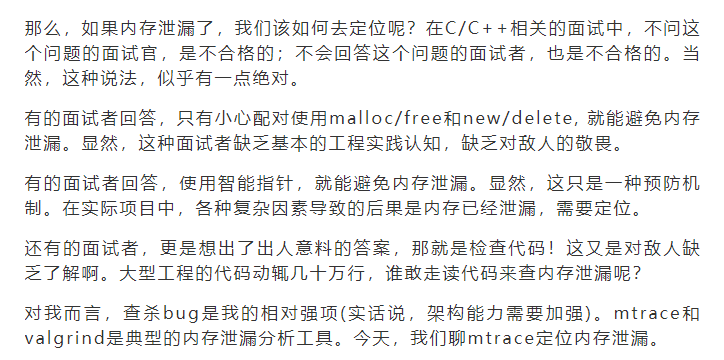


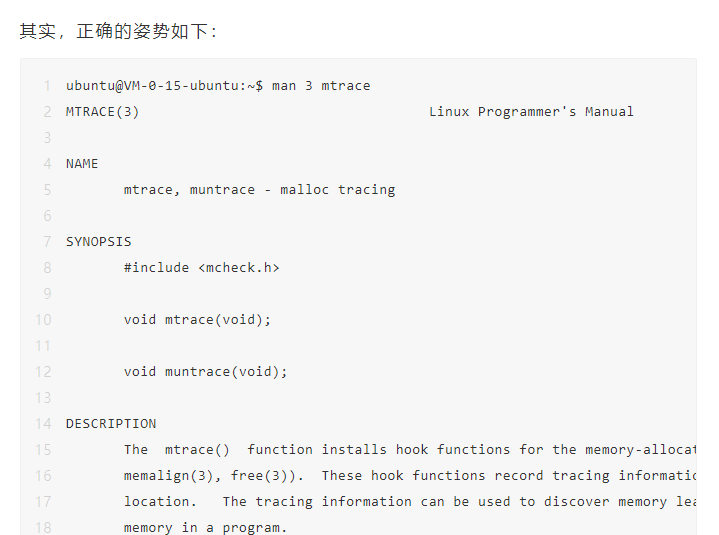
修复后，验证：





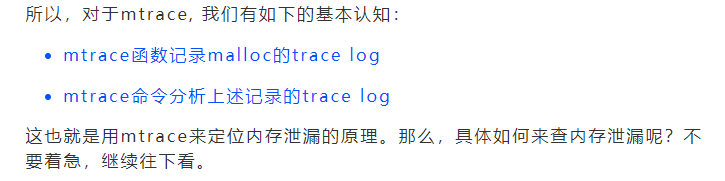
1. **mtrace定位内存泄漏。**





需要mcheck.h头文件支持，且需要函数实现。日志log记录加a.out二进制











1. 内存泄漏解决办法

说了三点，并比较详细的解释了补：（1）智能指针 （2）string代替char\* （3）RAII

1. C++中的静态绑定和动态绑定（多态）

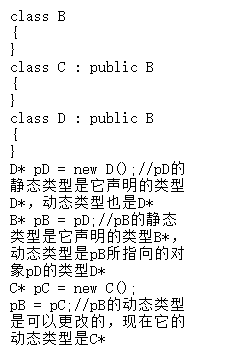
为了支持C++的多态性才用了动态绑定和静态绑定。理解他们的区别有助于更好的理解多态性，以及在编程的过程中避免犯错误。

需要理解四个名词：

1、对象的静态类型：对象在声明时采用的类型。是在编译期确定的。

2、对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。是指“**目前所指对象的类型**”

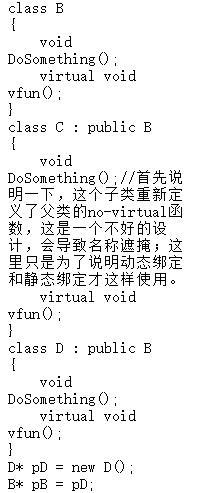
关于对象的静态类型和动态类型，看一个示例：



3、静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。通过对象调用

4、动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。通过地址实现

**virtual函数是动态绑定，non-virtual函数是静态绑定，缺省参数值也是静态绑定**。



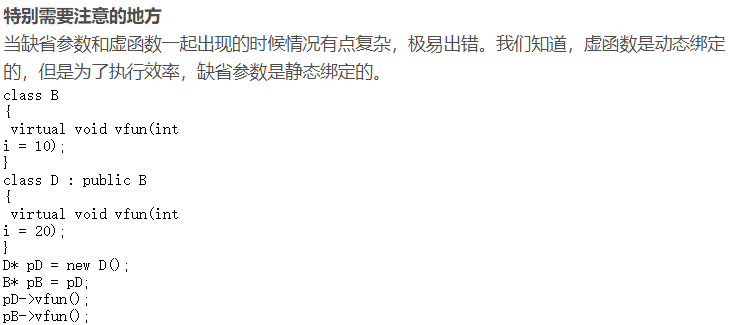
让我们看一下，pD->DoSomething()和pB->DoSomething()调用的是同一个函数吗？

不是的，虽然pD和pB都指向同一个对象。因为函数DoSomething是一个no-virtual函数，它是静态绑定的，也就是编译器会在编译期根据对象的静态类型来选择函数。pD的静态类型是D\*，那么编译器在处理pD->DoSomething()的时候会将它指向D::DoSomething()。同理，pB的静态类型是B\*，那pB->DoSomething()调用的就是B::DoSomething()。

