1 c++概述

1.1 c++简介

“c++”中的++来自于 c 语言中的递增运算符++，该运算符将变量加 1。c++起初也叫”c with class”.通过名称表明，c++是对 C 的扩展，因此 c++是 c 语言的超集，这意味着任何有效的 c 程序都是有效的 c++程序。c++程序可以使用已有的 c 程序库。

库是编程模块的集合，可以在程序中调用它们。库对很多常见的编程问题提供了可靠的解决方法，因此可以节省程序员大量的时间和工作量。

c++语言在 c 语言的基础上添加了面向对象编程和泛型编程的支持。c++继承了 c 语言高效，简洁，快速和可移植的传统。 c++融合了 3 种不同的编程方式: c 语言代表的过程性语言. c++在 c 语言基础上添加的类代表的面向对象语言. c++模板支持的泛型编程。

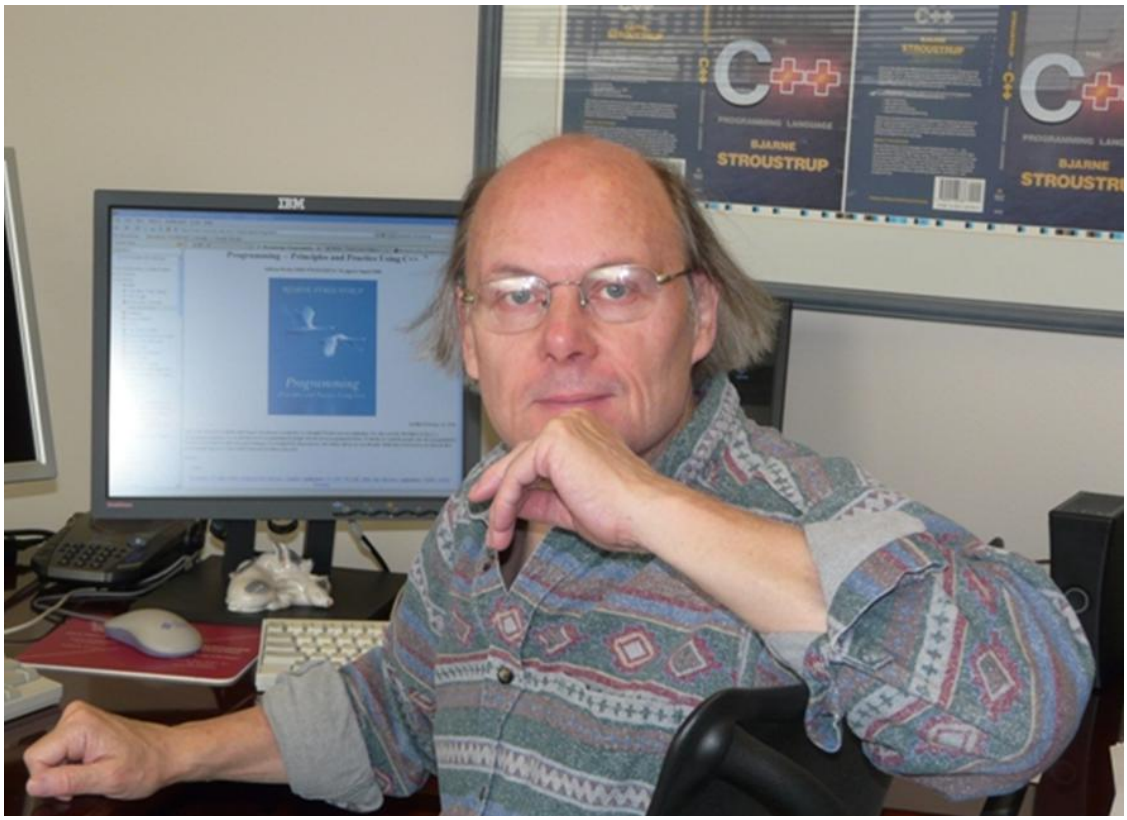
c 语言和 c++语言的关系: c++语言是在 C 语言的基础上，添加了面向对象、模板等现代程序设计语言的特性而发展起来的。两者无论是从语法规则上，还是从运算符的数量和使用上，都非常相似，所以我们常常将这两门语言统称为“C/C++”。

C 语言和 C++并不是对立的竞争关系： 1.C++是 C 语言的加强，是一种更好的 C 语言。 2.C++是以 C 语言为基础的，并且完全兼容 C 语言的特性。

c 语言和 C++语言的学习是可以相互促进。学好 C 语言，可以为我们将来进一步地学习 C++语言打好基础，而 C++语言的学习，也会促进我们对于 C 语言的理解，从而更好地运用 C 语言。

1.2 c++起源

与 c 语言一样，c++也是在贝尔实验室诞生的，Bjarne Stroustrup(本贾尼·斯特劳斯特卢普)在 20 世纪 80 年代在这里开发了这种语言。



(C++之父-本贾尼·斯特劳斯特卢普)

Stroustrup 关心的是让 c++更有用，而不是实施特定的编程原理或风格。在确定语言特性方面，真正的编程比纯粹的原理更重要。Stroustrup 之所以在 c 的基础上创建 c++,是因为 c 语言简洁、适合系统编程、使用广泛且与 UNIX 操作系统联系紧密。 用他自己的话来说，“C++主要是为了我的朋友和我不必再使用汇编语言、C 语言或者其他现代高级语言来编程而设计的。它的主要功能是可以更方便得编写出好程序，让每个程序员更加快乐”。

1.3 可移植性和标准

假设为运行 windows 2000 的老式奔腾 pc 编写了一个很好用的 c++程序，而管理员决定使用不同操作系统(比如说 Mac OS 或 Linux)和处理器的计算机替换它。该程序是否可在新平台运行呢？当然，但是必须使用为新平台设计的 c++编译器重新编译。但是是否需要修改写好的代码？如果不需要修改代码的情况下，重新编译程序后，程序依然运行良好，该程序是可移植的。 程序是否可移植性有两个问题需要解决。第一是硬件，针对特定硬件编程的程序是不可移植的。第二，语言的实现，windows xp c++ 和 Redhat Linux 或 Mac OS X 对 c++的实现不一定相同。虽然我们希望 c++版本与其他版本兼容，但是如果没有一个公开的标准，很难做到。因此，美国国家标准局(American National Standards Institute,ANSI)在 1990 年设立一个

委员会专门负责制定 c++标准(ANSI 制定了 c 语言的标准)。国际标准化组织(International Organization for Standardization，ISO)很快通过自己的委员会加入到这个行列，创建了联合组织 ANSI/ISO,制定 c++标准。 经过多年的努力，制定出了一个国际标准 ISO/IEC 14882:1998 ，并于 1998 年获得了 ISO、IEC(International Electrotechnical Committee,国际电工技术委员会)和 ANSI 的批准。这个标准就是我们经常所说的 c++98。它不仅描述了已有的 c++特性，还对语言进行了扩展，添加了异常、运行阶段类型识别(RTTI)、模板和标准模板库(STL). 2003 年，发布了 c++标准第二版(IOS/IEC 14882:2003),这一版本对第一版修订了一些错误，但并没有改变语言特性，因此 c++98 表示 c++98/c++2003. c++不断发展。IOS 标准委员会于 2011 年 8 月批准了新标准 ISO/IEC 14882:2011,该标准被称为 c++11,与 c++98 一样 c++11 也新增了许多特性。 ISO c++标准还吸收了 ANSI c 语言标准，c++尽量做到是 c 的超集。意味着在理想情况下，任何有效的 c 程序都应该是有效的 c++程序。 ANSI 不仅定义了 c 语言，还 定义了一个 ANSI c 必须实现的标准 c 库。c++也在使用这个库，另外 ANSI/ISO c++标准还提供了一个 c++标准类库。、

2 c++初始

2.1 c++的 helloworld

第一个 helloworld 程序:

```
#include<iostream>
using namespace std;
int main(){
	cout << "hello world" << endl;
	return EXIT_SUCCESS;
}
```

分析: #include; 预编译指令，引入头文件 iostream. using namespace std; 使用标准命名空间 cout << “hello world”<< endl; 和 printf 功能一样，输出字符串”hello wrold” 问题 1：c++头文件为什么没有.h？ 在 c 语言中头文件使用扩展名.h,将其作为一种通过名称标识文件类型的简单方式。但是 c++得用法改变了，c++头文件没有扩展名。但是有些 c 语言的头文件被转换为 c++的头文件，这些文件被重新命名，丢掉了扩展名.h(使之成为 c++风格头文件)，并在文件名称前面加上前缀 c(表明来自 c 语言)。例如 c++版本的 math.h 为 cmath. 由于 C 使用不同的扩展名来表示不同文件类型，因此用一些特殊的扩展名(如 hpp 或 hxx)表示 c++的头文件也是可以的，ANSI/IOS 标准委员会也认为是可以的，但是关键问题是用哪个比较好，最后一致同意不适用任何扩展名。

头文件类型	约定	示例	说明

c++旧式风格	以.h 结尾	iostream.h	c++程序可用
c 旧式风格	以.h 结尾	math.h	c/c++程序可用
c++新式风格	无扩展名	iostream	c++程序可用，使用 namespace std
转换后的 c	加上前缀 c,无扩展名	cmath	c++程序可用，可使用非 c 特性，如 namespace std

问题 2：using namespace std 是什么? namespace 是指标识符的各种可见范围。命名空间用关键字 namespace 来定义。命名空间是 C++的一种机制，用来把单个标识符下的大量有逻辑联系的程序实体组合到一起。此标识符作为此组群的名字。
问题 3：cout 、endl 是什么？ cout 是 c++中的标准输出流，endl 是输出换行并刷新缓冲区。

2.2 面向过程

面向过程是一种以过程为中心的编程思想。 通过分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。面向过程编程思想的核心：功能分解，自顶向下，逐层细化（程序=数据结构+算法）。 面向过程编程语言存在的主要缺点是不符合人的思维习惯，而是要用计算机的思维方式去处理问题，而且面向过程编程语言重用性低，维护困难。

2.3 面向对象

面向对象编程（Object-Oriented Programming）简称 OOP 技术，是开发计算机应用程序的一种新方法、新思想。过去的面向过程编程常常会导致所有的代码都包含在几个模块中，使程序难以阅读和维护。在做一些修改时常常牵一动百，使以后的开发和维护难以为继。而使用 OOP 技术，常常要使用许多代码模块，每个模块都只提供特定的功能，它们是彼此独立的，这样就增大了代码重用的几率，更加有利于软件的开发、维护和升级。

在面向对象中，算法与数据结构被看做是一个整体，称作对象，现实世界中任何类的对象都具有一定的属性和操作，也总能用数据结构与算法两者合一地来描述，所以可以用下面的等式来定义对象和程序：

对象 = 算法 + 数据结构 程序 = 对象 + 对象 + ……

从上面的等式可以看出，程序就是许多对象在计算机中相继表现自己，而对象则是一个个程序实体。 面向对象编程思想的核心：应对变化，提高复用。

2.4 面向对象的三大特点

封装 把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。 类将成员变量和成员函数封装在类的内部，根据需要设置访问权限，通过成员函数管理内部状态。 **继承** 继承所表达的是类之间相关的关系，这种关系使得对象可以继承另外一类对象的特征和能力。 继承的作用：避免公用代码的重复开发，减少代码和数据冗余。 **多态** 多态性可以简单地概括为“一个接口，多种方法”，字面意思为多种形态。程序在运行时才决定调用的函数，它是面向对象编程领域的核心概念。

3 c++对 c 的扩展

3.1 ::作用域运算符

通常情况下，如果有两个同名变量，一个是全局变量，另一个是局部变量，那么局部变量在其作用域内具有较高的优先权，它将屏蔽全局变量。 //全局变量

```
int a = 10;
void test(){
	//局部变量
	int a = 20;
	//全局 a 被隐藏
	cout << "a:" << a << endl;
}
```

