面向对象程序设计(OOP)

C++(第11章：命名空间和其他高级主题)

福州大学·软件学院·软件工程系
王灿辉(wangcanhui@fzu.edu.cn)

命名空间

- 在使用命名空间之前，有大量的变量、函数和类的名称争用全局命名空间，产生了大量的冲突。例如，如果程序中定义了一个函数pow()，并且和标准库函数的参数表相同，它就覆盖了标准的库函数pow()，因为它们都存储在全局命名空间中。

第11章：命名空间和其他高级主题

命名空间

- 命名空间(namespace)创建了一个可以存放各种程序元素的声明区域，它用来本地化标识符名称，以防止名称冲突。各个命名空间中声明的程序元素都是相互隔离的。命名空间可以帮助组织大型的程序。

第11章：命名空间和其他高级主题

命名空间的定义

- 命名空间定义的一般格式为：

```
namespace name {
    //声明
}
```


- 在namespace语句中定义的任何内容都仅在该命名空间的作用域之内。

第11章：命名空间和其他高级主题

命名空间的引用

- 在命名空间中可以直接引用在该命名空间中声明的标识符，要从命名空间外部访问该命名空间的成员必须在成员名称前使用命名空间的名称和作用域解析运算符。一旦声明了某命名空间中定义的类的对象，就不必进一步对它或它的成员进行限定(因此时命名空间已被解析)。
- 命名空间的定义和引用实例

第11章：命名空间和其他高级主题

命名空间的叠加和嵌套

- 可以声明多个同名的命名空间，它们的定义是叠加的，这样就允许在多个文件中分割命名空间。
- 命名空间允许嵌套，即允许在一个命名空间中定义另一个命名空间。

第11章：命名空间和其他高级主题

命名空间的引入(using语句)

- 使用using可以引入命名空间的特定成员或引入整个命名空间，也可以为命名空间起别名。
- 一个程序可以引入多个命名空间，只要它们互不冲突是完全可以的。当存在冲突时引入命名空间前的问题又回来了。

第11章：命名空间和其他高级主题

命名空间的引入(using语句)

- 在引入命名空间时，一定要注意变量的隐藏和覆盖规则。下面的实例说明了全局变量、局部变量和从命名空间引入的变量的相互作用，同时说明引入特定成员和引入整个命名空间在有的时候存在一些微妙的区别。

第11章：命名空间和其他高级主题

匿名命名空间

- 匿名命名空间允许创建在某一个源程序文件中唯一的标识符，可以用于取代C中的静态(static)全局声明，格式为：

```
namespace {
    //仅在本文件可见的标识符
}
```

第11章：命名空间和其他高级主题

匿名命名空间

文件1	文件2

static int k;
void f1() {
k=99; //正确
}
	extern int k;
	void f2() {
	k=10; //错误
	}

C风格的声明(静态全局声明)

第11章：命名空间和其他高级主题

匿名命名空间

文件1	文件2

namespace {
  static int k;
}
void f1() {
  k=99; //正确
}
	extern int k;
	void f2() {
	  k=10; //错误
	}

C++风格的声明(匿名命名空间)

第11章：命名空间和其他高级主题

std命名空间

- 标准C++的std命名空间包含了它的整个库。所以在大多数例子中均包含语句：using namespace std;当然也可以不引入该命名空间(使用标准C++库较少或可能引发冲突)或仅引入其部分。如：

std::cout<<“Hello!”<<std::endl;

静态类成员

- 普通类的数据成员(每一个对象有其一个拷贝)：

```
class Myclass {
   int i;  //普通私有成员
public:
   Myclass(int x=0) {i=x;}
   void show(void) const {cout<<i<<endl;}
};
void main() {
   Myclass obj1,obj2(88);
   obj1.show(); //输出0
   obj2.show(); //输出88
}
```

第11章：高级主题

类的静态成员

- 在一个类中可以定义静态成员，静态成员是这个类的所有对象所共有的。静态成员分为静态数据成员和静态函数成员。
- 声明：static 类型说明符 成员名；
- (在全局区)定义(分配空间)和初始化：
  类型说明符 类名::成员名[=初值]；
  //没有进行初始化时初始值为0

第11章：高级主题

类的静态成员

- 对类的静态成员的操作和其他成员一样，但注意其只有一个拷贝。也受访问权限控制。对公有静态成员可以用下述格式访问：“类名::公有静态数据成员名”，如：Myclass::pi，私有静态成员只能通过(静态或非静态)成员函数访问(值只有1个)。

第11章：高级主题

类的静态成员

```
class Myclass{
  static int i; //类的私有静态成员
public:
  void show(void){cout<<"i:"<<i<<endl;}
};//私有,类外部不能直接用Myclass::i引用i
int Myclass::i=888; //定义和初始化
void main() {
  Myclass obj1,obj2;
  obj1.show();          //输出i:888
} //obj1改为obj2，输出结果一样
```

