和其它语言一样,堆内存中存放"不易失"的数据,底层由 malloc 和 free 进行管理 同样使用`new`关键字于堆上分配内存,但需注意的是,并不是每一次`new`都会于底层调用 malloc() 方法,因为有 着内存池的存在,例如 Python 的对象池 这么写是有问题的 void buzz() { vector<int> *p = new vector<int>(100); 原因在于如果中间省略的代码抛出了异常,将会 函数中若非必需, 导致 delete p; 无法执行,从而导致内存泄露 不要使用堆内存 delete p; } 堆(heap) By the way,什么情况下会在一个函数中申请堆 内存, 然后又在同一个函数中释放? 在绝大部分情况,由一个函数于堆中申请内存,然后再另一个函数中释放申请的内存。但如此以来就会大大地增加忘记释放内 存的操作,所以 C++ 提供了 RAII 来帮助程序员更好的管理内存,可以认为这就是一个简易版的 GC。 Buzz *p = new Buzz[1024]; delete p; 如上述代码所示,delete p 将会完整的释放 malloc 所申请的堆内存,但是,编译器并不知道指针 p 释放内存常见错误 是单个对象,还是多个对象组成的数组,而 delete p 明确告诉编译器指针 p 是单个对象。所以,仅 会调用一次 Buzz 对象的析构函数,从而可能导致内存泄露。 正确的做法是 delete[] p,从而告诉编译器 p 是一个由多个对象所组成的数组,在释放堆内存的 同时,应该调用 1024 次对象的析构函数。 堆、栈与 RAII: 栈由栈帧所组成,函数参数(不考虑寄存器传值)、函数局部变量等内容均在栈帧中保存 C++ 管理资源的方式 栈内存由于其 FILO 的特性,将使得栈内存不可能出现内存碎片的情况,这也是栈内存的效率要比堆内存效率更高的原因 之一,堆内存在绝大多数情况下都会出现内存碎片 栈(stack) 简而言之,在栈中分配内存的对象编译器会在合适的地方插入对构造函数和析构函数的调用。并且,即使在抛出异常的情 况下,对象的析构函数也会被调用(此时我们称之为栈展开) 在栈中分配的对象有一个很重要的特性,即系统会自动地在合适的地方插入对其构造函数和析构函数的调用 C++ 所特有的资源管理方式,使得 C++ 本身并不需要实现 GC 就可以很好的管理内存 ○ RAII 有两大组成部分: 栈和析构函数 class Buzz { public: **RAII** Buzz() {puts("constructor");} (Resource Acquisition Is ~Buzz() {puts("destructor");} Initialization) 当 foo 函数返回时,不管有没有异常发生,都会自动地调 **}**; 用 wrapper 对象的析构函数,而该析构函数中又释放掉了 从堆中申请的内存,从而达到正确释放内存的目的 class BuzzWrapper { private: Buzz *ptr; 在有了 BuzzWrapper 类以后,我们就可以将堆内存中对 象的释放交给它,无需担心因忘记 delete 对象从而导致 public: BuzzWrapper(Buzz *p) : ptr(p) {} 内存泄露的问题了 ~BuzzWrapper() {delete ptr;} Buzz *get() const {return ptr;} 这其实就是 C++ 智能指针的一个基本雏形,当然了,它 **}**; 不是一个模板,所以只能作用在 Buzz 类及其派生类之 上,并且功能非常的单一 void foo() { BuzzWrapper wrapper = BuzzWrapper(new Buzz());