程序的输出结果是 a:20。在 test 函数的输出语句中，使用的变量 a 是 test 函数内定义的局部变量，因此输出的结果为局部变量 a 的值。 作用域运算符可以用来解决局部变量与全局变量的重名问题

```
//全局变量
int a = 10;
//1. 局部变量和全局变量同名
void test(){
	int a = 20;
	//打印局部变量 a
	cout << "局部变量 a:" << a << endl;
	//打印全局变量 a
	cout << "全局变量 a:" << ::a << endl;
}
```

这个例子可以看出，作用域运算符可以用来解决局部变量与全局变量的重名问题，即在局部变量的作用域内，可用::对被屏蔽的同名的全局变量进行访问。

3.2 名字控制

创建名字是程序设计过程中一项最基本的活动，当一个项目很大时，它会不可避免地包含大量名字。c++允许我们对名字的产生和名字的可见性进行控制。 我们之前在学习 c 语言可以通过 static 关键字来使得名字只得在本编译单元内可见，在 c++中我们将通过一种通过命名空间来控制对名字的访问。

3.2.1 C++命名空间(namespace)

在 c++中，名称（name）可以是符号常量、变量、函数、结构、枚举、类和对象等等。工程越大，名称互相冲突性的可能性越大。另外使用多个厂商的类库时，也可能导致名称冲突。为了避免，在大规模程序的设计中，以及在程序员使用各种各样的 C++库时，这些标识符的命名发生冲突，标准 C++引入关键字 namespace（命名空间/名字空间/名称空间），可以更好地控制标识符的作用域。

3.2.2 命名空间使用语法

创建一个命名空间:

```
namespace A{
	int a = 10;
}
namespace B{
	int a = 20;
}
void test(){
	cout << "A::a : " << A::a << endl;
	cout << "B::a : " << B::a << endl;
}
```

命名空间只能全局范围内定义（以下错误写法）

```
void test(){
	namespace A{
		int a = 10;
	}
	namespace B{
		int a = 20;
	}
	cout << "A::a : " << A::a << endl;
	cout << "B::a : " << B::a << endl;
}
```

命名空间可嵌套命名空间

```
namespace A{
	int a = 10;
	namespace B{
		int a = 20;
```

```
	}
}
void test(){
	cout << "A::a : " << A::a << endl;
	cout << "A::B::a : " << A::B::a << endl;
}
```

命名空间是开放的，即可以随时把新的成员加入已有的命名空间中

```
namespace A{
	int a = 10;
}

namespace A{
	void func(){
		cout << "hello namespace!" << endl;
	}
}

void test(){
	cout << "A::a : " << A::a << endl;
	A::func();
}
```

声明和实现可分离

```
#pragma once

namespace MySpace{
	void func1();
	void func2(int param);
}

void MySpace::func1(){
	cout << "MySpace::func1" << endl;
}
void MySpace::func2(int param){
	cout << "MySpace::func2 : " << param << endl;
}
```

无名命名空间，意味着命名空间中的标识符只能在本文件内访问，相当于给这个标识符加上了 static，使得其可以作为内部连接

```
namespace{

int a = 10;
void func(){ cout << "hello namespace" << endl; }

}
void test(){
```

```
	cout << "a : " << a << endl;
	func();
}
```

命名空间别名

```
namespace veryLongName{

int a = 10;
void func(){ cout << "hello namespace" << endl; }

}

void test(){
	namespace shortName = veryLongName;
	cout << "veryLongName::a : " << shortName::a << endl;
	veryLongName::func();
	shortName::func();
}
```

3.2.3 using 声明

using 声明可使得指定的标识符可用。

```
namespace A{
	int paramA = 20;
	int paramB = 30;
	void funcA(){ cout << "hello funcA" << endl; }
	void funcB(){ cout << "hello funcA" << endl; }
}

void test(){
	//1. 通过命名空间域运算符
	cout << A::paramA << endl;
	A::funcA();
	//2. using 声明
	using A::paramA;
	using A::funcA;
	cout << paramA << endl;
	//cout << paramB << endl; //不可直接访问
	funcA();
	//3. 同名冲突
	//int paramA = 20; //相同作用域注意同名冲突
}
```

using 声明碰到函数重载

```
namespace A{
	void func(){}
	void func(int x){}
```

```
        int  func(int x,int y){}
}
void test(){
        using A::func;
        func();
        func(10);
        func(10, 20);
}
```

如果命名空间包含一组用相同名字重载的函数，using 声明就声明了这个重载函数的所有集合。

3.2.4 using 编译指令

using 编译指令使整个命名空间标识符可用.

```
namespace A{
        int paramA = 20;
        int paramB = 30;
        void funcA(){ cout << "hello funcA" << endl; }
        void funcB(){ cout << "hello funcB" << endl; }
}

void test01(){
        using namespace A;
        cout << paramA << endl;
        cout << paramB << endl;
        funcA();
        funcB();

//不会产生二义性
int paramA = 30;
cout << paramA << endl;

}

namespace B{
        int paramA = 20;
        int paramB = 30;
        void funcA(){ cout << "hello funcA" << endl; }
        void funcB(){ cout << "hello funcB" << endl; }
}

void test02(){
        using namespace A;
        using namespace B;
        //二义性产生，不知道调用A 还是B 的paramA
        //cout << paramA << endl;
}
```

注意：使用 using 声明或 using 编译指令会增加命名冲突的可能性。也就是说，如果有名称空间，并在代码中使用作用域解析运算符，则不会出现二义性。

3.2.5 命名空间使用

我们刚讲的一些东西一开始会觉得难一些，这些东西以后还是挺常用，只要理解了它们的工作机理，使用它们非常简单。 需要记住的关键问题是当引入一个全局的 using 编译指令时，就为该文件打开了该命名空间，它不会影响任何其他的文件，所以可以在每一个实现文件中调整对命名空间的控制。比如，如果发现某一个实现文件中有太多的 using 指令而产生的命名冲突，就要对该文件做个简单的改变，通过明确的限定或者 using 声明来消除名字冲突，这样不需要修改其他的实现文件。

3.3 全局变量检测增强

c 语言代码：

```
int a = 10; //赋值，当做定义
int a; //没有赋值，当做声明

int main(){
	printf("a:%d\n",a);
	return EXIT_SUCCESS;
}
```

此代码在 c++下编译失败,在 c 下编译通过.

3.4 C++中所有的变量和函数都必须有类型

c 语言代码：

```
//i 没有写类型，可以是任意类型
int fun1(i){
	printf("%d\n", i);
	return 0;
}
//i 没有写类型，可以是任意类型
int fun2(i){
	printf("%s\n", i);
	return 0;
}
//没有写参数，代表可以传任何类型的实参
int fun3(){
	printf("fun33333333333333333\n");
	return 0;
}

//C 语言，如果函数没有参数，建议写 void，代表没有参数
```

```
int fun4(void){
	printf("fun4444444444444\n");
	return 0;
}

g(){
	return 10;
}

int main(){

fun1(10);
fun2("abc");
fun3(1, 2, "abc");
printf("g = %d\n", g());

return 0;

}
```

以上 c 代码 c 编译器编译可通过，c++编译器无法编译通过。

在 C 语言中，int fun() 表示返回值为 int，接受任意参数的函数，int fun(void) 表示返回值为 int 的无参函数。 在 C++ 中，int fun() 和 int fun(void) 具有相同的意义，都表示返回值为 int 的无参函数。

3.5 更严格的类型转换

在 C++，不同类型的变量一般是不能直接赋值的，需要相应的强转。 c 语言代码：

```
typedef enum COLOR{ GREEN, RED, YELLOW } color;
int main(){

color mycolor = GREEN;
mycolor = 10;
printf("mycolor:%d\n", mycolor);
char* p = malloc(10);
return EXIT_SUCCESS;
}
```

以上 c 代码 c 编译器编译可通过，c++编译器无法编译通过。

3.6 struct 类型加强

c 中定义结构体变量需要加上 struct 关键字，c++不需要。 c 中的结构体只能定义成员变量，不能定义成员函数。c++即可以定义成员变量，也可以定义成员函数。

//1. 结构体中即可以定义成员变量，也可以定义成员函数

```
struct Student{
	string mName;
	int mAge;
	void setName(string name){ mName = name; }
	void setAge(int age){ mAge = age; }
	void showStudent(){
		cout << "Name:" << mName << " Age:" << mAge << endl;
	}
};

//2. c++中定义结构体变量不需要加struct 关键字
void test01(){
	Student student;
	student.setName("John");
	student.setAge(20);
	student.showStudent();
}
```

3.7 新增”bool 类型关键字

标准 c++的 bool 类型有两种内建的常量 true(转换为整数 1)和 false(转换为整数 0)表示状态。这三个名字都是关键字。 bool 类型只有两个值，true(1 值)，false(0 值) bool 类型占 1 个字节大小 给 bool 类型赋值时，非 0 值会自动转换为 true(1),0 值会自动转换 false(0)

```
void test()
{	cout << sizeof(false) << endl; //为1，//bool 类型占一个字节大小
	bool flag = true; // c 语言中没有这种类型
	flag = 100; //给 bool 类型赋值时，非0 值会自动转换为true(1),0 值会自动转换false(0)
}
```

c 语言中也有 bool 类型，在 c99 标准之前是没有 bool 关键字，c99 标准已经有 bool 类型，包含头文件 stdbool.h,就可以使用和 c++一样的 bool 类型。

3.8 三目运算符功能增强

c 语言三目运算表达式返回值为数据值，为右值，不能赋值

```
int a = 10;
	int b = 20;
	printf("ret:%d\n", a > b ? a : b);
	//思考一个问题，(a > b ? a : b) 三目运算表达式返回的是什么？

	//(a > b ? a : b) = 100;
	//返回的是右值
```

c++语言三目运算表达式返回值为变量本身(引用)，为左值，可以赋值。

```
int a = 10;
	int b = 20;
	printf("ret:%d\n", a > b ? a : b);
	//思考一个问题，(a > b ? a : b) 三目运算表达式返回的是什么？

	cout << "b:" << b << endl;
	//返回的是左值，变量的引用
	(a > b ? a : b) = 100;//返回的是左值，变量的引用
	cout << "b:" << b << endl;
```

[左值和右值概念] 在 c++中可以放在赋值操作符左边的是左值，可以放到赋值操作符右面的是右值。 有些变量即可以当左值，也可以当右值。 左值为 Lvalue，L 代表 Location，表示内存可以寻址，可以赋值。 右值为 Rvalue，R 代表 Read,就是可以知道它的值。 比如:int temp = 10; temp 在内存中有地址，10 没有，但是可以 Read 到它的值。

3.9 C/C++中的 const

3.9.1 const 概述

const 单词字面意思为常数，不变的。它是 c/c++中的一个关键字，是一个限定符，它用来限定一个变量不允许改变，它将一个对象转换成一个常量。

```
const int a = 10;
A = 100; //编译错误,const 是一个常量，不可修改
```

3.9.2 C/C++中 const 的区别

3.9.2.1 C 中的 const

常量的引进是在 c++早期版本中，当时标准 C 规范正在制定。那时，尽管 C 委员会决定在 C 中引入 const,但是，他们 c 中的 const 理解为”一个不能改变的普通变量”，也就是认为 const 应该是一个只读变量，既然是变量那么就会给 const 分配内存，并且在 c 中 const 是一个全局只读变量，c 语言中 const 修饰的只读变量是外部连接的。 如果这么写:

```
const int arrSize = 10;
int arr[arrSize];
```

看似是一件合理的编码，但是这将得出一个错误。 因为 arrSize 占用某块内存，所以 C 编译器不知道它在编译时的值是多少？

3.9.2.2 C++中的 const

在 c++中，一个 const 不必创建内存空间，而在 c 中，一个 const 总是需要一块内存空间。在 c++中，是否为 const 常量分配内存空间依赖于如何使用。一般说来，如果一个 const 仅仅用来把一个名字用一个值代替(就像使用#define 一样)，那么