因为vfun是一个虚函数，它动态绑定的，也就是说它绑定的是对象的动态类型，pB和pD虽然静态类型不同，但是他们同时指向一个对象，他们的动态类型是相同的，都是D\*，所以，他们的调用的是同一个函数：D::vfun()。

永远记住：为了执行效率

“绝不重新定义继承而来的缺省参数（Never redefine function’s inherited default parameters value.）”



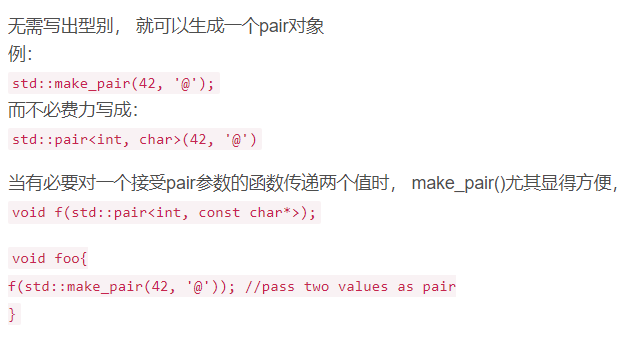
1. std::pair

C++标准程序库中凡是“必须返回两个值”的函数，也都会利用pair对象

class pair可以将两个值视为一个单元。容器类别map和multimap就是使用pairs来管理其健值/实值(key/value)的成对元素。

pair被定义为**struct**,因此可直接存取pair中的个别值.。两个pairs互相比较时， 第一个元素正具有较高的优先级.。

make\_pair的用法



比如queue<pair<int, int>>，获取队头元素auto [x, y] = front()

pair的应用

pair是将2个数据组合成一个数据，当需要这样的需求时就可以使用pair，如stl中的map就是将key和value放在一起来保存。另一个应用是，当一个函数需要返回2个数据的时候，可以选择pair。 pair的实现是一个结构体，主要的两个成员变量是first second 因为是使用struct不是class，所以可以直接使用pair的成员变量。

1. 结构体设计思想

https://mp.weixin.qq.com/s/FbaGdRgUFmfXYY52NxbbbA

1. 泛型编程（C++如何实现泛型编程、什么是泛型编程、优缺点）

所谓泛型编程就是独立于任何特定类型的方式编写代码，使用泛型程序时，需要提供具体程序实例所操作的类型或者值。我们经常用到的STL容器、迭代器和算法都是泛型编程的例子。

1. 模板是C++支持参数化多态的工具，使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型；
2. 通常有两种形式：函数模板和类模板；
3. 函数模板针对仅参数类型不同的函数；
4. 类模板针对仅数据成员和成员函数类型不同的类；
5. 使用模板的目的就是能够让程序员编写与类型无关的代码。比如编写了一个交换两个整型int 类型的swap函数，这个函数就只能实现int 型，对double，字符这些类型无法实现，要实现这些类型的交换就要重新编写另一个swap函数。使用模板的目的就是要让这程序的实现与类型无关，比如一个swap模板函数，即可以实现int 型，又可以实现double型的交换。

C++ 的泛型编程是基于模板实现的，而 C++ 的模板采用的是**代码膨胀技术**。例如 std::list 容器，如果你将 int 类型的数据存进去，C++ 编译器就为你生成一个专门用来存 int 类型数据的列表数据结构。也就是说，你向 std::list 容器中存放什么类型，C++ 编译器就为你生成相应的列表数据结构。理论上，数据的类型是无限的，因此 C++ 要生成的列表数据结构也是无限的。如果你的程序中有大量的数据类型要存到 std::list 容器，那么代码就会高度膨胀，这种膨胀是 C++ 编译器在目标文件连接阶段无法优化的。

建议：将与参数无关的代码抽离 templates

又带来了以下问题：

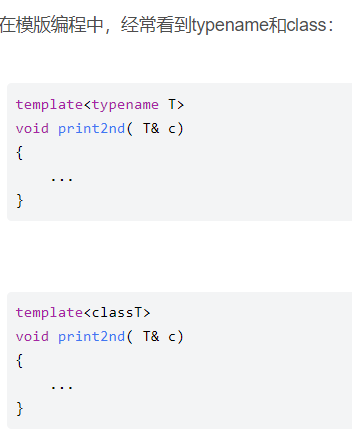
1.源代码膨胀了，因为程序猿要做『将与参数无关的代码从模板中抽离』这件事。有人做过试验，即使是一个不太大的 List 实现，将代码从模板中抽离后，导致源代码膨胀了 20%……其实开发效率也自然降低了很多。

2.编译时间被拖长了，因为编译器在代码编译阶段要对模板代码进行**『惰性计算』**，要产生模板的实例代码，在目标文件链接阶段还要消除各个目标文件中重复的模板代码。

3.目标文件膨胀了。有人说他用 boost::spirit 实现了一个很小的语法解析器，开了 GCC 的最大化优化选项，目标文件也要几十 MB，而一个 Lua 或 Python 解释器还不到 1 MB，Haskell 的解释器 ghc 刚 1 MB 多一点……