第11章：高级主题

类的静态成员

```
class Myclass {
   static int i; //类的静态成员
   int j;        //类的普通成员
public:
   //.....
};
//类的静态成员在类的外面必须有定义(和初始化)！
//static int Myclass::i; //报错，不能有static！
int Myclass::i;
//如果该语句和下一个语句都没有，运行时会报错。
//int Myclass::i=888;
```

第11章：高级主题

定义类的静态成员函数

```
//访问静态成员与非静态成员的方法一样！
Myclass(int x=0,int y=0)
    {i=x;j=y;}
//普通成员函数可访问类的静态数据成员
void showValue() const {
    cout<<"i:"<<i<<"  j:"<<j<<endl;
}
```

第11章：高级主题

类的静态成员函数

- 在成员函数定义前加static即成为类的静态成员函数(不能加const)。静态成员函数只能访问类的静态数据成员，不能**直接访问**类的非静态数据成员，在类的外部引用时必须加“类名::”(当然也可以通过对象名调用，但结果一样)

第11章：高级主题

定义类的静态成员函数

```
static void show(void) { //不能加const！
  cout<<"i:"<<i<<endl;
  //cout<<"j:"<<j<<endl; //错误，直接访问！
}
static void show(const Myclass &obj)
  {cout<<"i:"<<i<<"  j:"<<obj.j<<endl;}
   //将<<i改为<<obj.i结果相同！
```

第11章：高级主题

调用类的静态成员函数

```
Myclass::show();                    //输出i:888
obj1.show(); obj2.show();           //均输出i:518
Myclass::show();                    //也输出i:518
//无论通过类名或任何对象名调用
//(静态数据成员)都得到相同的结果
obj1.show(obj1);          //输出i:518 j:333
Myclass::show(obj1); //输出i:518 j:333
Myclass::show(obj2); //输出i:518 j:88
```

- 静态类成员定义和使用实例

第11章：高级主题

类的静态成员使用实例

- 设计一个学生类，包含学号、姓名、成绩，统计学生的总数和平均分。
- 分析：用对象数组存放学生信息，用两个静态变量分别存放学生总数和学生成绩总和，编写两个静态函数分别用于获得学生总数和学生平均分。
- 一个非常简单的学生类实例

类成员指针

- 在C++语言中，除了可以定义对象指针外，还可以定义类成员指针。类成员指针包括类数据成员指针和类函数成员指针。
- 类成员指针变量不是类的成员，它只是程序中的一个指针变量。所指向的成员必须具有public访问权限。
- 使用方法：
  1)定义类成员指针变量
  2)使类成员指针变量指向类中某个成员
  3)借助该指针变量访问所指向的类成员

第11章：高级主题

类成员指针：类数据成员指针

- 类数据成员指针就是程序中定义的用于指向类中数据成员的指针变量，借助该指针变量可以访问它所指向的类中的数据成员。定义格式如下：

  数据类型 类名::*指针变量名;

  例如：int Myclass::*p;

- 要使已经定义的类数据成员的指针变量指向类中某个数据成员，应使用语句：

  类数据成员指针变量=&类名::类数据成员变量;

  例如：p=&Myclass::a;

- **int Myclass::*q=&Myclass::a;**

第11章：高级主题

类成员指针：类数据成员指针

- 通过下述语句之一可访问类中的数据成员：

  对象名.*类数据成员指针变量名

  对象指针->*类数据成员指针变量名

  例如：obj.*p=100； pObj->*p=200；

- .*和->*两个是新增的运算符

- 完整实例

第11章：高级主题

类成员指针：类函数成员指针

- 类函数成员指针就是程序中定义的用于指向类中函数成员的指针变量，借助该指针变量可以访问它所指向的类中的函数成员。定义格式如下：
  数据类型（类名::*指针变量名）（形参表）；
- 要使已经定义的类函数成员的指针变量指向类中某个函数成员，应使用语句：
  指向类函数成员的指针变量名=&类名::类函数成员名；
- 通过下述语句可以调用类中的函数成员：
  （对象名.*指向类函数成员的指针变量名）（实参表）；

第11章：高级主题

类成员指针：类静态成员指针

- 对类的静态成员的访问不依赖于对象，所以可用普通的指针来指向/访问类的静态成员。
- 声明一个int型指针，指向类的静态数据成员

```
int *count=&Point::countP;
```

- 通过指针访问类的静态函数成员，声明指向函数的指针，指向类的静态成员函数

```
void (*gc)()=&Point::GetC;
```

实例

类的嵌套

- C++允许定义嵌套的类(极少使用)，但嵌套定义中的外部类的成员对内嵌类的成员并没有特殊的访问权，反之也一样，即内嵌类的成员也没有访问外部类的特权。如果他们之间确实想相互访问可以利用友元关系实现。
- 类的嵌套定义及访问权限实例