该存储局空间就不必创建。 如果存储空间没有分配内存的话，在进行完数据类型检查后，为了代码更加有效，值也许会折叠到代码中。 不过，取一个 const 地址,或者把它定义为 extern,则会为该 const 创建内存空间。 在 c++中，出现在所有函数之外的 const 作用于整个文件(也就是说它在该文件外不可见)，默认为内部连接，c++中其他的标识符一般默认为外部连接。

3.8.2.3 C/C++中 const 异同总结

c 语言全局 const 会被存储到只读数据段。c++中全局 const 当声明 extern 或者对变量取地址时，编译器会分配存储地址，变量存储在只读数据段。两个都受到了只读数据段的保护，不可修改。

```
  const int constA = 10;
  int main(){
       int* p = (int*)&constA;
       *p = 200;
 }
```

以上代码在 c/c++中编译通过，在运行期，修改 constA 的值时，发生写入错误。原因是修改只读数据段的数据。 c 语言中局部 const 存储在堆栈区，只是不能通过变量直接修改 const 只读变量的值，但是可以跳过编译器的检查，通过指针间接修改 const 值。

```
const int constA = 10;
      int* p = (int*)&constA;
      *p = 300;
      printf("constA:%d\n",constA);
      printf("*p:%d\n", *p);
```

运行结果：

c 语言中，通过指针间接赋值修改了 constA 的值。 c++中对于局部的 const 变量要区别对待：

1 对于基础数据类型，也就是 const int a = 10 这种，编译器会把它放到符号表中，不分配内存，当对其取地址时，会分配内存。

```
const int constA = 10;
int* p = (int*)&constA;
*p = 300;
cout << "constA:" << constA << endl;
cout << "*p:" << *p << endl;
```

运行结果：

运行结果：

constA 在符号表中，当我们对 constA 取地址，这个时候为 constA 分配了新的空间，*p 操作的是分配的空间，而 constA 是从符号表获得的值。

2 对于基础数据类型，如果用一个变量初始化 const 变量，如果 const int a = b,那么也是会给 a 分配内存。

```
int b = 10;
const int constA = b;
int* p = (int*)&constA;
*p = 300;
cout << "constA:" << constA << endl;
cout << "*p:" << *p << endl;
```

运行结果：

constA 分配了内存，所以我们可以修改 constA 内存中的值。

3 对于自定数据类型，比如类对象，那么也会分配内存。

```
const Person person; //未初始化 age
//person.age = 50; //不可修改
Person* pPerson = (Person*)&person;
//指针间接修改
pPerson->age = 100;
cout << "pPerson->age:" << pPerson->age << endl;
pPerson->age = 200;
cout << "pPerson->age:" << pPerson->age << endl;
```

运行结果：

为 person 分配了内存，所以我们可以通过指针的间接赋值修改 person 对象。 c 中 const 默认为外部连接，c++中 const 默认为内部连接.当 c 语言两个文件中都有 const int a 的时候，编译器会报重定义的错误。而在 c++中，则不会，因为 c++中的 const 默认是内部连接的。如果想让 c++中的 const 具有外部连接，必须显示声明为: extern const int a = 10;

const 由 c++采用，并加进标准 c 中，尽管他们很不一样。在 c 中，编译器对待 const 如同对待变量一样，只不过带有一个特殊的标记，意思是”你不能改变我”。在 c++中定义 const 时，编译器为它创建空间，所以如果在两个不同文件定义多个同名的 const，链接器将发生链接错误。简而言之,const 在 c++中用的更好。

了解: 能否用变量定义数组: 在支持 c99 标准的编译器中，可以使用变量定义数组。

微软官方描述 vs2013 编译器不支持 c99.： Microsoft C conforms to the standard for the C language as set forth in the 9899:1990 edition of the ANSI C standard.

以下代码在 Linux GCC 支持 c99 编译器编译通过

```
int a = 10;
int arr[a];
int i = 0;
```

```
for(;i<10;i++)
      arr[i] = i;
i = 0;
for(;i<10;i++)
      printf("%d\n",arr[i]);
```

3.9.3 尽量以 const 替换#define

在旧版本 C 中，如果想建立一个常量，必须使用预处理器" #define MAX 1024; 我们定义的宏 MAX 从未被编译器看到过，因为在预处理阶段，所有的 MAX 已经被替换为了 1024，于是 MAX 并没有将其加入到符号表中。但我们使用这个常量获得一个编译错误信息时，可能会带来一些困惑，因为这个信息可能会提到 1024，但是并没有提到 MAX.如果 MAX 被定义在一个不是你写的头文件中，你可能并不知道 1024 代表什么，也许解决这个问题要花费很长时间。 解决办法就是用一个常量替换上面的宏。

```
const int max= 1024;
```

const 和#define 区别总结: 1．const 有类型，可进行编译器类型安全检查。#define 无类型，不可进行类型检查. 2．const 有作用域，而#define 不重视作用域，默认定义处到文件结尾.如果定义在指定作用域下有效的常量，那么#define 就不能用。

1. 宏常量没有类型，所以调用了 int 类型重载的函数。const 有类型，所以调用希望的 short 类型函数？

```
1. #define PARAM 128
   const short param = 128;

void func(short a){
    cout << "short!" << endl;
}
void func(int a){
    cout << "int" << endl;
}
```

1. 宏常量不重视作用域.

```
void func1(){
const int a = 10;
#define A 20
//#undef A  //卸载宏常量A
}
void func2(){
      //cout << "a:" << a << endl; //不可访问，超出了const int a 作用域
      cout << "A:" << A << endl; //#define 作用域从定义到文件结束或者到#undef，可访问
}
```

```
int main(){
	func2();
	return EXIT_SUCCESS;
}
```

问题: 宏常量可以有命名空间吗?

```
namespace MySpace{
	#define num 1024
}
void test(){
	//cout << MySpace::NUM << endl; //错误
	//int num = 100; //命名冲突
	cout << num << endl;
}
```

3.10 引用(reference)

3.10.1 引用基本用法

引用是 c++对 c 的重要扩充。在 c/c++中指针的作用基本都是一样的，但是 c++增加了另外一种给函数传递地址的途径，这就是按引用传递(pass-by-reference)，它也存在于其他一些编程语言中，并不是 c++的发明。

变量名实质上是一段连续内存空间的别名，是一个标号(门牌号) 程序中通过变量来申请并命名内存空间 通过变量的名字可以使用存储空间

对一段连续的内存空间只能取一个别名吗？ c++中新增了引用的概念，引用可以作为一个已定义变量的别名。 基本语法: Type& ref = val; 注意事项： &在此不是求地址运算，而是起标识作用。 类型标识符是指目标变量的类型 必须在声明引用变量时进行初始化。 引用初始化之后不能改变。 不能有 NULL 引用。必须确保引用是和一块合法的存储单元关联。 可以建立对数组的引用。

```
//1. 认识引用
void test01(){

	int a = 10;
	//给变量a 取一个别名b
	int& b = a;
	cout << "a:" << a << endl;
	cout << "b:" << b << endl;
	cout << "------------" << endl;
	//操作b 就相当于操作a 本身
	b = 100;
	cout << "a:" << a << endl;
	cout << "b:" << b << endl;
	cout << "------------" << endl;
	//一个变量可以有n 个别名
```

```
	int& c = a;
	c = 200;
	cout << "a:" << a << endl;
	cout << "b:" << b << endl;
	cout << "c:" << c << endl;
	cout << "------------" << endl;
	//a,b,c 的地址都是相同的
	cout << "a:" << &a << endl;
	cout << "b:" << &b << endl;
	cout << "c:" << &c << endl;
}
//2. 使用引用注意事项
void test02(){
	//1) 引用必须初始化
	//int& ref; //报错:必须初始化引用
	//2) 引用一旦初始化，不能改变引用
	int a = 10;
	int b = 20;
	int& ref = a;
	ref = b; //不能改变引用
	//3) 不能对数组建立引用
	int arr[10];
	//int& ref3[10] = arr;
}

	//1. 建立数组引用方法一
	typedef int ArrRef[10];
	int arr[10];
	ArrRef& aRef = arr;
	for (int i = 0; i < 10;i ++){
		aRef[i] = i+1;
	}
	for (int i = 0; i < 10;i++){
		cout << arr[i] << " ";
	}
	cout << endl;
	//2. 建立数组引用方法二
	int(&f)[10] = arr;
	for (int i = 0; i < 10; i++){
		f[i] = i+10;
	}
	for (int i = 0; i < 10; i++){
		cout << arr[i] << " ";
	}
	cout << endl;
```

3.10.2 函数中的引用

最常见看见引用的地方是在函数参数和返回值中。当引用被用作函数参数的时，在函数内对任何引用的修改，将对还函数外的参数产生改变。当然，可以通过传递一个指针来做相同的事情，但引用具有更清晰的语法。 如果从函数中返回一个引用，必须像从函数中返回一个指针一样对待。当函数返回值时，引用关联的内存一定要存在。

```
//值传递
void ValueSwap(int m,int n){
	int temp = m;
	m = n;
	n = temp;
}
//地址传递
void PointerSwap(int* m,int* n){
	int temp = *m;
	*m = *n;
	*n = temp;
}
//引用传递
void ReferenceSwap(int& m,int& n){
	int temp = m;
	m = n;
	n = temp;
}
void test(){
	int a = 10;
	int b = 20;
	//值传递
	ValueSwap(a, b);
	cout << "a:" << a << " b:" << b << endl;
	//地址传递
	PointerSwap(&a, &b);
	cout << "a:" << a << " b:" << b << endl;
	//引用传递
	ReferenceSwap(a, b);
	cout << "a:" << a << " b:" << b << endl;
}
```

通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单： 1) 函数调用时传递的实参不必加“&”符 2) 在被调函数中不必在参数前加“*”符 引用作为其它变量的别名而存在，因此在一些场合可以代替指针。C++主张用引用传递取代地址传递的方式，因为引用语法容易且不易出错。

```
//返回局部变量引用
int& TestFun01(){
	int a = 10; //局部变量
```

```
	return a;
}
//返回静态变量引用
int& TestFunc02(){
	static int a = 20;
	cout << "static int a : " << a << endl;
	return a;
}
int main(){
	//不能返回局部变量的引用
	int& ret01 = TestFun01();
	//如果函数做左值，那么必须返回引用
	TestFunc02();
	TestFunc02() = 100;
	TestFunc02();

	return EXIT_SUCCESS;
}
```

不能返回局部变量的引用。 函数当左值，必须返回引用。

3.10.3 引用的本质

引用的本质在 c++内部实现是一个指针常量. Type& ref = val; // Type* const ref = &val;

c++编译器在编译过程中使用常指针作为引用的内部实现，因此引用所占用的空间大小与指针相同，只是这个过程是编译器内部实现，用户不可见。

```
//发现是引用，转换为 int* const ref = &a;
void testFunc(int& ref){
	ref = 100; // ref 是引用，转换为*ref = 100
}
int main(){
	int a = 10;
	int& aRef = a; //自动转换为 int* const aRef = &a;这也能说明引用为什么必须初始化
	aRef = 20; //内部发现 aRef 是引用，自动帮我们转换为: *aRef = 20;
	cout << "a:" << a << endl;
	cout << "aRef:" << aRef << endl;
	testFunc(a);
	return EXIT_SUCCESS;
}
```