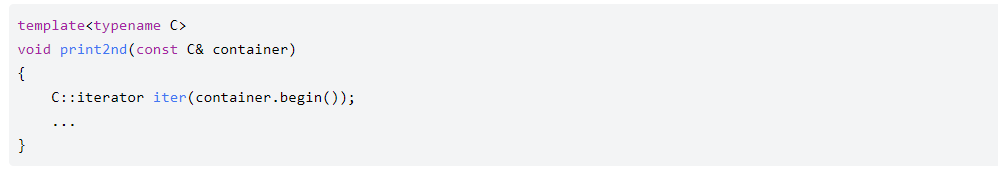
4.模板代码中如果存在错误，编译器产生的错误信息也膨胀了，特别是模板类的嵌套嵌套再嵌套，或者模板实例非常多的时候，编译出错信息无法卒读，甚至有人说编译出错信息甚至超出了他用的文本编辑器的缓存空间大小。

1. typename 和 class的区别

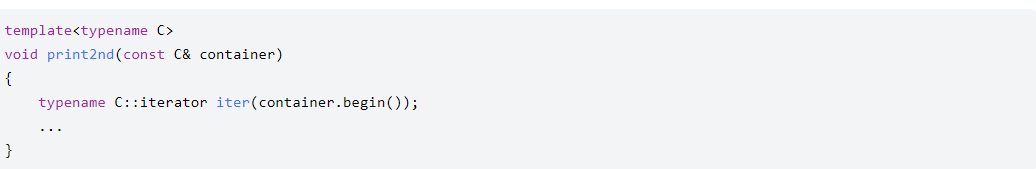


上面这种情况，typename和class是完全一样的。

那么，什么时候必须使用typename而不是class？



上面这个模版函数，我们的目的是处理容器，因而在里面使用容器的迭代器C::iterator iter，但是编译器并不知道C是什么东西，因此就不知道C::iterator 是什么了。C++有个规则可以解释这个状态：如果解析器在template中遭遇一个 嵌套从属 名称，它就假设这个名称不是个类型，除非你告诉它。所以，我们必须明确告诉它这是个类型：



只要在嵌套从属名称前面加上typename即可。

1. 多态和泛型

C++的多态性

 静态多态性：函数多态性——函数重载

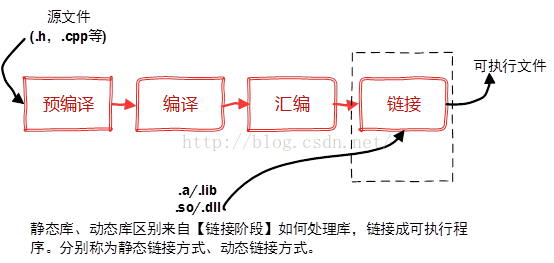
                           模板多态性——*C++*模板（类模板、函数模板）

 动态多态性：虚函数（只有用地址才能实现动态多态性）

1. 静态链接库和动态链接库（巩固）

平时我们写程序都必须include很多头文件，因为可以避免重复造轮子，软件大厦可不是单靠一个人就能完成的。但是你是否知道引用的那些头文件中的函数是怎么被执行的呢？这就要牵扯到链接库了！！！

库有两种，一种是静态链接库，一种是动态链接库，不管是哪一种库，要使用它们，都要在程序中包含相应的include头文件。我们先来回顾一下程序编译的过程。如下图：



我们结合gcc指令来看一下每个阶段生成的文件：

gcc -c helloWorld.c

生成一个helloWorld.o文件，该文件是将源文件编译成的汇编文件，在链接之前，该文件不是可执行文件。

gcc -o helloWorld helloWorld.c

生成的是一个helloWorld的执行文件，格式为ELF（与windows不一样）。该文件为链接后的可执行文件。

* + - 1. 静态链接库

什么是静态链接呢？即在链接阶段，将源文件中用到的库函数与汇编生成的目标文件.o合并生成可执行文件。该可执行文件可能会比较大。

这种链接方式的好处是：方便程序移植，因为可执行程序与库函数再无关系，放在如何环境当中都可以执行。

缺点是：文件太大，一个全静态方式生成的简单print文件都有857K。而动态链接生成的一样的可执行文件却只要8.４Ｋ。

文件内容很简单，就是一个printf("hello world!\n");

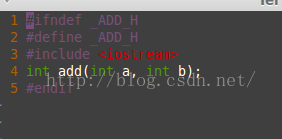
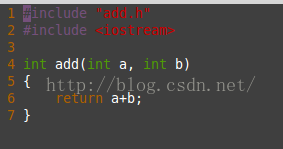
因为包含库文件stdio，所以静态编译出的文件很大。如果你想尝试的话，可以这样编译：

