# 纯虚函数

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在普通虚函数后面加上“=0”

virtual void Walk() = 0;

**引入原因**

1. 为了方便使用多态的特性，我们常常需要在基类中定义虚拟函数
2. 在很多情况下，基类本身生成对象不合理。例如动物本身作为一个基类被实例化，不合理，当动物类被具象化后，成为狮子老虎类时被实例化才有意义。

# 抽象类

抽象类是一种特殊的类，它不能被实例化，它是为了抽象和设计的目的而建立的。它处于继承层次的较上层。

**如何定义抽象类：带有纯虚函数的类，我们称之为抽象类**

**抽象类的作用**

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。

抽象类的注意点：

抽象类智能作为基类来使用，其纯虚函数的实现由派生类完成，基类不需要给予实现细节。

# 纯虚函数VS虚函数

我们应该注意，纯虚函数属于强制要求子类进行实现的函数，并且将父类构建成了抽象类。

纯虚函数强化了设计结构，同时弱化了父类被实例的能力。而虚函数则实现了纯虚函数的操作，同时兼顾父类可以被实例化的能力

当构建的函数，父类只是描述能力的接口，而非细节时，我们使用纯虚函数，这样强化了结构概念。

当我们需要父类编写的函数，在子类进行选择性延展时，我们可以考虑使用虚函数，父类只描绘了函数功能的一部分，子类需要继承完成。

# 重写，重载，重定义

重写：主要产生在子类和父类之间，指子类对于父类中的虚函数进行了重新的声明和定义，此过程需要保证子类的函数原型和父类的保持一致，并有关键字override标记。

重写注意：

1. 重写出现在继承关系中
2. 重写父类的虚函数，如果虚函数是私有的，或是受保护的，也可以进行重写
3. 重写虚函数后，可以更改函数的访问域

重载：在同一个类内部，针对函数而言，在编写函数时出现函数名相同，参数不同（参数类型，顺序，个数），返回类型可以相同或是不同的两个函数，我们称之为重载操作。返回类型不同不属于重载

注意：

1. 重载发生在同一个类中
2. 不同的返回类型，函数名相同参数列表相同，不属于重载

重定义：发生在子类和父类之间，指子类中重新声明定义了和父类中完全相同的函数，而父类中的函数并不是虚函数，重定义在同一类内部是禁止的。

# 面向对象编程的6个基本准则

1. 单一职责原则：一个类只负责一项职责，一个函数只做一件事情。降低代码的复杂度，增加可读性和维护性。
2. 里氏转换原则：子类可以扩展父类的功能，但不能改变父类原有的功能。

LSP原则：只要父类出现的地方，子类就可以出现（想一下函数参数传递时，参数类型是父类，你填入子类类型数据依旧可以通过）

1. 依赖倒置原则：高层模块不应该依赖底层模块。抽象不应该依赖细节。细节应该依赖抽象。核心就是面向对象编程。
2. 接口隔离原则：一个类对另一个类的依赖应该建立在最小接口上。
3. 迪米特法则：一个对象应该对其他对象保持最少的理解。
4. 开闭原则：对扩展开放，对修改关闭。

# 函数默认参数

在程序设计中，我们经常遇到设计函数时有默认值的情况可以提供给逻辑使用（调用函数时，大部分时间某个参数都使用同一个值），一般我们会填入默认值，以构建更加稳定的函数逻辑，这个默认值就是函数的默认参数值

格式如下

返回类型 函数名（参数类型 参数名=默认值）

### 注意：

1. 函数的默认参数必须在函数列表的末尾，并且在出现的第一个默认参数之后所有参数均要使用默认参数构建
2. 当出现函数重载时，带有默认参数的函数，除去默认参数外剩余参数禁止和重载函数参数相同，否则将产生调用的二义性。
3. 函数在调用时，当函数中存在多个默认参数时，当一个默认参数被填入，那么在此之前的所有默认参数均必须被有效填入

# Const关键字

### 修饰全局或是局部基本数据类型变量

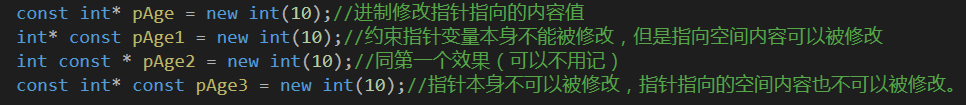
const int H = 10;

int const M = 10;