3.10.4 指针引用

在 c 语言中如果想改变一个指针的指向而不是它所指向的内容，函数声明可能这样:

```
void fun(int**);
给指针变量取一个别名。
Type* pointer = NULL;
Type*& = pointer;
Type* pointer = NULL;  Type*& = pointer;
struct Teacher{
	int mAge;
};
//指针间接修改 teacher 的年龄
void AllocateAndInitByPointer(Teacher** teacher){
	*teacher = (Teacher*)malloc(sizeof(Teacher));
	(*teacher)->mAge = 200;
}
//引用修改 teacher 年龄
void AllocateAndInitByReference(Teacher*& teacher){
	teacher->mAge = 300;
}
void test(){
	//创建 Teacher
	Teacher* teacher = NULL;
	//指针间接赋值
	AllocateAndInitByPointer(&teacher);
	cout << "AllocateAndInitByPointer:" << teacher->mAge << endl;
	//引用赋值,将 teacher 本身传到 ChangeAgeByReference 函数中
	AllocateAndInitByReference(teacher);
	cout << "AllocateAndInitByReference:" << teacher->mAge << endl;
	free(teacher);
}
```

对于 c++中的定义那个，语法清晰多了。函数参数变成指针的引用，用不着取得指针的地址。

3.10.5 常量引用

常量引用的定义格式:

```
const Type& ref = val;
```

常量引用注意： 字面量不能赋给引用，但是可以赋给 const 引用 const 修饰的引用，不能修改。

```
void test01(){
	int a = 100;
	const int& aRef = a; //此时 aRef 就是 a
	//aRef = 200; 不能通过 aRef 的值
	a = 100; //OK
	cout << "a:" << a << endl;
	cout << "aRef:" << aRef << endl;
}
```

```
void test02(){
	//不能把一个字面量赋给引用
	//int& ref = 100;
	//但是可以把一个字面量赋给常引用
	const int& ref = 100; //int temp = 200; const int& ret = temp;
}
```

[const 引用使用场景] 常量引用主要用在函数的形参，尤其是类的拷贝/复制构造函数。 将函数的形参定义为常量引用的好处: 引用不产生新的变量，减少形参与实参传递时的开销。 由于引用可能导致实参随形参改变而改变，将其定义为常量引用可以消除这种副作用。 如果希望实参随着形参的改变而改变，那么使用一般的引用，如果不希望实参随着形参改变，那么使用常引用。

```
//const int& param 防止函数中意外修改数据
void ShowVal(const int& param){
	cout << "param:" << param << endl;
}
```

3.11 练习作业

1. 设计一个类，求圆的周长。
2. 设计一个学生类，属性有姓名和学号，可以给姓名和学号赋值，可以显示学生的姓 名和学号

3.12 内联函数(inline function)

3.12.1 内联函数的引出

c++从 c 中继承的一个重要特征就是效率。假如 c++的效率明显低于 c 的效率，那么就会有很大的一批程序员不去使用 c++了。 在 c 中我们经常把一些短并且执行频繁的计算写成宏，而不是函数，这样做的理由是为了执行效率，宏可以避免函数调用的开销，这些都由预处理来完成。 但是在 c++出现之后，使用预处理宏会出现两个问题： 第一个在 c 中也会出现，宏看起来像一个函数调用，但是会有隐藏一些难以发现的错误。 第二个问题是 c++特有的，预处理器不允许访问类的成员，也就是说预处理器宏不能用作类类的成员函数。

为了保持预处理宏的效率又增加安全性，而且还能像一般成员函数那样可以在类里访问自如，c++引入了内联函数(inline function).

内联函数为了继承宏函数的效率，没有函数调用时开销，然后又可以像普通函数那样，可以进行参数，返回值类型的安全检查，又可以作为成员函数。

3.12.2 预处理宏的缺陷

预处理器宏存在问题的关键是我们可能认为预处理器的行为和编译器的行为是一样的。当然也是由于宏函数调用和函数调用在外表看起来是一样的，因为也容易被混淆。但是其中也会有一些微妙的问题出现: 问题一：

```
#define ADD(x,y) x+y
inline int Add(int x,int y){
	return x + y;
}
void test(){
	int ret1 = ADD(10, 20) * 10; //希望的结果是300
	int ret2 = Add(10, 20) * 10; //希望结果也是300
	cout << "ret1:" << ret1 << endl; //210
	cout << "ret2:" << ret2 << endl; //300
}
```

问题二：

```
#define COMPARE(x,y) ((x) < (y) ? (x) : (y))
int Compare(int x,int y){
	return x < y ? x : y;
}
void test02(){
	int a = 1;
	int b = 3;
	//cout << "COMPARE(++a, b):" << COMPARE(++a, b) << endl; // 3
	cout << "Compare(int x,int y):" << Compare(++a, b) << endl; //2
}
```

问题三:

预定义宏函数没有作用域概念，无法作为一个类的成员函数，也就是说预定义宏没有办法表示类的范围。

3.12.3 内联函数

3.12.3.1 内联函数基本概念

在 c++中，预定义宏的概念是用内联函数来实现的，而内联函数本身也是一个真正的函数。内联函数具有普通函数的所有行为。唯一不同之处在于内联函数会在适当的地方像预定义宏一样展开，所以不需要函数调用的开销。因此应该不使用宏，使用内联函数。 在普通函数(非成员函数)函数前面加上 inline 关键字使之成为内联函数。但是必须注意必须函数体和声明结合在一起，否则编译器将它作为普通函数来对待。 inline void func(int a); 以上写法没有任何效果，仅仅是声明函数，应该如下方式来做: inline int func(int a){return ++;} 注意: 编译器将会检查函数参数列表使用是否正确，并返回值(进行必要的转换)。这些事预处理器无法完成的。 内联函数

的确占用空间，但是内联函数相对于普通函数的优势只是省去了函数调用时候的压栈，跳转，返回的开销。我们可以理解为内联函数是以空间换时间。

3.12.3.2 类内部的内联函数

为了定义内联函数，通常必须在函数定义前面放一个 inline 关键字。但是在类内部定义内联函数时并不是必须的。任何在类内部定义的函数自动成为内联函数。

```
class Person{
public:
	Person(){ cout << "构造函数!" << endl; }
	void PrintPerson(){ cout << "输出 Person!" << endl; }
}
```

3.12.3.3 内联函数和编译器

内联函数并不是何时何地都有效，为了理解内联函数何时有效，应该要知道编译器碰到内联函数会怎么处理？ 对于任何类型的函数，编译器会将函数类型(包括函数名字，参数类型，返回值类型)放入到符号表中。同样，当编译器看到内联函数，并且对内联函数体进行分析没有发现错误时，也会将内联函数放入符号表。 当调用一个内联函数的时候，编译器首先确保传入参数类型是正确匹配的，或者如果类型不正完全匹配，但是可以将其转换为正确类型，并且返回值在目标表达式里匹配正确类型，或者可以转换为目标类型，内联函数就会直接替换函数调用，这就消除了函数调用的开销。假如内联函数是成员函数，对象 this 指针也会被放入合适位置。 类型检查和类型转换、包括在合适位置放入对象 this 指针这些都是预处理器不能完成的。

但是 c++内联编译会有一些限制，以下情况编译器可能考虑不会将函数进行内联编译： 不能存在任何形式的循环语句 不能存在过多的条件判断语句 函数体不能过于庞大 不能对函数进行取址操作

内联仅仅只是给编译器一个建议，编译器不一定会接受这种建议，如果你没有将函数声明为内联函数，那么编译器也可能将此函数做内联编译。一个好的编译器将会内联小的、简单的函数。

3.13 函数的默认参数

c++在声明函数原型的时可为一个或者多个参数指定默认(缺省)的参数值，当函数调用的时候如果没有指定这个值，编译器会自动用默认值代替。

```
void TestFunc01(int a = 10, int b = 20){
	cout << "a + b  = " << a + b << endl;
}
//注意点:
//1. 形参 b 设置默认参数值，那么后面位置的形参 c 也需要设置默认参数
void TestFunc02(int a,int b = 10,int c = 10){}
```

```
//2. 如果函数声明和函数定义分开，函数声明设置了默认参数，函数定义不能再设置默认参数
void TestFunc03(int a = 0,int b = 0);
void TestFunc03(int a, int b){}

int main(){
	//1.如果没有传参数，那么使用默认参数
	TestFunc01();
	//2. 如果传一个参数，那么第二个参数使用默认参数
	TestFunc01(100);
	//3. 如果传入两个参数，那么两个参数都使用我们传入的参数
	TestFunc01(100, 200);

	return EXIT_SUCCESS;
}
```

注意点： 函数的默认参数从左向右，如果一个参数设置了默认参数，那么这个参数之后的参数都必须设置默认参数。 如果函数声明和函数定义分开写，函数声明和函数定义不能同时设置默认参数。

3.14 函数的占位参数

c++在声明函数时，可以设置占位参数。占位参数只有参数类型声明，而没有参数名声明。一般情况下，在函数体内部无法使用占位参数。

```
void TestFunc01(int a,int b,int){
	//函数内部无法使用占位参数
	cout << "a + b = " << a + b << endl;
}
//占位参数也可以设置默认值
void TestFunc02(int a, int b, int = 20){
	//函数内部依旧无法使用占位参数
	cout << "a + b = " << a + b << endl;
}
int main(){

	//错误调用，占位参数也是参数，必须传参数
	//TestFunc01(10,20);
	//正确调用
	TestFunc01(10,20,30);
	//正确调用
	TestFunc02(10,20);
	//正确调用
	TestFunc02(10, 20, 30);

	return EXIT_SUCCESS;
}
```

什么时候用，在后面我们要讲的操作符重载的后置++要用到这个.

3.15 函数重载(overload)

3.15.1 函数重载概述

能使名字方便使用，是任何程序设计语言的一个重要特征！

我们现实生活中经常会碰到一些字在不同的场景下具有不同的意思，比如汉语中的多音字“重”。 当我们说:“他好重啊，我都背不动!”我们根据上下文意思，知道“重”在此时此地表示重量的意思。 如果我们说“你怎么写了那么多重复的代码? 维护性太差了!”这个地方我们知道，“重”表示重复的意思。 同样一个字在不同的场景下具有不同的含义。那么在 c++中也有一种类似的现象出现，同一个函数名在不同场景下可以具有不同的含义。 在传统 c 语言中，函数名必须是唯一的，程序中不允许出现同名的函数。在 c++中是允许出现同名的函数，这种现象称为函数重载。 函数重载的目的就是为了方便的使用函数名。 函数重载并不复杂，等大家学完就会明白什么时候需要用到他们，以及是如何编译，链接的。

3.15.2 函数重载

3.15.2.1 函数重载基本语法

实现函数重载的条件： 同一个作用域 参数个数不同 参数类型不同 参数顺序不同

```
//1. 函数重载条件
namespace A{
	void MyFunc(){ cout << "无参数!" << endl; }
	void MyFunc(int a){ cout << "a: " << a << endl; }
	void MyFunc(string b){ cout << "b: " << b << endl; }
	void MyFunc(int a, string b){ cout << "a: " << a << " b:" << b <<
 endl;}
    void MyFunc(string b, int a){cout << "a: " << a << " b:" << b << en
dl;}
}
//2.返回值不作为函数重载依据
namespace B{
	void MyFunc(string b, int a){}
	//int MyFunc(string b, int a){} //无法重载仅按返回值区分的函数
}
```

注意: 函数重载和默认参数一起使用，需要额外注意二义性问题的产生。

```
void MyFunc(string b){
	cout << "b: " << b << endl;
}
//函数重载碰上默认参数
void MyFunc(string b, int a = 10){
	cout << "a: " << a << " b:" << b << endl;
```

```
}
int main(){
      MyFunc("hello"); //这时，两个函数都能匹配调用，产生二义性
      return 0;
}
```

思考：为什么函数返回值不作为重载条件呢？ 当编译器能从上下文中确定唯一的函数的时，如 int ret = func(),这个当然是没有问题的。然而，我们在编写程序过程中可以忽略他的返回值。那么这个时候,一个函数为 void func(int x);另一个为 int func(int x); 当我们直接调用 func(10),这个时候编译器就不确定调用那个函数。所以在 c++中禁止使用返回值作为重载的条件。