gcc -static -o print print.c

在linux中，静态库为lib\*.a，动态库为lib\*.so。

下面我们来写一个库文件，然后生成一个静态库，然后尝试着调用一下它。

一个简单的add函数，头文件为 头文件对于的源文件：

下面我们来生成静态库：

输入：g++ -c add.cpp 生成.o目标文件

然后用ar命令进一步生成库libadd.a：

ar -crv libadd.a  add.o

这样就生成了一个静态链接库libadd.a。

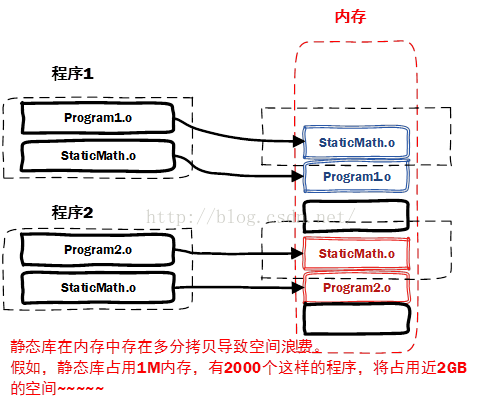
g++ -o test test.cpp -L./addlib -ladd

即汇编后，生成可执行文件之前，与库文件的库链接方式的不同。

* + - 1. 动态链接库

我们知道静态链接的话，文件会很大，往往实现很小的一个功能就需要占用很大的空间，而且每次库文件升级的话，都要重新编译源文件，很不方便。

具体下面如下：



对于静态编译的程序１和程序２，都应用库staticMath。在内存中就又两份相同的staticＭath目标文件，很浪费空间，一旦程序数量过多就很可能会内存不足。

这么大的内存才只能运行这几个程序，实在不甘心。

这样就有了动态库发挥威力的地方了。我们来看看动态链接的结果：

我们看到在这种模型中，两个程序只应用一个库，这个目标文件在内存中只有一份，供所有程序使用。

并且在程序运行过程中动态调用库文件，很方便，又不占空间，但是动态链接有一个缺点就是可移植性太差，如果两台电脑运行环境不同，动态库存放的位置不一样，很可能导致程序运行失败。

在具体的应用中，静态与动态应当合理选择！！！

下面我们来生成一个动态库：

输入：g++ -fPIC -shared -o libadd.so add.cpp

g++ -o test test.cpp -L./addlib -ladd

该命令和刚刚静态链接一样。注意-l后面接的是lib与so中间的库名称。

但是



发现执行程序找不到libadd.so

可以看到test执行程序用到的libadd.so没有找到。。。

原因是在/etc/ld.so.conf文件中设置了动态链接库了寻找路径。



可以看到有很多路径设置文件，在ld.so.conf.d中，我们在下面添加一下我们libadd.so的路径。

然后再执行一下ldconfig命令。

这下就可以成功执行test文件了。

综上说述，静态和动态链接库的选择要视情况而定。一般比较推荐动态链接方式，因为可以很好的节约内存，而且方便以后的库文件升级。

1. C++ 类内存分布

道理非常简单

其实就是类中内存分布的情况

如：

* + - 1. 成员变量所占字节
      2. 虚函数指针所占字节
      3. 继承父类所占字节
      4. 类的非虚函数其实不占用类对象的内存

虚函数也只是vptr占内存空间，32位占4个字节，64位占8个字节

类不会储存在什么地方，只是告诉程序，如何去创建这个类型的变量、需要多少个字节、以及如何销毁等等等。

至于实例，也可以说对象，在什么地方，就是我们通常说的堆栈。抛开全局常量区以及静态存储区这些不谈。

如果你new了一个对象（还有其他的c的分配方式，比如malloc不推荐使用），这种就属于动态分配内存，分配在堆上。堆相对于栈很大，所以很多时候我们分配在堆上，但也意味着，你需要知道什么时候去销毁他，所以与delete或delete []成对出现。

Student stu; 类似于这样的我们没有使用new的对象就分配在了栈上。他的对象是自己管理的。在 {}函数体中的最后会自动按照栈的后进先出，销毁对象。所以不需要你去管理对象的内存，但也意味着对象所在的生命周期通常就是一个{}中。

1. C++聊强行访问私有成员，答按地址访问，反问能不能用优雅地强行访问
2. malloc 底层实现

（看malloc底层.md文件）

# malloc 底层原理

Linux 的虚拟内存管理有几个关键概念： 1、每个进程都有独立的虚拟地址空间，进程访问的虚拟地址并不是真正的物理地址； 2、虚拟地址可通过每个进程上的页表(在每个进程的内核虚拟地址空间)与物理地址进行映射，获得真正物理地址； 3、如果虚拟地址对应物理地址不在物理内存中，则产生缺页中断，真正分配物理地址，同时更新进程的页表；如果此时物理内存已耗尽，则根据内存替换算法淘汰部分页面至物理磁盘中。

### Linux 虚拟地址空间如何分布

Linux 使用虚拟地址空间，大大增加了进程的寻址空间，由低地址到高地址分别为： 1、只读段：该部分空间只能读，不可写；(包括：代码段、rodata 段(C常量字符串和#define定义的常量) ) 2、数据段：保存全局变量、静态变量的空间； 3、堆 ：就是平时所说的动态内存， malloc/new 大部分都来源于此。其中堆顶的位置可通过函数 brk 和 sbrk 进行动态调整。 4、文件映射区域 ：如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间。 5、栈：用于维护函数调用的上下文空间，一般为 8M ，可通过 ulimit –s 查看。 6、内核虚拟空间：用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)。

32 位系统有4G 的地址空间::

其中 0x08048000~0xbfffffff 是用户空间，0xc0000000~0xffffffff 是内核空间，包括内核代码和数据、与进程相关的数据结构（如页表、内核栈）等。另外，%esp 执行栈顶，往低地址方向变化；brk/sbrk 函数控制堆顶\_edata往高地址方向变化。

64位系统结果怎样呢？ 64 位系统是否拥有 2^64 的地址空间吗？ 事实上， 64 位系统的虚拟地址空间划分发生了改变： 1、地址空间大小不是2^32，也不是2^64，而一般是2^48。因为并不需要 2^64 这么大的寻址空间，过大空间只会导致资源的浪费。64位Linux一般使用48位来表示虚拟地址空间，40位表示物理地址， 这可通过 /proc/cpuinfo 来查看 address sizes : 40 bits physical, 48 bits virtual