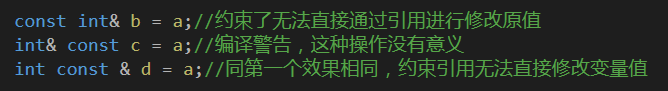
两种写法均声明了常量

常量在头文件中进行构建时，作用域自动转为全局，无需再进行额外的处理。

### 修饰全局或是局部基本数据类型指针



### 修饰引用



### 修饰成员属性

修饰成员属性时，如果在类声明时没有为const修饰的变量进行初始化，编译将报错无法通过。可以通过直接给予变量赋值，**或通过使用初始化参数列表的方式进行赋值均可**。

## 修饰函数参数 保护数据

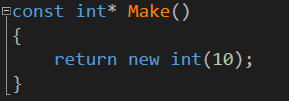
1. 如果修饰普通变量，则没有实际意义，因为参数作为值传递时会被拷贝副本，修改不影响原值，无需进行保护
2. 修饰输入参数指针同指针意义相同
3. 修饰引用，意义如同普通修饰引用

### 修饰函数

1. 如果const挂载到成员函数的末尾，则该函数变为常函数。表明当前函数禁止修改对象的成员变量，并且只能调用常函数（非常函数的函数可能会出现修改成员属性）。

void ChangeAge() const;

1. 函数返回类型是指针，const修饰是返回指针类型，则表示返回后的指针指向的地址禁止修改，并且外部接收的指针变量也必须为const指针





const int\* Hello()；

const int\* a = b1.Hello();

1. 如果函数返回类型是**引用**，const修饰引用，则代表引用的原对象禁止被修改。并且接受函数返回变量是引用的，则必须用const修饰，如果是非引用变量接收，则可以不需要加const修饰

**不建议函数进行引用返回**

1. 如果返回的是值类型，则修饰无意义

Const type GetA（） 没有意义 值会被拷贝

### 修饰自定义数据类型

修饰指针

如果const修饰指针对象，并且在最前面（约束内容禁止被修改）。那么指针只能调用常函数，或是获取成员属性，但是不能够修改成员属性

修饰栈对象

效果同上

# 作用域

一段程序代码中所用到的标识符并不总是有效/可用的，而限定这个标识符的可用性的代码范围就是这个名字的作用域。

**C++作用域分为文件（编译单元头文件和CPP文件），函数，复合语句（｛｝圈定的空间，在函数内），类**

切记

C++中对于命名管理严格，切勿在同一操作域内重复定义变量或函数。

源文件CPP之间属于全局域，注意直接声明在CPP中的变量，即使不在同一个CPP中，也不能重复定义（CPP隶属全局域）

# 声明域

用于进行标识符声明的区域。一般在函数内部，类内部，文件内均可进行声明，不同的声明域将决定了文件的作用域！对应的函数内声明，作用域将在函数内部。类内部声明的非静态成员，作用域在类内部。文件内进行声明，作用域将在文件内。

# Namespace命名空间

是一种描述逻辑分组的机制，可以将按某些标准在逻辑上属于同一个操作的声明放在同一个命名空间中，以解决名字冲突，并提高了声明的可控性，降低了命名空间污染

可以使用Using namespace name；引入空间名

格式

Namespace 空间名

｛

｝

## ：：双冒号

空间内容引用

减少使用using namespace ；

由于编码过程中，C++类库非常的庞大，我们可能在不熟悉类库的过程中操作或是覆盖了已有的函数或是对象，导致无可预知的bug，在一定程度上增加了名称冲突的可能性。

**命名空间可以不连续，每一个命名空间都是一个作用域，作用域中必须保证成员的唯一性，防止出现二义性（变量重复定义操作）**

## Using关键字

Using指令：操作可以达到一次操作，整个空间中的内容均可以使用，作用域的范围延展到整个指令操作域。缺点，容易造成操作域交叉污染。

using namespace *std*;

Using 声明：指定某一个空间中的变量，函数，在当前作用域中有效，再次使用时无需进行完整空间路径书写，优点：方便空间内容调用，缺点：每一个空间内容如果想调用均需要进行声明

using *std*::*cout*;

##### 头文件中构建命名空间

在头文件构建命名空间需要注意，全局变量需要使用extern关键字进行变量声明和定义拆分，以便解决重定义问题

为了解决头文件在多个CPP中进行引用，空间中如果有全局变量将会出现重定义。

.h文件

namespace ZQ

{

extern int PMMM;

}

.cpp文件

int ZQ::PMMM = 100;

Extern关键字：告知编译器遇到此关键字修饰的变量或是函数，只关注声明，定义可以到其他地方查找。定义也可以在声明的地方进行，此关键字帮助将变量声明为全局变量，在头文件中进行全局变量定义使用。

##### 全局命名空间

最外层的命名空间，在全局作用域中定义的名字（即在所有类，函数，及命名空间之外定义的名字）

可以使用：：进行引用，由于全局作用域是隐式的，所以并没有名字，也可以省略双冒号