2018年10月9日 星期二

const 和 static

---

const 常量

常量指针和指针常量

1.**常量指针**通常有两种表示方法:

const double * ptr　or　double const *ptr;

可以理解为常量的指针，指针指向的是个常量

关键点:

1. 常量指针指向的对象不能通过这个指针来修改，可是仍然可以通过原来的声明修改;
2. 常量指针可以被赋值为变量的地址，之所以叫常量指针，是限制了通过这个指针修改变量的值;
3. 指针还可以指向别处，因为指针本身只是个变量，可以指向任意地址;

```
int main() {
    int i = 10;
    int i2 = 11;
    const int *p = &i;
    printf("%d\n", *p);//10
    i = 9; //OK,仍然可以通过原来的声明修改值，
    //Error,*p是const int的，不可修改，即常量指针不可修改其指向地址
    //*p = 11;  //error: assignment of read-only location ‘*p’
    p = &i2;//OK,指针还可以指向别处，因为指针只是个变量，可以随意指向;
    printf("%d\n", *p);//11
    return 0;
}
```

2.**指针常量**通常表示为：

double * const ptr;

method 1: const double * ptr;//const读作常量，*读作指针，按照顺序读作常量指针

method 2: double const *ptr;//const读作常量，*读作指针，按照顺序读作常量指针

double * const ptr;//const读作常量，*读作指针，按照顺序读作指针常量

指针常量(指针本身是常量)
定义：
本质是一个常量，而用指针修饰它。指针常量的值是指针，这个值因为是常量，所以不能被赋值。

```
#include <stdio.h>

int main() {
    int i = 10;
    int *const p = &i;
    printf("%d\n", *p);//10
    //Error,因为p是const 指针，因此不能改变p指向的内容
    //p++;//error: increment of read-only variable ‘p’
    (*p)++;  //OK,指针是常量，指向的地址不可以变化,但是指向的地址所对应的内容可以变化
    printf("%d\n", *p);//11
    i = 9;//OK,仍然可以通过原来的声明修改值，
    return 0;
}
```

Static

为什么要引入static?

函数内部定义的变量，在程序执行到它的定义处时，编译器为它在栈上分配空间，大家知道，函数在栈上分配的空间在此函数执行结束时会释放掉，这样就产生了一个问题: 如果想将函数中此变量的值保存至下一次调用时，如何实现？ 最容易想到的方法是定义一个全局的变量，但定义为一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅受此函数控制）。

什么时候用static?

需要一个数据对象为整个类而非某个对象服务,同时又力求不破坏类的封装性,即要求此成员隐藏在类的内部，对外不可见。

static的内部机制：

静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。 这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的main（）函数前的全局数据声明和定义处。 静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。**类声明只声明一个类的“尺寸和规格”，并不进行实际的内存分配，所以在类声明中写成定义是错误的。**它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。 static被引入以告知编译器，将变量存储在程序的静态存储区而非栈上空间，静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。

static的优势：

可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。

引用静态数据成员时，采用如下格式：

<类名>::<静态成员名>　如果静态数据成员的访问权限允许的话(即public的成员)，可在程序中，按上述格式来引用静态数据成员。

注意事项：

(1)类的静态成员函数是属于整个类而非类的对象，所以它没有this指针，这就导致了它仅能访问类的静态数据和静态成员函数。

(2)不能将静态成员函数定义为虚函数。

(3)由于静态成员声明于类中，操作于其外，所以对其取地址操作，就多少有些特殊，变量地址是指向其数据类型的指针 ，函数地址类型是一个“nonmember函数指针”。

(4)由于静态成员函数没有this指针，所以就差不多等同于nonmember函数，结果就产生了一个意想不到的好处：成为一个callback函数，使得我们得以将C++和C-based X Window系统结合，同时也成功的应用于线程函数身上。

(5)static并没有增加程序的时空开销，相反她还缩短了子类对父类静态成员的访问时间，节省了子类的内存空间。

(6)静态数据成员在<定义或说明>时前面加关键字static。

(7)静态数据成员是静态存储的，所以必须对它进行初始化。

(8)静态成员初始化与一般数据成员初始化不同:　初始化在类体外进行，而前面不加static，以免与一般静态变量或对象相混淆；　初始化时不加

该成员的访问权限控制符private，public等；　　初始化时使用作用域运算符来标明它所属类；　　所以我们得出静态数据成员初始化的格式：　　<数据类型><类名>::<静态数据成员名>=<值>

(9)为了防止父类的影响，可以在子类定义一个与父类相同的静态变量，以屏蔽父类的影响。这里有一点需要注意：我们说静态成员为父类和子类共享，但我们有重复定义了静态成员，这会不会引起错误呢？不会，我们的编译器采用了一种绝妙的手法：name-mangling 用以生成唯一的标志。

参考 https://blog.csdn.net/artechtor/article/details/2312766