3.15.2.2 函数重载实现原理

编译器为了实现函数重载，也是默认为我们做了一些幕后的工作，编译器用不同的参数类型来修饰不同的函数名，比如 void func(); 编译器可能会将函数名修饰成 *func，当编译器碰到 void func(int x),编译器可能将函数名修饰为* func*int,当编译器碰到 void func(int x,char c),编译器可能会将函数名修饰为* func*int*char 我这里使用”可能”这个字眼是因为编译器如何修饰重载的函数名称并没有一个统一的标准，所以不同的编译器可能会产生不同的内部名。

```
void func(){}
void func(int x){}
void func(int x,char y){}
```

以上三个函数在 linux 下生成的编译之后的函数名为:

```
_Z4funcv //v 代表 void,无参数
_Z4funci //i 代表参数为 int 类型
_Z4funcic //i 代表第一个参数为 int 类型，第二个参数为 char 类型
```

3.15.3 extern “C”浅析

以下在 Linux 下测试: c 函数: void MyFunc(){} ,被编译成函数: MyFunc c++函数: void MyFunc(){},被编译成函数: _Z6Myfuncv

通过这个测试，由于 c++中需要支持函数重载，所以 c 和 c++中对同一个函数经过编译后生成的函数名是不相同的，这就导致了一个问题，如果在 c++中调用一个使用 c 语言编写模块中的某个函数，那么 c++是根据 c++的名称修饰方式来查找并链接这个函数，那么就会发生链接错误，以上例，c++中调用 MyFunc 函数，在链接阶段会去找 Z6Myfuncv，结果是没有找到的，因为这个 MyFunc 函数是 c 语言编写的，生成的符号是 MyFunc。 那么如果我想在 c++调用 c 的函数怎么办？ extern "C"的主要作用就是为了实现 c++代码能够调用其他 c 语言代码。加上 extern "C" 后，这部分代码编译器按 c 语言的方式进行编译和链接，而不是按 c++的方式。 MyModule.h

```
#ifndef MYMODULE_H
#define MYMODULE_H

#include<stdio.h>

#if __cplusplus
extern "C"{
#endif

	void func1();
	int func2(int a,int b);

#if __cplusplus
}
#endif

#endif
```

MyModule.c

```
#include"MyModule.h"

void func1(){
	printf("hello world!");
}
int func2(int a, int b){
	return a + b;
}
```

TestExternC.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;

#if 0

	#ifdef __cplusplus
	extern "C" {
		#if 0
			void func1();
			int func2(int a, int b);
		#else
			#include"MyModule.h"
		#endif
	}

	#endif

#else
```

```
	extern "C" void func1();
	extern "C" int func2(int a, int b);

#endif

int main(){
	func1();
	cout << func2(10, 20) << endl;
	return EXIT_SUCCESS;
}
```

4 类和对象

4.1 类和对象的基本概念

4.1.1 C 和 C++中 struct 区别

c 语言 struct 只有变量 c++语言 struct 既有变量，也有函数

4.1.2 类的封装

我们编写程序的目的是为了解决现实中的问题，而这些问题的构成都是由各种事物组成，我们在计算机中要解决这种问题，首先要做就是要将这个问题的参与者：事和物抽象到计算机程序中，也就是用程序语言表示现实的事物。 那么现在问题是如何用程序语言来表示现实事物？现实世界的事物所具有的共性就是每个事物都具有自身的属性，一些自身具有的行为，所以如果我们能把事物的属性和行为表示出来，那么就可以抽象出来这个事物。 比如我们要表示人这个对象，在 c 语言中，我们可以这么表示:

```
typedef struct _Person{

char name[64];
int age;

}Person;
typedef struct _Aninal{
	char name[64];
	int age;
	int type; //动物种类
}Ainmal;

void PersonEat(Person* person){
	printf("%s 在吃人吃的饭!\n",person->name);
}
void AnimalEat(Ainmal* animal){
```

```
    printf("%s 在吃动物吃的饭!\n", animal->name);
}

int main(){

Person person;
strcpy(person.name, "小明");
person.age = 30;
AnimalEat(&person);

return EXIT_SUCCESS;

}
```

定义一个结构体用来表示一个对象所包含的属性，函数用来表示一个对象所具有的行为，这样我们就表示出来一个事物，在 c 语言中，行为和属性是分开的，也就是说吃饭这个属性不属于某类对象，而属于所有的共同的数据，所以不单单是 PeopleEat 可以调用 Person 数据，AnimalEat 也可以调用 Person 数据，那么万一调用错误，将会导致问题发生。 从这个案例我们应该可以体会到，属性和行为应该放在一起，一起表示一个具有属性和行为的对象。 假如某对象的某项属性不想被外界获知，比如说漂亮女孩的年龄不想被其他人知道，那么年龄这条属性应该作为女孩自己知道的属性；或者女孩的某些行为不想让外界知道，只需要自己知道就可以。那么这种情况下，封装应该再提供一种机制能够给属性和行为的访问权限控制住。 所以说封装特性包含两个方面，一个是属性和变量合成一个整体，一个是给属性和函数增加访问权限。

封装

1. 把变量（属性）和函数（操作）合成一个整体，封装在一个类中
2. 对变量和函数进行访问控制

   访问权限
3. 在类的内部(作用域范围内)，没有访问权限之分，所有成员可以相互访问
4. 在类的外部(作用域范围外)，访问权限才有意义：public，private，protected
5. 在类的外部，只有 public 修饰的成员才能被访问，在没有涉及继承与派生时， private 和 protected 是同等级的，外部不允许访问

访问属性	属性	对象内部	对象外部
public	公有	可访问	可访问
protected	保护	可访问	不可访问
private	私有	可访问	不可访问

/封装两层含义

```
//1. 属性和行为合成一个整体
//2. 访问控制，现实事物本身有些属性和行为是不对外开放
class Person{
//人具有的行为(函数)
public:
	void Dese(){ cout << "我有钱，年轻，个子又高，就爱嘚瑟!" << endl;}
//人的属性(变量)
public:
	int mTall; //多高，可以让外人知道
protected:
	int mMoney; // 有多少钱,只能儿子孙子知道
private:
	int mAge; //年龄，不想让外人知道
};

int main(){

	Person p;
	p.mTall = 220;
	//p.mMoney 保护成员外部无法访问
	//p.mAge 私有成员外部无法访问
	p.Dese();

	return EXIT_SUCCESS;
}
```

[struct 和 class 的区别?] class 默认访问权限为 private,struct 默认访问权限为 public.

```
class A{
	int mAge;
};
struct B{
	int mAge;
};

void test(){
	A a;
```

```
    B b;
    //a.mAge; //无法访问私有成员
    b.mAge; //可正常外部访问
}
```

4.1.3 将成员变量设置为 private

1. 可赋予客户端访问数据的一致性。 如果成员变量不是 public，客户端唯一能够访问对象的方法就是通过成员函数。如果类中所有 public 权限的成员都是函数，客户在访问类成员时只会默认访问函数，不需要考虑访问的成员需不需要添加(),这就省下了许多搔首弄耳的时间。

2. 可细微划分访问控制。 使用成员函数可使得我们对变量的控制处理更加精细。如果我们让所有的成员变量为 public，每个人都可以读写它。如果我们设置为 private，我们可以实现“不准访问”、“只读访问”、“读写访问”，甚至你可以写出“只写访问”。

```
class AccessLevels{
public:
    //对只读属性进行只读访问
    int getReadOnly(){ return readOnly; }
    //对读写属性进行读写访问
    void setReadWrite(int val){ readWrite = val; }
    int getReadWrite(){ return readWrite; }
    //对只写属性进行只写访问
    void setWriteOnly(int val){ writeOnly = val; }
private:
    int readOnly; //对外只读访问
    int noAccess; //外部不可访问
    int readWrite; //读写访问
    int writeOnly; //只写访问
};
```

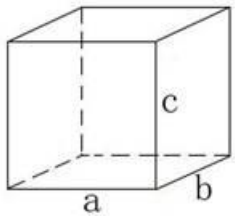
4.1.4 课堂练习

请设计一个 Person 类，Person 类具有 name 和 age 属性，提供初始化函数(Init)，并提供对 name 和 age 的读写函数(set，get)，但必须确保 age 的赋值在有效范围内(0-100),超出有效范围，则拒绝赋值，并提供方法输出姓名和年龄.(10 分钟)

4.2 面向对象程序设计案例

4.2.1 设计立方体类

设计立方体类(Cube)，求出立方体的面积(2*a*b + 2*a*c + 2*b*c)和体积(a * b * c)，分别用全局函数和成员函数判断两个立方体是否相等。




```
//立方体类
class Cub{
public:
	void setL(int l){ mL = l; }
	void setW(int w){ mW = w; }
	void setH(int h){ mH = h; }
	int getL(){ return mL; }
	int getW(){ return mW; }
	int getH(){ return mH; }
	//立方体面积
	int caculateS(){ return (mL*mW + mL*mH + mW*mH) * 2; }
	//立方体体积
	int caculateV(){ return mL * mW * mH; }
	//成员方法
	bool CubCompare(Cub& c){
		if (getL() == c.getL() && getW() == c.getW() && getH() == c.
getH()){
			return true;
		}
		return false;
	}
private:
	int mL; //长
	int mW; //宽
	int mH; //高
};

//比较两个立方体是否相等
bool CubCompare(Cub& c1, Cub& c2){
	if (c1.getL() == c2.getL() && c1.getW() == c2.getW() && c1.getH()
 == c2.getH()){
		return true;
	}
	return false;
}

void test(){
	Cub c1, c2;
	c1.setL(10);
	c1.setW(20);
```

```
	c1.setH(30);

	c2.setL(20);
	c2.setW(20);
	c2.setH(30);

	cout << "c1 面积:" << c1.caculateS() << " 体积:" << c1.caculateV()
 << endl;
	cout << "c2 面积:" << c2.caculateS() << " 体积:" << c2.caculateV()
 << endl;

	//比较两个立方体是否相等
	if (CubCompare(c1, c2)){
		cout << "c1 和 c2 相等!" << endl;
	}
	else{
		cout << "c1 和 c2 不相等!" << endl;
	}

	if (c1.CubCompare(c2)){
		cout << "c1 和 c2 相等!" << endl;
	}
	else{
		cout << "c1 和 c2 不相等!" << endl;
	}
}
```

4.2.2 点和圆的关系

设计一个圆形类（AdvCircle），和一个点类（Point），计算点和圆的关系。 假如圆心坐标为 x0, y0, 半径为 r，点的坐标为 x1, y1： 1）点在圆上：(x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) == r*r 2）点在圆内：(x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) < r*r 3）点在圆外：(x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) > r*r

```
//点类
class Point{
public:
	void setX(int x){ mX = x; }
	void setY(int y){ mY = y; }
	int getX(){ return mX; }
	int getY(){ return mY; }
private:
	int mX;
	int mY;
};