2、其中，0x0000000000000000~0x00007fffffffffff 表示用户空间， 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF 表示内核空间，共提供 256TB(2^48) 的寻址空间。 这两个区间的特点是，第 47 位与 48~63 位相同，若这些位为 0 表示用户空间，否则表示内核空间。 3、用户空间由低地址到高地址仍然是只读段、数据段、堆、文件映射区域和栈；

### malloc和free是如何分配和释放内存？

#### 缺页导致陷入内核态

当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作： 1、检查要访问的虚拟地址是否合法 2、查找/分配一个物理页 3、填充物理页内容（读取磁盘，或者直接置0，或者啥也不干） 4、建立映射关系（虚拟地址到物理地址） 重新执行发生缺页中断的那条指令 如果第3步，需要读取磁盘，那么这次缺页中断就是majflt，否则就是minflt。

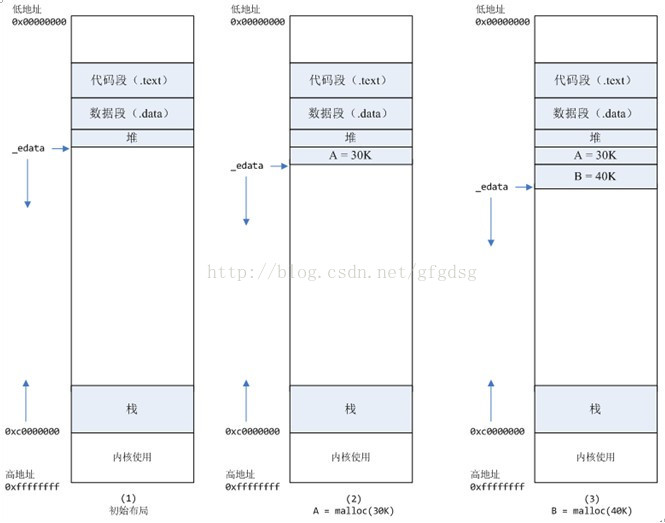
### malloc内存分配的原理

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap（不考虑共享内存）。 　　1、brk是将数据段(.data)的最高地址指针\_edata往高地址推； 　　2、mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。 在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

#### 情况一：malloc小于128k的内存

使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)，如下图：



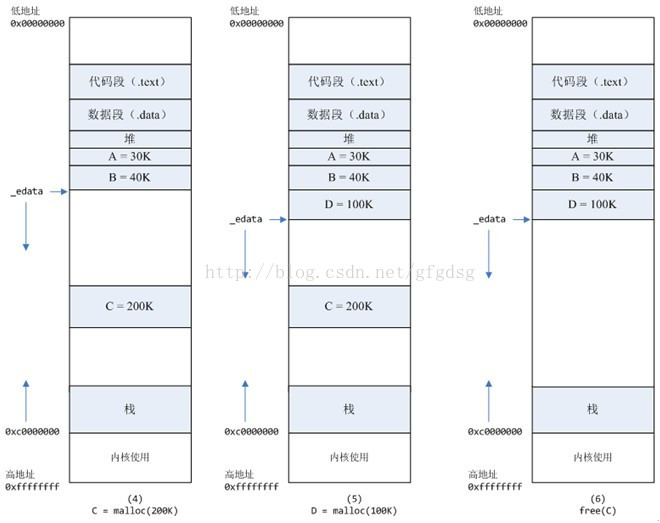
1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。 其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。 edata指针（glibc里面定义）指向数据段的最高地址。 2、进程调用A=malloc(30K)以后，内存空间如图2： malloc函数会调用brk系统调用，将edata指针往高地址推30K，就完成虚拟内存分配。 你可能会问：只要把edata+30K就完成内存分配了？ 事实是这样的，edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。

　　也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。

3、进程调用B=malloc(40K)以后，内存空间如图3

#### 情况二：malloc大于128k的内存

使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：

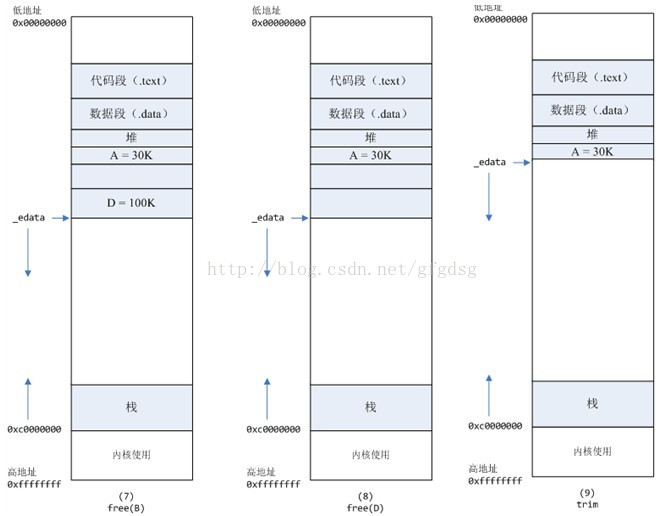


* 进程调用C=malloc(200K)以后，内存空间如图4： 默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M\_MMAP\_THRESHOLD选项调节），那就不是去推\_edata指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。

这样子做主要是因为::

brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候 紧缩 看下面），而mmap分配的内存可以单独释放。 当然，还有其它的好处，也有坏处，再具体下去，有兴趣的同学可以去看glibc里面malloc的代码了。

