奋
--- 勤能补拙

昵称：中土
园龄：11年11个月
粉丝：199
关注：1
+加关注

< 2007年6月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

搜索

找找看

谷歌搜索

随笔分类(128)

Algorithm(3)

随笔-231 文章-29 评论-142

static, const, inline, virtual function 辨析

static 是c++中很常用的修饰符，它被用来控制变量的存储方式和可见性，下面我将从 static 修饰符的产生原因、作用谈起，全面分析static 修饰符的实质。

static 的两大作用：

一、控制存储方式：

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。

1、引出原因：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）。

2、 解决方案：因此c++ 中引入了static，用它来修饰变量，它能够指示编译器将此变量在程序的静态存储区分配空间保存，这样即实现了目的，又使得此变量的存取范围不变。

二、控制可见性与连接类型 ：

static还有一个作用，它会把变量的可见范围限制在编译单元中，使它成为一个内部连接，这时，它的

Assembly/IA-32
C(6)
C++ Implementation(3)
C++ Miscellaneous (3)
C++ Specification(33)
C++ Standard Library(4)
Complier/Linker/Loader
Computer Basic(6)
Data Structure(18)
Embedded System(1)
GNU/Linux Dev(17)
Interview Question(23)
Misc.(10)
Network
Operation System(1)

随笔档案(231)
2009年8月 (2)
2008年11月 (2)
2008年10月 (16)
2008年9月 (10)
2008年8月 (9)
2008年6月 (1)
2008年3月 (1)
2008年2月 (1)
2008年1月 (3)
2007年12月 (20)
2007年11月 (43)
2007年10月 (35)
2007年9月 (38)

反义词为”extern”.

static作用分析总结：static总是使得变量或对象的存储形式变成静态存储，连接方式变成内部连接，对于局部变量（已经是内部连接了），它仅改变其存储方式；对于全局变量（已经是静态存储了），它仅改变其连接类型。

类中的static成员：

一、出现原因及作用：

1、需要在一个类的各个对象间交互，即需要一个数据对象为整个类而非某个对象服务。

2、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。

类的static成员满足了上述的要求，因为它具有如下特征：有独立的存储区，属于整个类。

二、注意：

1、对于静态的数据成员，连接器会保证它拥有一个单一的外部定义。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。

2、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。

const 是c++中常用的类型修饰符，但我在工作中发现，许多人使用它仅仅是想当然尔，这样,有时也会用对，但在某些微妙的场合，可就没那么幸运了，究其实质原由，大多因为没有搞清本源。故在本篇中我将对const进行辨析。溯其本源，究其实质，希望能对大家理解const有所帮助，根据思维的承接关系，分为如下

2007年8月 (5)
2007年7月 (21)
2007年6月 (24)

几个部分进行阐述。

c++中为什么会引入const

c++的提出者当初是基于什么样的目的引入（或者说保留）const关键字呢？，这是一个有趣又有益的话题，对理解const很有帮助。

1． 大家知道，c++有一个类型严格的编译系统，这使得c++程序的错误在编译阶段即可发现许多，从而使得出错率大为减少，因此，也成为了c++与c相比，有着突出优点的一个方面。

2． c中很常见的预处理指令 #define variablename variablevalue 可以很方便地进行值替代，这种值替代至少在三个方面优点突出：

一是避免了意义模糊的数字出现，使得程序语义流畅清晰，如下例：
#define user_num_max 107 这样就避免了直接使用107带来的困惑。

二是可以很方便地进行参数的调整与修改，如上例，当人数由107变为201时，进改动此处即可，

三是提高了程序的执行效率，由于使用了预编译器进行值替代，并不需要为这些常量分配存储空间，所以执行的效率较高。

鉴于以上的优点，这种预定义指令的使用在程序中随处可见。

3． 说到这里，大家可能会迷惑上述的1点、2点与const有什么关系呢?,好，请接着向下

看来：

预处理语句虽然有以上的许多优点，但它有个比较致命的缺点，即，预处理语句仅仅只是简单值替代，

缺乏类型的检测机制。这样预处理语句就不能享受c++严格类型检查的好处，从而可能成为引发一系列错误的隐患。

4．好了，第一阶段结论出来了：
结论： const 推出的初始目的，正是为了取代预编译指令，消除它的缺点，同时继承它的优点。

现在它的形式变成了：

const datatype variablename = variablevalue ;

为什么const能很好地取代预定义语句?
const 到底有什么大神通，使它可以振臂一挥取代预定义语句呢?