显式类型转换

- 在**单参数构造函数**前加explicit关键字可以禁止隐式类型转换!但使用形如“类型(…)”或“(类型)…”的显式类型转换仍然可以进行类型转换。例如：

  Complex c3=Complex(8.0);

  c3=(Complex)d+c2; 均可以执行。

- 显式类型转换实例

关键字mutable(易变的)

- 被声明为const的成员函数不能修改类的成员，但当某一数据成员前加上关键字mutable(易变的)后就允许常成员函数对其修改(当然其他非常成员函数更可以修改了)。
- 关键字mutable(易变的)使用实例

类型转换函数

- 可以将自己定义的类型转换为其他数据类型进行混合运算（其他数据类型转换为自己定义的类型可以使用类型转换构造函数），定义格式为：operator 类型(){//函数体} //与析构函数类似，无参也无返回值！同其他函数一样，类型转换函数也可以重载。
- 类型转换函数实例　计量单位转换实例

类型转换函数

- 如果Complex类有单参构造函数，重载过运算符+，并且也存在类型转换函数operator double(){//函数体}，当有语句：c2=c1+2.5;将导致编译时的“二义性”错误。

运行时类型标识(RTTI)

- C++使用类层次结构、虚函数和基类指针实现多态化。基类指针可用于指向基类的对象或指向派生类的对象。因此必须在运行时使用运行时类型标识来确定。使用格式为：typeid(object)//运行时类型识别或typeid(type)　//类型识别
- 如果表达式的类型是类类型且至少包含有一个虚函数，则typeid返回表达式的动态类型，需要在运行时计算；否则，typeid返回表达式的静态类型，在编译时就可以计算。

运行时类型标识(RTTI)

- typeid是C++的关键字(不是函数)，类似于sizeof。typeid操作符的返回结果是名为type_info的标准库类型的对象的引用，ISO C++标准并没有确切定义type_info，它的确切定义编译器相关的，但是标准却规定了其实现必需提供如下几种操作：==、!=、.name()。
- 运行时类型标识(RTTI)实例

第11章：高级主题

运行时类型标识

- typeid最重要的用途是通过多态基类(即至少包含一个虚函数的类)的指针应用它。typeid可以在运行时自动返回基类指针(可以指向基类和任何派生类的对象)指向的实际对象的类型。
- 确定基类指针的类型的实例

第11章：高级主题

运行时类型标识

- 多态类对象引用的用法类似于对象指针。当typeid应用于多态类的对象引用时，它将返回引用指向对象的实际类型，可能是基类型，也可能是派生类型。当对象引用被作为参数传递给函数时，可以使用typeid来获得对象的实际类型。
- 确定多态类对象引用类型的实例

第11章：高级主题

Typeid可以应用于类模板

- ```
  template <class T>
  class MyClass {……}
  MyClass<double> obj3(10.0, 3.0 );
  //……
  cout<<typeid(obj3).name()<<endl;
  ```

- Typeid应用于类模板的实例
  ```

第11章：高级主题

运行时类型标识的应用

- //随机创建派生类的对象(对象工厂)

```
Figure *factory() {
  switch (rand()%4) {
    case 0: return new Triangle(10.1,5.3);
    case 1: return new Rectangle(10.1,5.3);
    case 2: return new Square(10.1);
    case 3: return new Circle(10.1);
  }
  return 0;
}
```


- 运行时类型标识应用实例

强制类型转换运算符

- 宏和强制类型转换是C错误的主要根源。因此C++新增4个强制类型转换运算符：
  dynamic_cast：运行时检查的转换(仅用于在多态类型上执行强制转换)
  static_cast:编译时检查的转换
  const_cast:const转换
  reinterpret_cast:不检查的转换
- 强制类型转换运算符实例

第11章：高级主题

动态类型转换

- 在某些特定的情况下，运算符dynamic_cast可以用来代替typeid。例如：

```
Base *bp;// Base是Derived的多态基类
Derived *dp;
//……
if (typeid(*bp)==typeid(Derived))
  dp=(Derived *)bp; //传统形式的强制转换，它是安全的(因为先检查合法性)，但可以用更好的办法实现，参看下一张幻灯片。
```

第11章：高级主题

动态类型转换

- 可用dynamic_cast代替typeid和if语句：

```
Base *bp;//Base是Derived的多态基类
Derived *dp;
//……
dp=dynamic_cast<Derived *>(bp);
if (dp) //……转换成功，做相应处理
else //cast failed.
```

关键字export

- export关键字(低版本不可用)表示“在其他的编译单位可以访问”，例如：
- //file1.cpp:

```
export template <class T> T twice(T x)
  {return x+x;}
```

- //file2.cpp:

```
template <class T> T twice(T x);
int func(int i){return twice(i);}
```

本章内容讲授到此结束！

福州大学·软件学院·软件工程系
王灿辉(wangcanhui@fzu.edu.cn)