//圆类
class Circle{
```

```
public:
	void setP(int x,int y){
		mP.setX(x);
		mP.setY(y);
	}
	void setR(int r){ mR = r; }
	Point& getP(){ return mP; }
	int getR(){ return mR; }
	//判断点和圆的关系
	void IsPointInCircle(Point& point){
		int distance = (point.getX() - mP.getX()) * (point.getX() -
 mP.getX()) + (point.getY() - mP.getY()) * (point.getY() - mP.getY());
		int radius = mR * mR;
		if (distance < radius){
			cout << "Point(" << point.getX() << "," << point.getY
() << ")在圆内!" << endl;
		}
		else if (distance > radius){
			cout << "Point(" << point.getX() << "," << point.getY
() << ")在圆外!" << endl;
		}
		else{
			cout << "Point(" << point.getX() << "," << point.getY
() << ")在圆上!" << endl;
		}
	}
private:
	Point mP; //圆心
	int mR; //半径
};

void test(){
	//实例化圆对象
	Circle circle;
	circle.setP(20, 20);
	circle.setR(5);
	//实例化点对象
	Point point;
	point.setX(25);
	point.setY(20);

	circle.IsPointInCircle(point);
}
```

4.3 对象的构造和析构

4.3.1 初始化和清理

我们大家在购买一台电脑或者手机，或者其他的产品，这些产品都有一个初始设置，也就是这些产品对被创建的时候会有一个基础属性值。那么随着我们使用手机和电脑的时间越来越久，那么电脑和手机会慢慢被我们手动创建很多文件数据，某一天我们不用手机或电脑了，那么我们应该将电脑或手机中我们增加的数据删除掉，保护自己的信息数据。 从这样的过程中，我们体会一下，所有的事物在起初的时候都应该有个初始状态，当这个事物完成其使命时，应该及时清除外界作用于上面的一些信息数据。 那么我们 c++中 OO 思想也是来源于现实，是对现实事物的抽象模拟，具体来说，当我们创建对象的时候,这个对象应该有一个初始状态，当对象销毁之前应该销毁自己创建的一些数据。 对象的初始化和清理也是两个非常重要的安全问题，一个对象或者变量没有初始时，对其使用后果是未知，同样的使用完一个变量，没有及时清理，也会造成一定的安全问题。c++为了给我们提供这种问题的解决方案，构造函数和析构函数，这两个函数将会被编译器自动调用，完成对象初始化和对象清理工作。 无论你是否喜欢，对象的初始化和清理工作是编译器强制我们要做的事情，即使你不提供初始化操作和清理操作，编译器也会给你增加默认的操作，只是这个默认初始化操作不会做任何事，所以编写类就应该顺便提供初始化函数。 为什么初始化操作是自动调用而不是手动调用？既然是必须操作，那么自动调用会更好，如果靠程序员自觉，那么就会存在遗漏初始化的情况出现。

4.3.2 构造函数和析构函数

构造函数主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。 析构函数主要用于对象销毁前系统自动调用，执行一些清理工作。 构造函数语法： 构造函数函数名和类名相同，没有返回值，不能有 void，但可以有参数。 ClassName(){}

析构函数语法： 析构函数函数名是在类名前面加”~”组成,没有返回值，不能有 void,不能有参数，不能重载。 ~ClassName(){}

```
class Person{
public:
	Person(){
		cout << "构造函数调用!" << endl;
		pName = (char*)malloc(sizeof("John"));
		strcpy(pName, "John");
		mTall = 150;
		mMoney = 100;
	}
	~Person(){
		cout << "析构函数调用!" << endl;
		if (pName != NULL){
```

```
                free(pName);
                pName = NULL;
            }
        }
public:
        char* pName;
        int mTall;
        int mMoney;
};

void test(){
        Person person;
        cout << person.pName << person.mTall << person.mMoney << endl;
}
```

4.3.3 构造函数的分类及调用

按参数类型：分为无参构造函数和有参构造函数 按类型分类：普通构造函数和拷贝构造函数(复制构造函数)

```
class Person{
public:
        Person(){
                cout << "no param constructor!" << endl;
                mAge = 0;
        }
        //有参构造函数
        Person(int age){
                cout << "1 param constructor!" << endl;
                mAge = age;
        }
        //拷贝构造函数(复制构造函数) 使用另一个对象初始化本对象
        Person(const Person& person){
                cout << "copy constructor!" << endl;
                mAge = person.mAge;
        }
        //打印年龄
        void PrintPerson(){
                cout << "Age:" << mAge << endl;
        }
private:
        int mAge;
};
//1. 无参构造调用方式
void test01(){

        //调用无参构造函数
        Person person1;
        person1.PrintPerson();
```

```
	//无参构造函数错误调用方式
	//Person person2();
	//person2.PrintPerson();
}
//2. 调用有参构造函数
void test02(){

	//第一种 括号法，最常用
	Person person01(100);
	person01.PrintPerson();

	//调用拷贝构造函数
	Person person02(person01);
	person02.PrintPerson();

	//第二种 匿名对象(显示调用构造函数)
	Person(200); //匿名对象，没有名字的对象

	Person person03 = Person(300);
	person03.PrintPerson();

	//注意: 使用匿名对象初始化判断调用哪一个构造函数，要看匿名对象的参数类型
	Person person06(Person(400)); //等价于 Person person06 = Person(400);
	person06.PrintPerson();

	//第三种 =号法 隐式转换
	Person person04 = 100; //Person person04 =  Person(100)
	person04.PrintPerson();

	//调用拷贝构造
	Person person05 = person04; //Person person05 =  Person(person04)
	person05.PrintPerson();
}
```

b 为 A 的实例化对象,A a = A(b) 和 A(b)的区别？ 当 A(b) 有变量来接的时候，那么编译器认为他是一个匿名对象，当没有变量来接的时候，编译器认为你 A(b) 等价于 A b.

注意:不能调用拷贝构造函数去初始化匿名对象,也就是说以下代码不正确:

```
class Teacher{
public:
	Teacher(){
		cout << "默认构造函数!" << endl;
	}
```

```
	Teacher(const Teacher& teacher){
		cout << "拷贝构造函数!" << endl;
	}
public:
	int mAge;
};
void test(){

	Teacher t1;
	//error C2086:“Teacher t1”: 重定义
	Teacher(t1);  //此时等价于 Teacher t1;
}
```

4.3.4 拷贝构造函数的调用时机

对象以值传递的方式传给函数参数 函数局部对象以值传递的方式从函数返回(vs debug 模式下调用一次拷贝构造，qt 不调用任何构造) 用一个对象初始化另一个对象

```
class Person{
public:
	Person(){
		cout << "no param contructor!" << endl;
		mAge = 10;
	}
	Person(int age){
		cout << "param constructor!" << endl;
		mAge = age;
	}
	Person(const Person& person){
		cout << "copy constructor!" << endl;
		mAge = person.mAge;
	}
	~Person(){
		cout << "destructor!" << endl;
	}
public:
	int mAge;
};
//1. 旧对象初始化新对象
void test01(){

	Person p(10);
	Person p1(p);
	Person p2 = Person(p);
	Person p3 = p; // 相当于Person p2 = Person(p);
}

//2. 传递的参数是普通对象，函数参数也是普通对象，传递将会调用拷贝构造
```

```
void doBussiness(Person p){}

void test02(){
      Person p(10);
      doBussiness(p);
}

//3. 函数返回局部对象
Person MyBusiness(){
      Person p(10);
      cout << "局部 p:" << (int*)&p << endl;
      return p;
}
void test03(){
      //vs release、qt 下没有调用拷贝构造函数
      //vs debug 下调用一次拷贝构造函数
      Person p = MyBusiness();
      cout << "局部 p:" << (int*)&p << endl;
}
```

[Test03 结果说明:] 编译器存在一种对返回值的优化技术,RVO(Return Value Optimization).在 vs debug 模式下并没有进行这种优化，所以函数 MyBusiness 中创建 p 对象，调用了一次构造函数，当编译器发现你要返回这个局部的对象时，编译器通过调用拷贝构造生成一个临时 Person 对象返回，然后调用 p 的析构函数。 我们从常理来分析的话，这个匿名对象和这个局部的 p 对象是相同的两个对象，那么如果能直接返回 p 对象，就会省去一个拷贝构造和一个析构函数的开销，在程序中一个对象的拷贝也是非常耗时的，如果减少这种拷贝和析构的次数，那么从另一个角度来说，也是编译器对程序执行效率上进行了优化。 所以在这里，编译器偷偷帮我们做了一层优化： 当我们这样去调用: Person p = MyBusiness(); 编译器偷偷将我们的代码更改为:

```
void MyBussiness(Person& _result){
       _result.X:X(); //调用 Person 默认拷贝构造函数
       //.....对_result 进行处理
       return;
   }
int main(){
   Person p; //这里只分配空间，不初始化
   MyBussiness(p);
}
```

4.3.5 构造函数调用规则

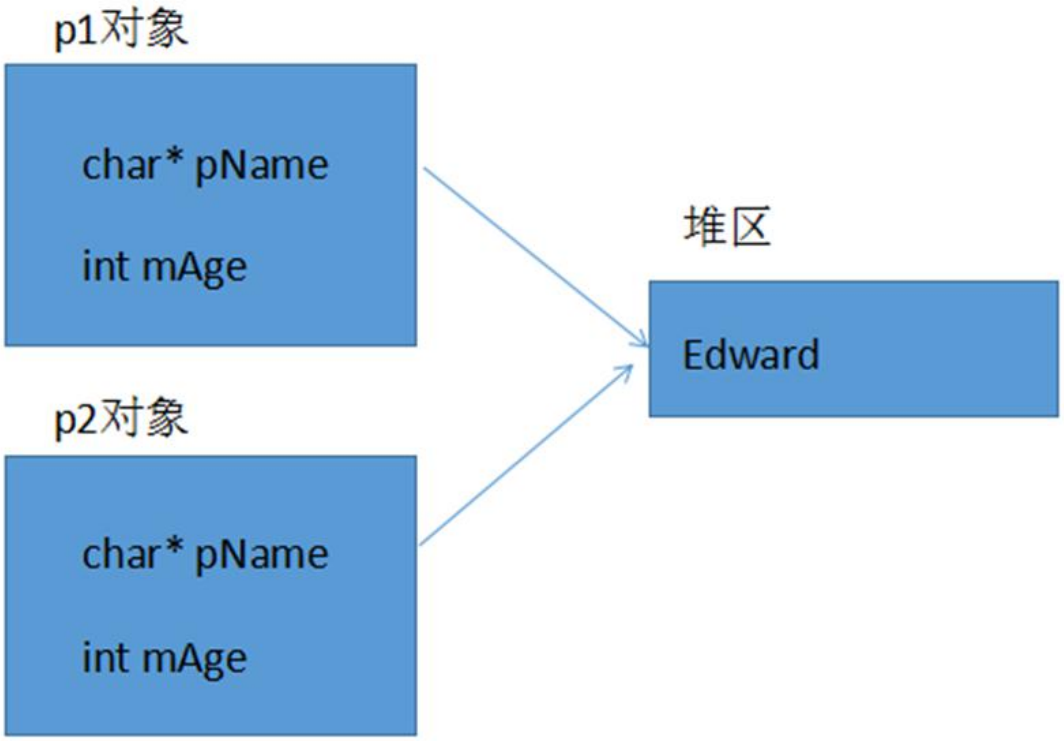
默认情况下，c++编译器至少为我们写的类增加 3 个函数 1．默认构造函数(无参，函数体为空) 2．默认析构函数(无参，函数体为空) 3．默认拷贝构造函数，对类中非静态成员属性简单值拷贝 如果用户定义拷贝构造函数，c++不会再提供任何默认

构造函数 如果用户定义了普通构造(非拷贝)，c++不在提供默认无参构造，但是会提供默认拷贝构造

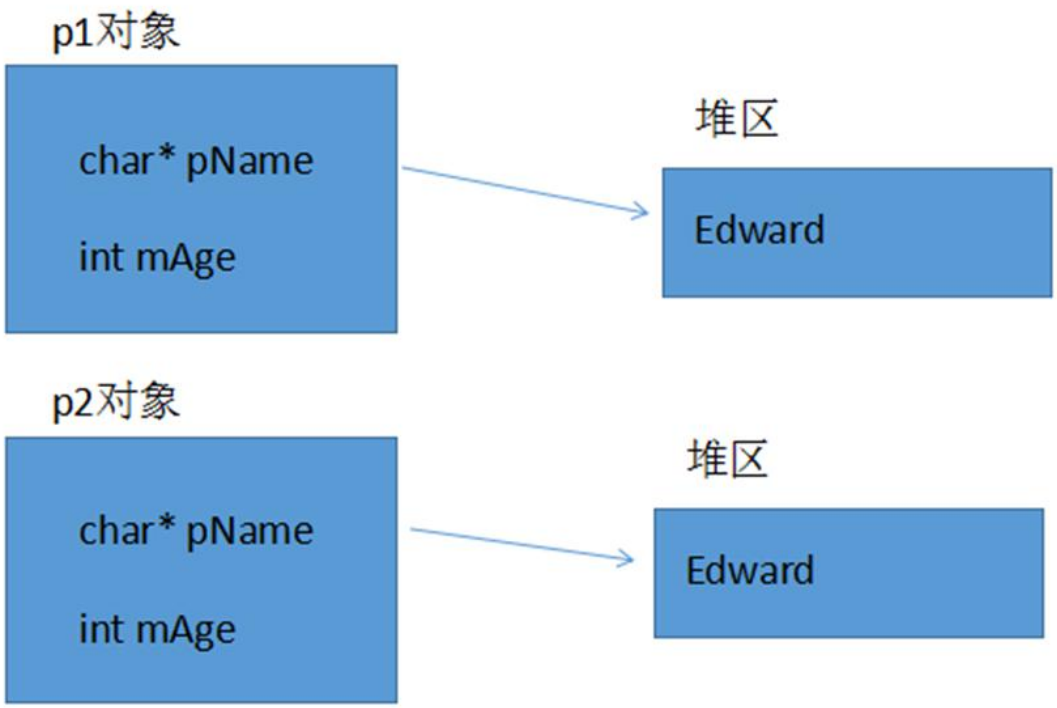
4.3.6 深拷贝和浅拷贝

4.3.6.1 浅拷贝

同一类型的对象之间可以赋值，使得两个对象的成员变量的值相同，两个对象仍然是独立的两个对象，这种情况被称为浅拷贝. 一般情况下，浅拷贝没有任何副作用，但是当类中有指针，并且指针指向动态分配的内存空间，析构函数做了动态内存释放的处理，会导致内存问题。



4.3.6.2 深拷贝 当类中有指针，并且此指针有动态分配空间，析构函数做了释放处理，往往需要自定义拷贝构造函数，自行给指针动态分配空间，深拷贝。