1． 首先，以const 修饰的常量值，具有不可变性，这是它能取代预定义语句的基础。

2． 第二，很明显，它也同样可以避免意义模糊的数字出现，同样可以很方便地进行参数的调整和修改。

3． 第三，**c++的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，同时，这也是它取代预定义语句的重要基础。这里，我要提一下，为什么说这一点是也是它能取代预定义语句的基础，这是因为，编译器不会去读存储的内容，如果编译器为const分配了存储空间，它就不能够成为一个编译期间的常量了。

4． 最后，const定义也像一个普通的变量定义一样，它会由编译器对它进行类型的检测，消除了预定义语句的隐患。

const 使用情况分类详析

1.const 用于指针的两种情况分析：

```
const int *a;    //a可变，*a不可变 这种写法存在也正确: int const *a;
int *const a;    //a不可变，*a可变
```

分析：const 是一个左结合的类型修饰符，它与其左侧的类型修饰符和为一个类型修饰符，所以，int const 限定 *a,不限定a。int *const 限定a,不限定*a。

2. 传递与返回值

(1)const 限定函数的传递值参数：

```
void fun(const int var);
```

分析：上述写法限定参数在函数体中不可被改变。由值传递的特点可知，var在函数体中的改变不会影响到函数外部。所以，此限定与函数的使用者无关，仅与函数的编写者有关。
结论：最好在函数的内部进行限定，对外部调用者屏蔽，以免引起困惑。如可改写如下：

```
void fun(int var){
const int & varalias = var;

varalias ....

.....

}
```

(2).const 限定函数的值型返回值：

```
const int fun1();
```

```
const myclass fun2();
```

分析:上述写法限定函数的返回值不可被更新，当函数返回内部的类型时（如fun1），已经是一个常量值，当然不可被赋值更新(即作为左值)，所以，此时const无意义，最好去掉，以免困惑。

当函数返回自定义的类型时（如fun2），这个类型仍然包含可以被赋值的变量成员，const表明它就不能作为左值, 即不能被赋值,不能修改.所以，此时有意义。

如果不加const: myclass fun2();
那么很可能就可以这样写: fun2() = obj;//虽然很丑陋，但是的确可以编译通过

3. 传递与返回地址(指针或引用): 此种情况最为常见，由地址变量的特点可知，适当使用const，意义昭然。

(1)const 限定函数的传递指针或引用参数:

```
void fun(const int *pvar);
  void fun(const int &rval);
```

分析: 上述写法传递的为地址，因此函数内很可能改变参数指向的变量，使用const可有效限定参数在函数体中不可被改变。
结论: 除的确需要在函数内改变指针参数指向的变量，一般以地址传递的参数加上const是不错的风格.

(2).const 限定函数的值型返回值:

```
const int * fun1();
const int &fun1();
```

```
//不加const, fun1()可以作为左值而被赋值改变

const myclass *fun2();
const myclass &fun2();
```

分析：一般情况使用引用作为返回类型需要注意：

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生runtime error!

注意事项：

（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak。

（3）可以返回类成员的引用，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（4）流操作符重载返回值申明为“引用”的作用：

流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl; 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用<<操

作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是C++语言中引入引用这个概念的原因吧。赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

例3

```
#i nclude <iostream.h>
int &put(int n);
int vals[10];
int error=-1;
void main()
{
put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;
put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;
cout<<vals[0];
cout<<vals[9];
}
int &put(int n)
{
if (n>=0 && n<=9 ) return vals[n];
else { cout<<"subscript error"; return error; }
}
```

（5）在另外的一些操作符中，却千万不能返回引用：+-*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

5. const 限定类的成员函数：

```
class classname {

   public:

      int fun() const;

   .....

}
```