* 进程调用D=malloc(100K)以后，内存空间如图5；
* 进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。



* 进程调用free(B)以后，如图7所示： B对应的虚拟内存和物理内存都没有释放，因为只有一个\_edata指针，如果往回推，那么D这块内存怎么办呢？ 　　当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。
* 进程调用free(D)以后，如图8所示： B和D连接起来，变成一块140K的空闲内存。
* 默认情况下： 当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩，变成图9所示。

### 既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢？

既然堆内碎片不能直接释放，导致疑似“内存泄露”问题，为什么 malloc 不全部使用 mmap 来实现呢(mmap分配的内存可以会通过 munmap 进行 free ，实现真正释放)？而是仅仅对于大于 128k 的大块内存才使用 mmap ？

其实，进程向 OS 申请和释放地址空间的接口 sbrk/mmap/munmap 都是系统调用，频繁调用系统调用都比较消耗系统资源的。并且， mmap 申请的内存被 munmap 后，重新申请会产生更多的缺页中断。

例如使用 mmap 分配 1M 空间，第一次调用产生了大量缺页中断 (1M/4K 次 ) ，当munmap 后再次分配 1M 空间，会再次产生大量缺页中断。缺页中断是内核行为，会导致内核态CPU消耗较大。另外，如果使用 mmap 分配小内存，会导致地址空间的分片更多，内核的管理负担更大。 ​ 同时堆是一个连续空间，并且堆内碎片由于没有归还 OS ，如果可重用碎片，再次访问该内存很可能不需产 生任何系统调用和缺页中断，这将大大降低 CPU 的消耗。 因此， glibc 的 malloc 实现中，充分考虑了 sbrk 和 mmap 行为上的差异及优缺点，默认分配大块内存 (128k) 才使用 mmap 获得地址空间，也可通过 mallopt(M\_MMAP\_THRESHOLD, <SIZE>) 来修改这个临界值。

### 如何查看进程的缺页中断信息？

可通过以下命令查看缺页中断信息 ：

　　　　ps -o majflt,minflt -C <program\_name>  
　　　　ps -o majflt,minflt -p <pid>

　　其中:: majflt 代表 major fault ，指大错误； minflt 代表 minor fault ，指小错误。 　　这两个数值表示一个进程自启动以来所发生的缺页中断的次数。 majflt 与 minflt 的不同是:: majflt 表示需要读写磁盘，可能是内存对应页面在磁盘中需要load 到物理内存中，也可能是此时物理内存不足，需要淘汰部分物理页面至磁盘中。

Linux内存管理的基本思想之一，是只有在真正访问一个地址的时候才建立这个地址的物理映射。

1. new、delete操作符的底层实现new

new和delete最终调用malloc和free

首先需要知道new*底层调用malloc函数分配内存*



分析：调用malloc失败后会调用\_callnewh。如果\_callnewh返回0则抛出bac\_alloc异常，返回非零则继续分配内存。

这个\_callnewh是什么呢？它是一个new handler，通俗来讲就是new失败的时候调用的回调函数。可以通过\_set\_new\_handler来设置。

1. 创建对象的时候，new做了两件事：
   1. 分配内存（失败返回std::bad\_alloc）
   2. 调用构造函数
2. new操作符对数据类型的处理，分为两种情况
   1. 简单数据类型（包括基本数据类型和不需要构造函数的类型）

总结：

简单类型直接调用operator new分配内存；可以通过new\_handler来处理new失败的情况；new分配失败的时候不像malloc那样返回NULL，它直接抛出异常。要判断是否分配成功应该用异常捕获的机制；

* 1. 复杂数据类型（需要由构造函数初始化对象）

new 复杂数据类型的时候先调用operator new，然后在分配的内存上调用构造函数。

**delete**

delete也分两种情况

第一种是简单数据类型（包括基本数据类型和不需要析构函数的类型）



底层封装也是free。

RTCCALLBACK默认是空的宏定义，所以这个函数默认情况下就是简单的调用free函数。

第二种是复杂数据类型（需要由析构函数销毁对象）

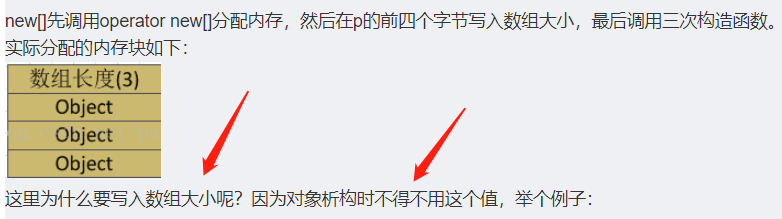
**delete复杂数据类型先调用析构函数再调用operator delete。**

new[]



针对简单类型，new[]计算好大小后调用operator new。

复杂数据类型（需要由析构函数销毁对象）



释放内存之前会调用每个对象的析构函数。但是编译器并不知道p实际所指对象的大小。如果没有储存数组大小，编译器如何知道该把p所指的内存分为几次来调用析构函数呢？总结：**针对复杂类型，new[]会额外存储数组大小**。

