# c++中的数据类型转换

|  |
| --- |
|  |

### 注意：把大类型转化为小类型如int转化为char会丢失精度，而且编译器会发出警告。把小类型转化为大类如char转换int是没有任何问题，也不会有警告，可以进行隐式转换。

# c++的标准输入输出流

|  |
| --- |
|  |

# c++的特性

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
| 注意：两个函数参数相同，返回值不一样是不能构成重载的，或者有默认值参数的和只有一个参数的两个函数也不构成重载，下面2种情况就不是重载 |

## 关于函数重载的一个很特殊的例子

|  |
| --- |
|  |

## 按理说，输出应该是5，5，可是结果却不是

|  |
| --- |
|  |

## 原因也很简单，因为形参的名字和成员变量的名字一样，可能形参被覆盖了。

## 所以，在c++中，函数形参的名字不要和成员变量一样。否则赋值无法成功

上面的问题有2种解决办法：

### 1.就是形参的名字和成员变量的名字不一样。

|  |
| --- |
|  |

### 2.就是使用this指针

|  |
| --- |
|  |

## this指针

|  |
| --- |
|  |

### 注意：this指针是指向类实例对象而不是类本身

### cv++编程的小技巧

|  |
| --- |
|  |

# c++类的继承

## c++有3大特性:继承，封装，多态，我们先来看继承

|  |
| --- |
|  |
| 类的继承方式有public，protected，private 3种，类里面的方法和属性也有这三种方式，public方法可以再类的外面访问，protected的方法子类可以继承，但是不能在外面访问，只能在类里面使用，在外面访问protected方法会报错  私有的方法子类不能继承，也不能访问 |

## c++类继承方式以及规则

|  |
| --- |
|  |

### 通常情况下，我们都是使用public的继承方式，

### 子类初始化对象的时候是会先调用父类的构造函数再调用他本类的构造函数，

|  |
| --- |
|  |

### 程序结束析构的时候，想先执行子类的析构函数，再执行父类的析构函数。顺序会实例化的时候反过来。子类先析构，然后父类才析构。

## 关于子类调用父类带参数的构造函数的问题

### 如果父类只有带参数的构造函数，而子类只有无参数的构造函数，

|  |
| --- |
|  |

### 那么直接运行代码会报错

|  |
| --- |
|  |

### 解决办法有几种，一种种是给父类添加应该没有参数的构造函数，另外一种办法是显示调用父类的带参数构造函数

|  |
| --- |
|  |

### 注意：是在构造函数后面加:然后调用父类的带参数构造函数

### 如果你的类的构造函数没有参数，而你有需要在构造对象的时候给你的类的常量成员变量初始化，你可以怎么写

|  |
| --- |
|  |

## 子类重写父类方法，

### 当父类的方法不能够满足我们子类的需要的时候，你可以重写父类方法，写法如下

|  |
| --- |
|  |

### 注意：子类重写父类的方法必须保证返回值函数名称和参数都完全一样，否则就是新建方法而不是重写

### 函数重写也叫函数覆盖

|  |
| --- |
|  |

### 也可以在子类重写的方法里面调用父类的方法，然后添加子类特有的方法

|  |
| --- |
|  |

# 3.c++的多态特性：可以用父类指针来接受子类对象的地址，是提高虚函数来实现

### 我们先定义一个全局函数，用Animal类的指针作为参数，然后用Animal类的指针变量来接受子类对象的地址，你猜猜此时会调用谁的breathe方法？

|  |
| --- |
|  |

### 结果或许会让你意外，他居然调用Animal类的breathe方法

|  |
| --- |
|  |

### 要搞清楚这个问题，可以看看下面的内存图

|  |
| --- |
|  |

### 当fish对象构建好后，this指针指向他的内存的首地址，也就是animal对象的部分，所以他调用的是Animal类的方法，

### 那么如果我们就是想调用Fish类的breathe方法，该怎么办？

### 需要把父类的方法改为虚函数，在他前面加virtual关键字

|  |
| --- |
|  |

### 此时再运行函数，就会调用Fish类的breathe方法

|  |
| --- |
|  |

### 这种情况就叫做多态，多态的描述如下

|  |
| --- |
|  |

### 把父类的方法设置为虚函数后，如果子类有这个方法的实现，就会调用子类的代码，如果子类没有实现代码，就调用父类的方法，如我们把Fish类的breathe方法注释

|  |
| --- |
|  |

### 然后运行程序，效果如下

|  |
| --- |
|  |

### 发现此时他就调用父类的方法了

## c++中的纯虚函数，就是应该没有实现，并且=0的函数，如图

|  |
| --- |
|  |

### 那么，问题来了，他究竟有什么用？

定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。 纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的默认实现。 所以类纯虚函数的声明就是在告诉子类的设计者，"你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它"。