注意：采用此种const 后置的形式是一种规定，亦为了不引起混淆。在此函数的声明中和定义中均要使用const,因为const已经成为类型信息的一部分。

获得能力：可以操作常量对象。

失去能力：不能修改类的数据成员，不能在函数中调用其他不是const的函数。

在本篇中，const方面的知识我讲的不多，因为我不想把它变成一本c++的教科书。我只是想详细地阐述它的实质和用处. 我会尽量说的很详细，因为我希望在一种很轻松随意的气氛中说出自己的某些想法，毕竟，编程也是轻松，快乐人生的一部分。有时候，你会惊叹这其中的世界原来是如此的精美。

在上篇谈了const后，本篇再来谈一下inline这个关键字，之所以把这篇文章放在这个位置,是因为inline这个关键字的引入原因和const十分相似，下面分为如下几个部分进行阐述。

c++中引入inline关键字的原因：

inline 关键字用来定义一个类的内联函数，引入它的主要原因是用它替代c中表达式形式的宏定义。

表达式形式的宏定义一例：

```
#define expressionname(var1,var2) (var1+var2)*(var1-var2)
```

为什么要取代这种形式呢，且听我道来：

1． 首先谈一下在c中使用这种形式宏定义的原因，c语言是一个效率很高的语言，这种宏定义在形式及使用上像一个函数，但它使用预处理器实现，没有了参数压栈，代码生成等一系列的操作,因此，效率很高，这是它在c中被使用的一个主要原因。

2． 这种宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受c++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。

3． 在c++中引入了类及类的访问控制，这样，如果一个操作或者说一个表达式涉及到类的保护成员或私有成员，你就不可能使用这种宏定义来实现(因为无法将this指针放在合适的位置)。

4． inline 推出的目的，也正是为了取代这种表达式形式的宏定义，它消除了它的缺点，同时又很好地继承了它的优点。

为什么inline能很好地取代表达式形式的预定义呢?

对应于上面的1-3点，阐述如下：

1． inline 定义的类的内联函数，函数的代码被放入符号表中，在使用时直接进行替换，（像宏一样展开），没有了调用的开销，效率也很高。

2． 很明显，类的内联函数也是一个真正的函数，编译器在调用一个内联函数时，会首先检查它的参数的类型，保证调用正确。然后进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。

3． inline 可以作为某个类的成员函数，当然就可以在其中使用所在类的保护成员及私有成员。
在何时使用inline函数：

首先，你可以使用inline函数完全取代表达式形式的宏定义。

另外要注意，内联函数一般只会用在函数内容非常简单的时候，这是因为，内联函数的代码会在任何调用它的地方展开，如果函数太复杂，代码膨胀带来的恶果很可能会大于效率的提高带来的益处。 内联函数最重要的使用地方是用于类的存取函数。

如何使用类的inline函数：

简单提一下inline 的使用吧：

1.在类中定义这种函数：

```
class classname{

.....

....
```

getwidth(){return m_lpicwidth;}; // 如果在类中直接定义，可以不使用inline修饰

....

....

}

2.在类中声明，在类外定义：

class classname{

.....

....

getwidth(); // 如果在类中直接定义，可以不使用inline修饰

....

....

}

inline getwidth(){

return m_lpicwidth;

}

在本篇中，谈了一种特殊的函数，类的inline函数，它的源起和特点在某种说法上与const很类似，可以与const搭配起来看。另外，最近有许多朋友与我mail交往，给我谈论了许多问题，给了我很多启发，在此表示感谢。

面向对象程序设计的基本观点是用程式来仿真大千世界，这使得它的各种根本特性非常人性化，如封装、继承、多态等等，而虚拟函数就是c++中实现多态性的主将。为了实现多态性，c++编译器也革命性地提供了动态绑定（或叫晚捆绑）这一特征。

虚拟函数亦是mfc编程的关键所在,mfc编程主要有两种方法：一是响应各种消息，进行对应的消息处理。二就是重载并改写虚拟函数，来实现自己的某些要求或改变系统的某些默认处理。