```
class Person{
public:
	Person(char* name,int age){
		pName = (char*)malloc(strlen(name) + 1);
		strcpy(pName,name);
		mAge = age;
	}
	//增加拷贝构造函数
	Person(const Person& person){
		pName = (char*)malloc(strlen(person.pName) + 1);
		strcpy(pName, person.pName);
		mAge = person.mAge;
	}
	~Person(){
		if (pName != NULL){
			free(pName);
		}
	}
private:
	char* pName;
	int mAge;
};

void test(){
	Person p1("Edward",30);
	//用对象p1 初始化对象p2,调用c++提供的默认拷贝构造函数
```

```
	Person p2 = p1;
}
```

4.3.7 多个对象构造和析构

4.3.7.1 初始化列表

构造函数和其他函数不同，除了有名字，参数列表，函数体之外还有初始化列表。初始化列表简单使用:

```
class Person{
public:
#if 0
	//传统方式初始化
	Person(int a,int b,int c){
		mA = a;
		mB = b;
		mC = c;
	}
#endif
	//初始化列表方式初始化
	Person(int a, int b, int c):mA(a),mB(b),mC(c){}
	void PrintPerson(){
		cout << "mA:" << mA << endl;
		cout << "mB:" << mB << endl;
		cout << "mC:" << mC << endl;
	}
private:
	int mA;
	int mB;
	int mC;
};
```

注意：初始化成员列表(参数列表)只能在构造函数使用。

4.3.7.2 类对象作为成员

在类中定义的数据成员一般都是基本的数据类型。但是类中的成员也可以是对象，叫做对象成员。 C++中对对象的初始化是非常重要的操作，当创建一个对象的时候，c++编译器必须确保调用了所有子对象的构造函数。如果所有的子对象有默认构造函数，编译器可以自动调用他们。但是如果子对象没有默认的构造函数，或者想指定调用某个构造函数怎么办？ 那么是否可以在类的构造函数直接调用子类的属性完成初始化呢？但是如果子类的成员属性是私有的，我们是没有办法访问并完成初始化的。 解决办法非常简单：对于子类调用构造函数，c++为此提供了专门的语法，即构造函数初始化列表。 当调用构造函数时，首先按各对象成员在类定义中的顺序（和参数列表的顺序无关）依次调用它们的构造函数，对这些对象初始化，最后再调用本身的函数体。也就是说，先调用对象成员的构造函数，再调用本身的构造函数。 析构函数和构造函数调用顺序相反，先构造，后析构。

```
//汽车类
class Car{
public:
	Car(){
		cout << "Car 默认构造函数!" << endl;
		mName = "大众汽车";
	}
	Car(string name){
		cout << "Car 带参数构造函数!" << endl;
		mName = name;
	}
	~Car(){
		cout << "Car 析构函数!" << endl;
	}
public:
	string mName;
};

//拖拉机
class Tractor{
public:
	Tractor(){
		cout << "Tractor 默认构造函数!" << endl;
		mName = "爬土坡专用拖拉机";
	}
	Tractor(string name){
		cout << "Tractor 带参数构造函数!" << endl;
		mName = name;
	}
	~Tractor(){
		cout << "Tractor 析构函数!" << endl;
	}
public:
	string mName;
};

//人类
class Person{
public:
#if 1
	//类mCar 不存在合适的构造函数
	Person(string name){
		mName = name;
	}
#else
	//初始化列表可以指定调用构造函数
	Person(string carName, string tracName, string name) : mTractor(t
racName), mCar(carName), mName(name){
```

```
            cout << "Person 构造函数!" << endl;
        }
#endif

        void GoWorkByCar(){
            cout << mName << "开着" << mCar.mName << "去上班!" << endl;
        }
        void GoWorkByTractor(){
            cout << mName << "开着" << mTractor.mName << "去上班!" << en
dl;
        }
        ~Person(){
            cout << "Person 析构函数!" << endl;
        }
private:
        string mName;
        Car mCar;
        Tractor mTractor;
};

void test(){
        //Person person("宝马", "东风拖拉机", "赵四");
        Person person("刘能");
        person.GoWorkByCar();
        person.GoWorkByTractor();
}
```

4.3.8 explicit 关键字

c++提供了关键字 explicit，禁止通过构造函数进行的隐式转换。声明为 explicit 的构造函数不能在隐式转换中使用。

[explicit 注意] explicit 用于修饰构造函数,防止隐式转化。 是针对单参数的构造函数(或者除了第一个参数外其余参数都有默认值的多参构造)而言。

```
class MyString{
public:
        explicit MyString(int n){
            cout << "MyString(int n)!" << endl;
        }
        MyString(const char* str){
            cout << "MyString(const char* str)" << endl;
        }
};

int main(){

        //给字符串赋值？还是初始化？
        //MyString str1 = 1;
```

```
	MyString str2(10);

	//寓意非常明确，给字符串赋值
	MyString str3 = "abcd";
	MyString str4("abcd");

	return EXIT_SUCCESS;
}
```

4.3.9 动态对象创建

当我们创建数组的时候，总是需要提前预定数组的长度，然后编译器分配预定长度的数组空间，在使用数组的时，会有这样的问题，数组也许空间太大了，浪费空间，也许空间不足，所以对于数组来讲，如果能根据需要来分配空间大小再好不过。 所以动态的意思意味着不确定性。 为了解决这个普遍的编程问题，在运行中可以创建和销毁对象是最基本的要求。当然 c 早就提供了动态内存分配（dynamic memory allocation）,函数 malloc 和 free 可以在运行时从堆中分配存储单元。 然而这些函数在 c++中不能很好的运行，因为它不能帮我们完成对象的初始化工作。

4.3.9.1 对象创建

当创建一个 c++对象时会发生两件事:

1. 为对象分配内存
2. 调用构造函数来初始化那块内存 第一步我们能保证实现，需要我们确保第二步一定能发生。c++强迫我们这么做是因为使用未初始化的对象是程序出错的一个重要原因。 4.3.9.2 C 动态分配内存方法 为了在运行时动态分配内存，c 在他的标准库中提供了一些函数,malloc 以及它的变种 calloc 和 realloc,释放内存的 free,这些函数是有效的、但是原始的，需要程序员理解和小心使用。为了使用 c 的动态内存分配函数在堆上创建一个类的实例，我们必须这样做:

```
class Person{
public:
	Person(){
		mAge = 20;
		pName = (char*)malloc(strlen("john")+1);
		strcpy(pName, "john");
	}
	void Init(){
		mAge = 20;
		pName = (char*)malloc(strlen("john")+1);
		strcpy(pName, "john");
	}
	void Clean(){
		if (pName != NULL){
			free(pName);
		}
```

```
	}
public:
	int mAge;
	char* pName;
};
int main(){

	//分配内存
	Person* person = (Person*)malloc(sizeof(Person));
	if(person == NULL){
		return 0;
	}
	//调用初始化函数
	person->Init();
	//清理对象
	person->Clean();
	//释放person 对象
	free(person);

	return EXIT_SUCCESS;
}
```

问题： 1) 程序员必须确定对象的长度。 2) malloc 返回一个 void *指针，c++不允许将 void* 赋值给其他任何指针，必须强转。 3) malloc 可能申请内存失败，所以必须判断返回值来确保内存分配成功。 4) 用户在使用对象之前必须记住对他初始化，构造函数不能显示调用初始化(构造函数是由编译器调用)，用户有可能忘记调用初始化函数。

c 的动态内存分配函数太复杂，容易令人混淆，是不可接受的，c++中我们推荐使用运算符 new 和 delete.

4.3.9.3 new operator

C++中解决动态内存分配的方案是把创建一个对象所需要的操作都结合在一个称为 new 的运算符里。当用 new 创建一个对象时，它就在堆里为对象分配内存并调用构造函数完成初始化。

```
Person* person = new Person;
相当于:
Person* person = (Person*)malloc(sizeof(Person));
	if(person == NULL){
		return 0;
	}
person->Init(); 构造函数
```

New 操作符能确定在调用构造函数初始化之前内存分配是成功的，所有不用显式确定调用是否成功。 现在我们发现在堆里创建对象的过程变得简单了，只需要一

个简单的表达式，它带有内置的长度计算、类型转换和安全检查。这样在堆创建一个对象和在栈里创建对象一样简单。

4.3.9.4 delete operator

new 表达式的反面是 delete 表达式。delete 表达式先调用析构函数，然后释放内存。正如 new 表达式返回一个指向对象的指针一样，delete 需要一个对象的地址。 delete 只适用于由 new 创建的对象。 如果使用一个由 malloc 或者 calloc 或者 realloc 创建的对象使用 delete,这个行为是未定义的。因为大多数 new 和 delete 的实现机制都使用了 malloc 和 free,所以很可能没有调用析构函数就释放了内存。如果正在删除的对象的指针是 NULL,将不发生任何事，因此建议在删除指针后，立即把指针赋值为 NULL，以免对它删除两次，对一些对象删除两次可能会产生某些问题。

```
class Person{
public:
	Person(){
		cout << "无参构造函数!" << endl;
		pName = (char*)malloc(strlen("undefined") + 1);
		strcpy(pName, "undefined");
		mAge = 0;
	}
	Person(char* name, int age){
		cout << "有参构造函数!" << endl;
		pName = (char*)malloc(strlen(name) + 1);
		strcpy(pName, name);
		mAge = age;
	}
	void ShowPerson(){
		cout << "Name:" << pName << " Age:" << mAge << endl;
	}
	~Person(){
		cout << "析构函数!" << endl;
		if (pName != NULL){
			delete pName;
			pName = NULL;
		}
	}
public:
	char* pName;
	int mAge;
};

void test(){
	Person* person1 = new Person;
	Person* person2 = new Person("John",33);

	person1->ShowPerson();
```

```
	person2->ShowPerson();

	delete person1;
	delete person2;
}
```

4.3.9.5 用于数组的 new 和 delete

使用 new 和 delete 在堆上创建数组非常容易。

```
//创建字符数组
char* pStr = new char[100];
//创建整型数组
int* pArr1 = new int[100];
//创建整型数组并初始化
int* pArr2 = new int[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