## 抽象类的介绍

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。

**（1）抽象类的定义：** 称带有纯虚函数的类为抽象类。

**（2）抽象类的作用：** 抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

**（3）使用抽象类时注意：**

* 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。

### 需要注意：

* 1、纯虚函数声明如下： **virtual void funtion1()=0;** 纯虚函数一定没有定义，纯虚函数用来规范派生类的行为，即接口。包含纯虚函数的类是抽象类，抽象类不能定义实例，但可以声明指向实现该抽象类的具体类的指针或引用。
* 2、虚函数声明如下：**virtual ReturnType FunctionName(Parameter)** 虚函数必须实现，如果不实现，编译器将报错，错误提示为：

error LNK\*\*\*\*: unresolved external symbol "public: virtual void \_\_thiscall ClassName::virtualFunctionName(void)"

* 3、对于虚函数来说，父类和子类都有各自的版本。由多态方式调用的时候动态绑定。
* 4、实现了纯虚函数的子类，该纯虚函数在子类中就变成了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。
* 5、虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数。
* 6、在有动态分配堆上内存的时候，析构函数必须是虚函数，但没有必要是纯虚的。
* 7、友元不是成员函数，只有成员函数才可以是虚拟的，因此友元不能是虚拟函数。但可以通过让友元函数调用虚拟成员函数来解决友元的虚拟问题。
* 8、析构函数应当是虚函数，将调用相应对象类型的析构函数，因此，如果指针指向的是子类对象，将调用子类的析构函数，然后自动调用基类的析构函数。

有纯虚函数的类是抽象类，不能生成对象，只能派生。他派生的类的纯虚函数没有被改写，那么，它的派生类还是个抽象类。

定义纯虚函数就是为了让基类不可实例化化，因为实例化这样的抽象数据结构本身并没有意义，或者给出实现也没有意义。

实际上我个人认为纯虚函数的引入，是出于两个目的：

* 1、为了安全，因为避免任何需要明确但是因为不小心而导致的未知的结果，提醒子类去做应做的实现。
* 2、为了效率，不是程序执行的效率，而是为了编码的效率。

# 4.c++变量的引用

## 什么是引用

引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。

## 引用和指针的区别，

### 先看看他们的内存图，指针变量也是需要使用内存空间的。指针变量他是用来保存内存地址的，但是他也有自己的内存地址，不要把指针变量的内存地址好指针变量保存的内存地址混淆。引用不需要内存空间。

|  |
| --- |
|  |

#### 思考一下，如果在一个程序的好多地方都需要使用指针，此时如果用引用来代替指针，是不是就能够提高程序的性能？

### 引用很容易与指针混淆，它们之间有三个主要的不同：

* 不存在空引用。引用必须连接到一块合法的内存，但是有空指针，虽然我们不建议这么做。
* 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
* 引用必须在创建时被初始化。指针可以在任何时间被初始化。

### 实例，如我们可以先创建应该变量a，然后创建应该变量b作为a的引用

|  |
| --- |
|  |

### 此时当我们修改了b内存的的内容，a的值也会改变

|  |
| --- |
|  |
|  |

### 注意，在代码中直接使用引用是没有意义的，反而使得程序的可读性变差而且容易出错，引用是用来做函数传参的。用引用传递参数比用指针传递更加安全高效。

# 5.c++作用域运算符

::是运算符中等级最高的，它分为三种：全局作用域符，类作用域符，命名空间作用域符

## 全局作用

全局作用域符号：当全局变量在局部函数中与其中某个变量重名，那么就可以用::来区分如：   
　　char ch; //全局变量   
　　void sleep（）   
　　{   
　　char ch; //局部变量   
　　ch(局部变量) = ch(局部变量) \*ch(局部变量) ;   
　　::ch(全局变量) =::ch(全局变量) \*ch(局部变量);   
　　}

## 类作用域符号

类作用域符号::的前面一般是类名称，后面一般是该类的成员名称，C++为了避免不同的类有名称相同的成员而采用作用域的方式进行区分。   
　　例如A,B表示两个类，在A,B中都有成员member，那么   
　　A::member就表示类A中的成员member   
　　B::member就表示类B中的成员member 

## 命名空间

“::”是作用域限定符或者称作用域运算符或者作用域操作符（scope operator），例如命名空间。   
“::”作用：namespace::name   
:: 的另一种用法   
直接用在全局函数前，表示是全局函数。当类的成员函数跟类外的一个全局函数同名时，在类内定义的时候，用此函数名默认调用的是本身的成员函数；如果要调用同名的全局函数时，就必须打上::以示区别。

# 6.c ++头文件重复包含的问题

## 如果你在同一个文件里面包含了a.h而a.h又包含了b.h,然后你在这个文件又包含b.h，如果b.h里面又函数或者类的定义，就会方式类或者函数的重复定义，编译器是会报错的，解决办法如下

#### C/C++避免头文件重复包含的方法

* [1. #ifndef](read://https_blog.csdn.net/?url=https%3A%2F%2Fblog.csdn.net%2Fqq_35481167%2Farticle%2Fdetails%2F83619298#1_ifndef_4)
* [2. #pragma once](read://https_blog.csdn.net/?url=https%3A%2F%2Fblog.csdn.net%2Fqq_35481167%2Farticle%2Fdetails%2F83619298#2_pragma_once_39)
* [3. 混合使用](read://https_blog.csdn.net/?url=https%3A%2F%2Fblog.csdn.net%2Fqq_35481167%2Farticle%2Fdetails%2F83619298#3__59)