虚函数的地位是如此的重要，对它进行穷根究底，力求能知其然并知其所以然 对我们编程能力的提高大有好处。下面且听我道来。

多态性和动态绑定的实现过程分析

一、基础略提（限于篇幅，请参阅相应的c++书籍）：

1、多态性：使用基础类的指针动态调用其派生类中函数的特性。

2、动态联编：在运行阶段，才将函数的调用与对应的函数体进行连接的方式，又叫运行时联编或晚捆绑。

二、过程描述：

1、编译器发现一个类中有虚函数，编译器会立即为此类生成虚拟函数表 vtable（后面有对vtable的

分析）。虚拟函数表的各表项为指向对应虚拟函数的指针。

2、编译器在此类中隐含插入一个指针vptr（对vc编译器来说，它插在类的第一个位置上）。

有一个办法可以让你感知这个隐含指针的存在，虽然你不能在类中直接看到它，但你可以比较一下含有虚拟函数时的类的尺寸和没有虚拟函数时的类的尺寸，你能够发现，这个指针确实存在。

```
class cnovirtualfun
  {
    private:
    long lmember;
    public:
    long getmembervalue（);
  } class chavevirtualfun
  {
    private:
      long lmember;
    public:
      virtual long getmembervalue（);
    }

  cnovirtualfun obj;
  sizeof(obj) -> == 4;
  chavevirtualfun obj;
  sizeof(obj) -> == 8;
```

3、在调用此类的构造函数时，在类的构造函数中，编译器会隐含执行vptr与vtable的关联代码，将vptr指向对应的vtable。这就将类与此类的vtable联系了起来。

4、在调用类的构造函数时，指向基础类的指针此时已经变成指向具体的类的this指针，这样依靠此this指针即可得到正确的vtable，从而实现了多态性。在此时才能真正与函数体进行连接，这就是动态联编。

三、vtable 分析：

分析1：虚拟函数表包含此类及其父类的所有虚拟函数的地址。如果它没有重载父类的虚拟函数，vtable中对应表项指向其父类的此函数。反之，指向重载后的此函数。

分析2：虚拟函数被继承后仍旧是虚拟函数，虚拟函数非常严格地按出现的顺序在 vtable 中排序，所以确定的虚拟函数对应 vtable 中一个固定的位置n，n是一个在编译时就确定的常量。所以，使用vptr加上对应的n，就可得到对应函数的入口地址。

四、编译器调用虚拟函数的汇编码（参考think in c++）：

push funparam ； 先将函数参数压栈

push si ； 将this指针压栈,以确保在当前类上操作

mov bx,word ptr[si] ； 因为vc++编译器将vptr放在类的第一个位置上，所以bx内为vptr

call word ptr[bx+n] ； 调用虚拟函数。n = 所调用的虚拟函数在对应 vtable 中的位置

纯虚函数：

一、引入原因：

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual returntype function()= 0;），则编译器要求在派生类中必须予以重载以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。

二、纯虚函数实质：

1、类中含有纯虚函数则它的vtable表不完全，有一个空位，所以，不能生成对象（编译器绝对不允许有调用一个不存在函数的可能）。在它的派生类中，除非重载这个函数，否则，此派生类的vtable表亦不完整，亦不能生成对象，即它也成为一个纯虚基类。

虚函数与构造、析构函数：

1、构造函数本身不能是虚拟函数；并且虚机制在构造函数中不起作用（在构造函数中的虚拟函数只会调用它的本地版本）。

想一想，在基类构造函数中使用虚机制，则可能会调用到子类，此时子类（应为父类）尚未生成，有何后果！？。

2、析构函数本身常常要求是虚拟函数；但虚机制在析构函数中不起作用。

若类中使用了虚拟函数，析构函数一定要是虚拟函数，比如使用虚拟机制调用delete,没有虚拟的析构函数，怎能保证delete的是你希望delete的对象。

虚机制也不能在析构函数中生效，因为可能会引起调用已经被delete掉的类的虚拟函数的问题。

对象切片：

向上映射(子类被映射到父类)的时候，会发生子类的vtable 完全变成父类的vtable的情况。这就是对象切片。

原因：向上映射的时候，接口会变窄，而编译器绝对不允许有调用一个不存在函数的可能，所以，子类中新派生的虚拟函数的入口在vtable中会被强行“切”掉，从而出现上述情况。

虚拟函数使用的缺点

优点讲了一大堆，现在谈一下缺点，虚函数最主要的缺点是执行效率较低，看一看虚拟函数引发的多态性的实现过程，你就能体会到其中的原因。

分类: C++ Specification

« 上一篇：最小堆与堆排序
» 下一篇：【C++专题】重载(overload)、覆盖(override)、隐藏(hide) 辨析