**delete[]**

针对简单类型，delete和delete[]等同。

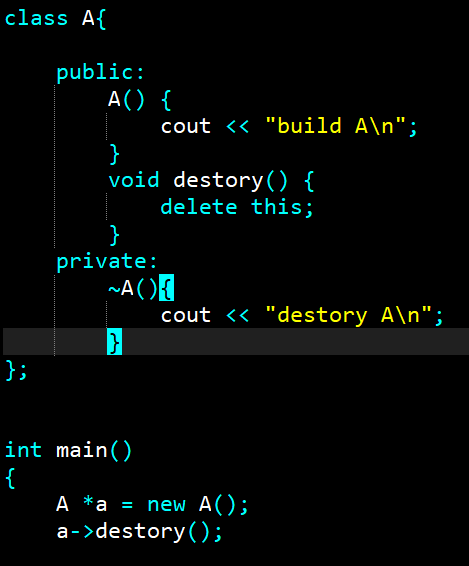
针对复杂类型，new[]出来的内存只能由delete[]释放。原因是因为new[]的过程不一样

1. c++如何定义一个只能在堆上（栈上）生成对象的类？

只能在堆上

方法：将析构函数设置为私有

原因：C++是静态绑定语言，编译器管理栈上对象的生命周期，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性。若析构函数不可访问，则不能再栈上创建对象。



只能在栈上

方法：将new 和 delete 重载为私有

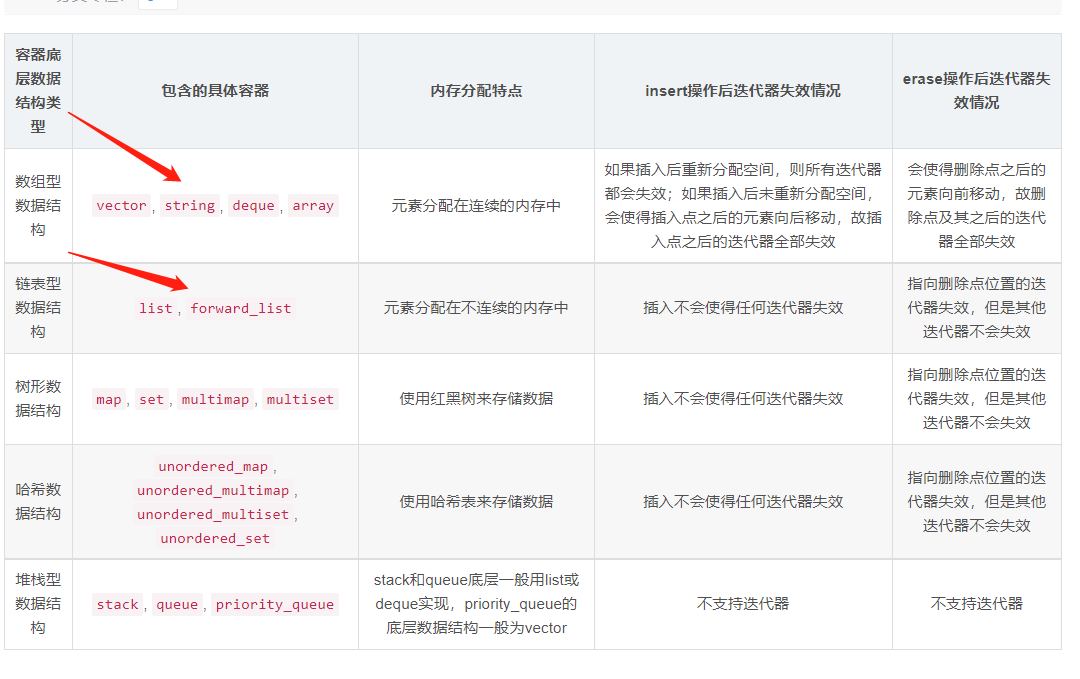
原因：在堆上生成对象，使用new关键词操作，其过程分为两个阶段：第一阶段，使用new在堆上寻找可用内存，分配对象；第二阶段，调用构造函数生成对象，将new操作设置为私有，那么第一阶段无法完成，就不能在堆上生成对象。

private :