在实际的编程过程中，因为会使用多个文件，所以在文件中不可避免的要引入一些头文件，这样就可能会出现一个头文件被 **include** 多次，在C/C++中有几种不同的方式可以解决这个问题，总体功能相差不大，但依然存在细微差别。

## 1. #ifndef

格式如下：

#ifndef \_\_COMPLEX\_\_

#define \_\_COMPLEX\_\_

//头文件主体

#endif

或者

#ifndef \_\_COMPLEX\_H\_\_

#define \_\_COMPLEX\_H\_\_

//头文件主体

#endif

**作用：相同的两个文件不会被重复包含。**

**优点：**

1. 受C/C++语言标准的支持，不受编译器的限制。
2. 不仅仅局限于避免同一个文件被重复包含，也能避免内容完全相同的两个文件（或代码片段）被重复包含。

**缺点：**

1. 如果不同头文件中的宏名恰好相同，可能就会导致你看到头文件明明存在，编译器却说找不到声明的情况。
2. 由于编译器每次都需要打开头文件才能判定是否有重复定义，因此在编译大型项目时，#ifndef会使得编译时间相对较长。

## 2. #pragma once

格式如下：

#pragma once

//头文件主体

**作用：物理上的同一个文件不会被重复包含。**

**优点：**

1. 避免#ifndef中因为宏名相同导致的问题。
2. 由于编译器不需要打开头文件就能判定是否有重复定义，因此在编译大型项目时，比#ifndef更快。

**缺点：**

1. #pragma once只针对同一文件有效，对相同的两个文件（或代码片段）使用无效。
2. #pragma once不受一些较老版本的编译器支持，一些支持了的编译器又打算去掉它，所以它的兼容性可能不够好。

## 3. 混合使用

格式如下：

#pragma once

#ifndef \_\_COMPLEX\_H\_\_

#define \_\_COMPLEX\_H\_\_

//头文件主体

#endif

混合使用需要注意的是：  
1.只要使用了#ifndef就会有宏名冲突的危险。  
2.混合使用无法避免不支持#pragma once的编译器报错。  
所以混合使用并不一定能得到很好的效果，具体怎么使用，需要视情况而定。

# 7.c++源程序的编译原理

|  |
| --- |
|  |

## 注意：只有cpp文件被编译了，h文件不参与编译，编译时每一个cpp文件是单独编译的，然后链接器会把使用的目标文件和c++标准库文件进行链接生成一个exe文件

# c++复习到处为止，下一节我们来学习MFC编程

# 项目实例

|  |
| --- |
|  |

## Animal.h

|  |
| --- |
| #pragma once //防止头文件被重复包含  #include<string>  using namespace std;  class Animal  {  public:  string name;  string home;  Animal(void);  //成员函数  virtual void eat();  void sleep();  void live();  ~Animal(void);  }; |

## Animal.cpp

|  |
| --- |
| #include "stdafx.h"  #include "Animal.h"  #include<iostream>  using namespace std;  Animal::Animal(void)  {  this->name = "animal";  this->home = "earth";  cout<<"Animal construct..."<<endl;  }  void Animal::eat()  {  cout<<this->name<<" is eating"<<endl;  }  void Animal::live()  {  cout<<this->name<<" lives in :"<<this->home<<endl;  }  void Animal::sleep()  {  cout<<this->name<<" is sleeping"<<endl;  }  Animal::~Animal(void)  {  cout<<"Animal deconstruct..."<<endl;  } |

## cmdapp.cpp

|  |
| --- |
| // cmdapp1.cpp : 定义控制台应用程序的入口点。  //  #include "stdafx.h"  #include"Animal.h"  #include<iostream>  using namespace std;  class Fish :public Animal  {  public:  Fish()  {  this->name = "Fish";  this->home = "lake or river or sea or stream or pond";  cout<<"Fish construct..."<<endl;  }  ~Fish()  {  cout<<"Fish deconstruct..."<<endl;  }  private:  };  class Cat :public Animal  {  public:  Cat()  {  this->name = "Tom Cat";  this->home = "My master's home";  }  ~Cat()  {  }  private:  };  void reference\_test()  {  int a = 100;  int &b = a;  b = 200;  cout<<"b="<<b<<endl;  cout<<"a="<<a<<endl;  }  int \_tmain(int argc, \_TCHAR\* argv[])  {  /\*Fish f;  f.eat();  f.live();  f.sleep();  cout<<"=============================="<<endl;\*/  /\*Cat c;  c.eat();  c.live();  c.sleep();\*/  reference\_test();  system("pause");  return 0;  } |