void \* operator new ( size\_t t){}

void operator delete ( void \* ptr){}

1. 怎么用一个指向子类的基类指针调用基类的虚函数
2. C++ vector插入和删除为什么会导致迭代器实效



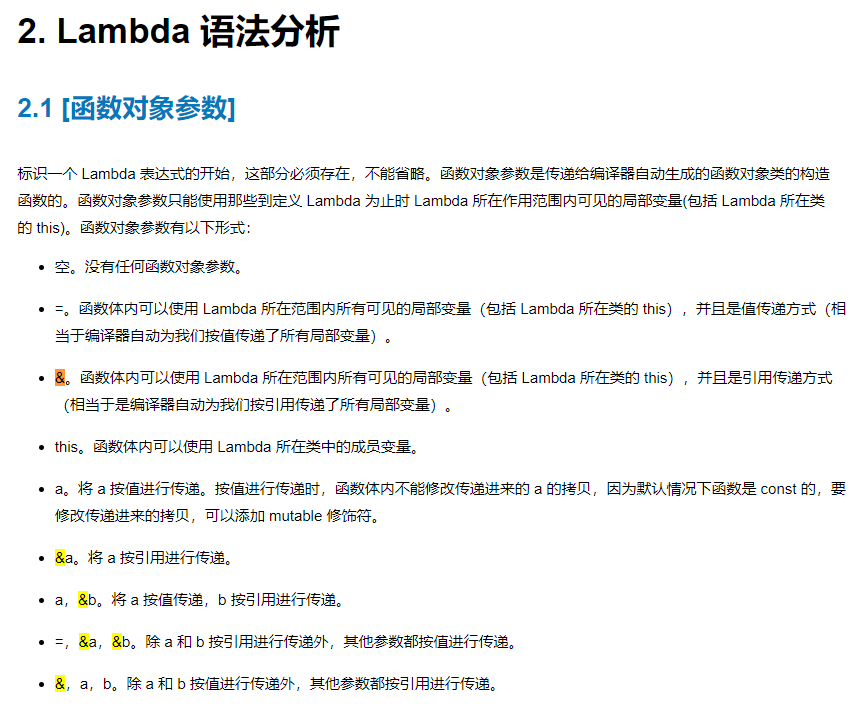
1. 什么是面向对象编程？

面向过程（Procedure Oriented 简称PO ：如C语言）：从名字可以看出它是注重过程的。当解决一个问题的时候，面向过程会把事情拆分成： 一个个函数和数据（用于方法的参数） 。然后按照一定的顺序，执行完这些方法（每个方法看作一个过程），等方法执行完了，事情就搞定了。

面向对象（Object Oriented简称OO ：如C++，JAVA等语言）：看名字它是注重对象的。当解决一个问题的时候，面向对象会把事物抽象成对象的概念，就是说这个问题里面有哪些对象，然后给对象赋一些属性和方法，然后让每个对象去执行自己的方法，问题得到解决。

例子一：问题： 洗衣机里面放有脏衣服，怎么洗干净？面向过程的解决方法：1、执行加洗衣粉方法；2、执行加水方法；3、执行洗衣服方法；4、执行清洗方法；5、 执行烘干方法；以上就是将解决这个问题的过程拆成一个个方法（是没有对象去调用的），通过一个个方法的执行来解决问题。面向对象的解决方法：1、我先弄出两个对象：“洗衣机”对象和“人”对象2、针对对象“洗衣机”加入一些属性和方法：“洗衣服方法”“清洗方法”、“烘干方法”3、针对对象“人”加入属性和方法：“加洗衣粉方法”、“加水方法”4、然后执行人.加洗衣粉人.加水洗衣机.洗衣服洗衣机.清洗洗衣机.烘干解决同一个问题 ，面向对象编程就是先抽象出对象，然后用对象执行方法的方式解决问题。

1. Lambda表达式



1. [c++](https://www.nowcoder.com/jump/super-jump/word?word=c%2B%2B)匿名函数有没有地址
2. STL如何实现哈希表？（\*\*从哈希方法，到哈希冲突解决，到哈希表扩容，以及为什么要用质数\*\*）
3. std::move()的实现原理

C++ 标准库使用比如vector::push\_back 等这类函数时,会对参数的对象进行复制,连数据也会复制.这就会造成对象内存的额外创建, 本来原意是想把参数push\_back进去就行了,通过std::move，可以避免不必要的拷贝操作。

std::move是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝所以可以提高利用效率,改善性能.。

对指针类型的标准库对象并不需要这么做.

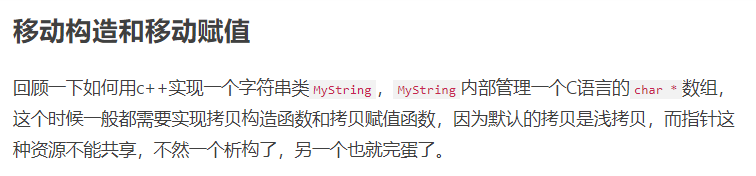
1. 右值引用和move语义（巩固）、移动语义和完美转发

https://www.jianshu.com/p/d19fc8447eaa

首先什么是右值，什么是左值，需要弄清楚



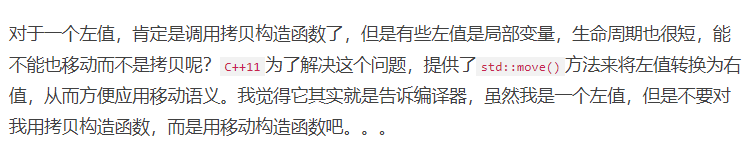
什么是移动语义？



什么是通用引用？



什么是move语义？

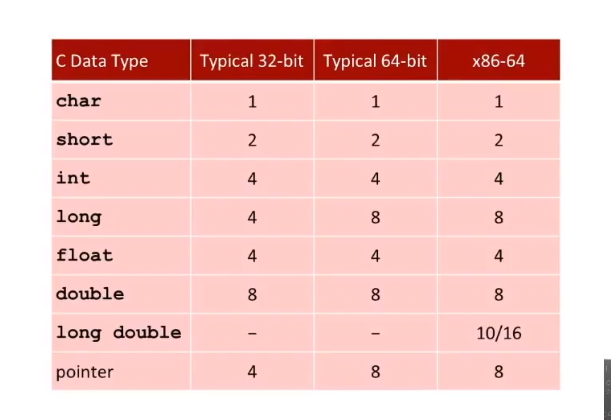


什么是完美转发？





1. gdb调试的多线程调试（去自己的项目中尝试）
2. C++数据类型大小



1. STL释放内存操作

C++ STL 的 vector 容器在 clear() 之后不会释放内存，需要 swap(empty vector)，这是有意为之（C++11 里增加了 shrink\_to\_fit() 函数）。不要记成了所有 STL 容器都需要 swap(empty one) 来释放内存，事实上其他容器（map/set/list/deque）都只需要 clear() 就能释放内存。只有含 reserve()/capacity() 成员函数的容器才需要用 swap 来释放空间，而 C++ 里只有 vector 和 string 这两个符合条件。