//释放数组内存
delete[] pStr;
delete[] pArr1;
delete[] pArr2;
```

当创建一个对象数组的时候，必须对数组中的每一个对象调用构造函数，除了在栈上可以聚合初始化，必须提供一个默认的构造函数。

```
class Person{
public:
	Person(){
		pName = (char*)malloc(strlen("undefined") + 1);
		strcpy(pName, "undefined");
		mAge = 0;
	}
	Person(char* name, int age){
		pName = (char*)malloc(sizeof(name));
		strcpy(pName, name);
		mAge = age;
	}
	~Person(){
		if (pName != NULL){
			delete pName;
		}
	}
public:
	char* pName;
	int mAge;
};

void test(){
	//栈聚合初始化
	Person person[] = { Person("john", 20), Person("Smith", 22) };
```

```
	cout << person[1].pName << endl;
    //创建堆上对象数组必须提供构造函数
	Person* workers = new Person[20];
}
```

4.3.9.6 delete void*可能会出错

如果对一个 void*指针执行 delete 操作，这将可能成为一个程序错误，除非指针指向的内容是非常简单的，因为它将不执行析构函数.以下代码未调用析构函数，导致可用内存减少。

```
class Person{
public:
	Person(char* name, int age){
		pName = (char*)malloc(sizeof(name));
		strcpy(pName,name);
		mAge = age;
	}
	~Person(){
		if (pName != NULL){
			delete pName;
		}
	}
public:
	char* pName;
	int mAge;
};

void test(){
	void* person = new Person("john",20);
	delete person;
}
```

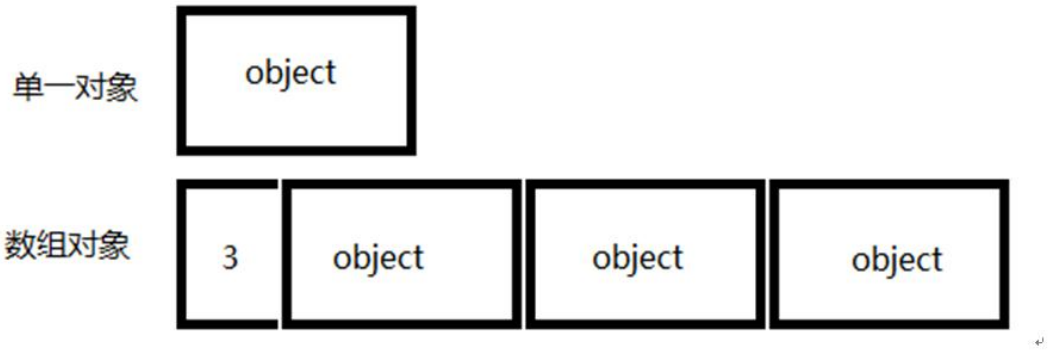
问题：malloc、free 和 new、delete 可以混搭使用吗？也就是说 malloc 分配的内存，可以调用 delete 吗？通过 new 创建的对象，可以调用 free 来释放吗？

4.3.9.7 使用 new 和 delete 采用相同形式

```
Person* person = new Person[10];
	delete person;
```

以上代码有什么问题吗？(vs 下直接中断、qt 下析构函数调用一次) 使用了 new 也搭配使用了 delete，问题在于 Person 有 10 个对象，那么其他 9 个对象可能没有调用析构函数，也就是说其他 9 个对象可能删除不完全，因为它们的析构函数没有被调用。 我们现在清楚使用 new 的时候发生了两件事: 一、分配内存；二、调用构造函数，那么调用 delete 的时候也有两件事：一、析构函数；二、释放内存。 那么刚才我们那段代码最大的问题在于：person 指针指向的内存中到底有多少个对象，因为这个决定应该有多少个析构函数应该被调用。换句话说，person 指针指向的是一个单一的对象还是一个数组对象，由于单一对象和数组对象的内存

布局是不同的。更明确的说，数组所用的内存通常还包括“数组大小记录”，使得 delete 的时候知道应该调用几次析构函数。单一对象的话就没有这个记录。单一对象和数组对象的内存布局可理解为下图:



本图只是为了说明，编译器不一定如此实现，但是很多编译器是这样做的。 当我们使用一个 delete 的时候，我们必须让 delete 知道指针指向的内存空间中是否存在一个“数组大小记录”的办法就是我们告诉它。当我们使用 delete[]，那么 delete 就知道是一个对象数组，从而清楚应该调用几次析构函数。 结论: 如果在 new 表达式中使用[]，必须在相应的 delete 表达式中也使用[].如果在 new 表达式中不使用[], 一定不要在相应的 delete 表达式中使用[].

4.3.10 静态成员

在类定义中，它的成员（包括成员变量和成员函数），这些成员可以用关键字 static 声明为静态的，称为静态成员。 不管这个类创建了多少个对象，静态成员只有一个拷贝，这个拷贝被所有属于这个类的对象共享。

4.3.10.1 静态成员变量

在一个类中，若将一个成员变量声明为 static，这种成员称为静态成员变量。与一般的数据成员不同，无论建立了多少个对象，都只有一个静态数据的拷贝。静态成员变量，属于某个类，所有对象共享。 静态变量，是在编译阶段就分配空间，对象还没有创建时，就已经分配空间。

静态成员变量必须在类中声明，在类外定义。 静态数据成员不属于某个对象，在为对象分配空间中不包括静态成员所占空间。 静态数据成员可以通过类名或者对象名来引用。

```
class Person{
public:
	//类的静态成员属性
	static int sNum;
private:
	static int sOther;
```

```
};

//类外初始化，初始化时不加static
int Person::sNum = 0;
int Person::sOther = 0;
int main(){


	//1. 通过类名直接访问
	Person::sNum = 100;
	cout << "Person::sNum:" << Person::sNum << endl;

	//2. 通过对象访问
	Person p1, p2;
	p1.sNum = 200;

	cout << "p1.sNum:" << p1.sNum << endl;
	cout << "p2.sNum:" << p2.sNum << endl;

	//3. 静态成员也有访问权限，类外不能访问私有成员
	//cout << "Person::sOther:" << Person::sOther << endl;
	Person p3;
	//cout << "p3.sOther:" << p3.sOther << endl;

	system("pause");
	return EXIT_SUCCESS;
}
```

4.3.10.2 静态成员函数

在类定义中，前面有 static 说明的成员函数称为静态成员函数。静态成员函数使用方式和静态变量一样，同样在对象没有创建前，即可通过类名调用。静态成员函数主要为了访问静态变量，但是，不能访问普通成员变量。 静态成员函数的意义，不在于信息共享，数据沟通，而在于管理静态数据成员，完成对静态数据成员的封装。

静态成员函数只能访问静态变量，不能访问普通成员变量 静态成员函数的使用和静态成员变量一样 静态成员函数也有访问权限 普通成员函数可访问静态成员变量、也可以访问非经常成员变量

```
class Person{
public:
	//普通成员函数可以访问static 和non-static 成员属性
	void changeParam1(int param){
		mParam = param;
		sNum = param;
	}
	//静态成员函数只能访问static 成员属性
```

```
	static void changeParam2(int param){
		//mParam = param; //无法访问
		sNum = param;
	}
private:
	static void changeParam3(int param){
		//mParam = param; //无法访问
		sNum = param;
	}
public:
	int mParam;
	static int sNum;
};

//静态成员属性类外初始化
int Person::sNum = 0;

int main(){

	//1. 类名直接调用
	Person::changeParam2(100);

	//2. 通过对象调用
	Person p;
	p.changeParam2(200);

	//3. 静态成员函数也有访问权限
	//Person::changeParam3(100); //类外无法访问私有静态成员函数
	//Person p1;
	//p1.changeParam3(200);
	return EXIT_SUCCESS;
}
```

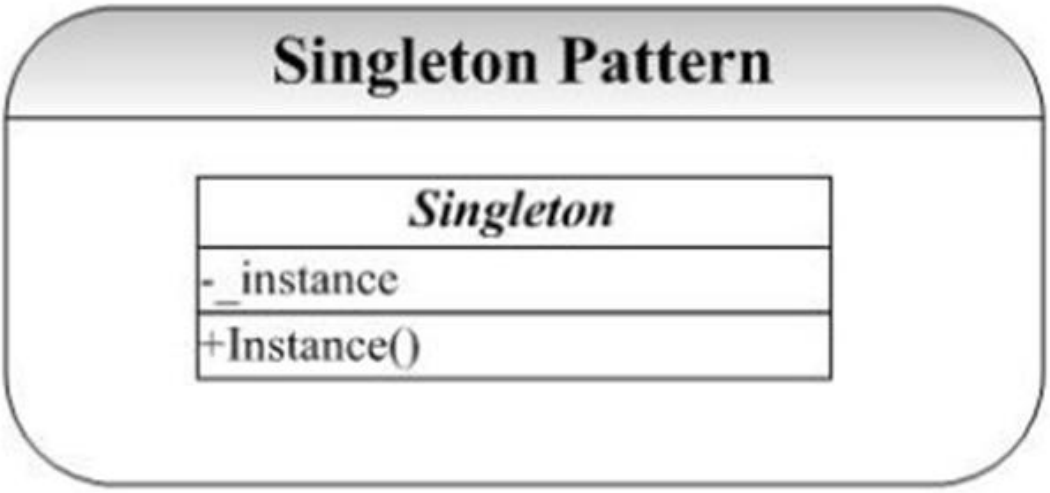
4.3.10.3 const 静态成员属性

如果一个类的成员，既要实现共享，又要实现不可改变，那就用 static const 修饰。定义静态 const 数据成员时，最好在类内部初始化。

```
class Person{
public:
	//static const int mShare = 10;
	const static int mShare = 10; //只读区，不可修改
};
int main(){

	cout << Person::mShare << endl;
	//Person::mShare = 20;
```

```
	return EXIT_SUCCESS;
}
```

4.3.10.4 静态成员实现单例模式 单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。



Singleton（单例）：在单例类的内部实现只生成一个实例，同时它提供一个静态的 getInstance()工厂方法，让客户可以访问它的唯一实例；为了防止在外部对其实例化，将其默认构造函数和拷贝构造函数设计为私有；在单例类内部定义了一个 Singleton 类型的静态对象，作为外部共享的唯一实例。

用单例模式，模拟公司员工使用打印机场景，打印机可以打印员工要输出的内容，并且可以累积打印机使用次数。

```
class Printer{
public:
	static Printer* getInstance(){ return pPrinter;}
	void PrintText(string text){
		cout << "打印内容:" << text << endl;
		cout << "已打印次数:" << mTimes << endl;
		cout << "--------------" << endl;
		mTimes++;
	}
private:
	Printer(){ mTimes = 0; }
	Printer(const Printer&){}
private:
	static Printer* pPrinter;
```

```
	int mTimes;
};

Printer* Printer::pPrinter = new Printer;

void test(){
	Printer* printer = Printer::getInstance();
	printer->PrintText("离职报告!");
	printer->PrintText("入职合同!");
	printer->PrintText("提交代码!");
}
```

4.4 C++面向对象模型初探

4.4.1 成员变量和函数的存储

在 c 语言中，“分开来声明的，也就是说，语言本身并没有支持“数据”和“函数”之间的关联性我们把这种程序方法称为“程序性的”，由一组“分布在各个以功能为导航的函数中”的算法驱动，它们处理的是共同的外部数据。

c++实现了“封装”，那么数据(成员属性)和操作(成员函数)是什么样的呢？ “数据”和“处理数据的操作(函数)”是分开存储的。 c++中的非静态数据成员直接内含在类对象中，就像 c struct 一样。 成员函数(member function)虽然内含在 class 声明之内，却不出现在对象中。 每一个非内联成员函数(non-inline member function)只会诞生一份函数实例.

```
class MyClass01{
public:
	int mA;
};

class MyClass02{
public:
	int mA;
	static int sB;
};

class MyClass03{
public:
	void printMyClass(){
		cout << "hello world!" << endl;
	}
public:
	int mA;
	static int sB;
};

class MyClass04{
```

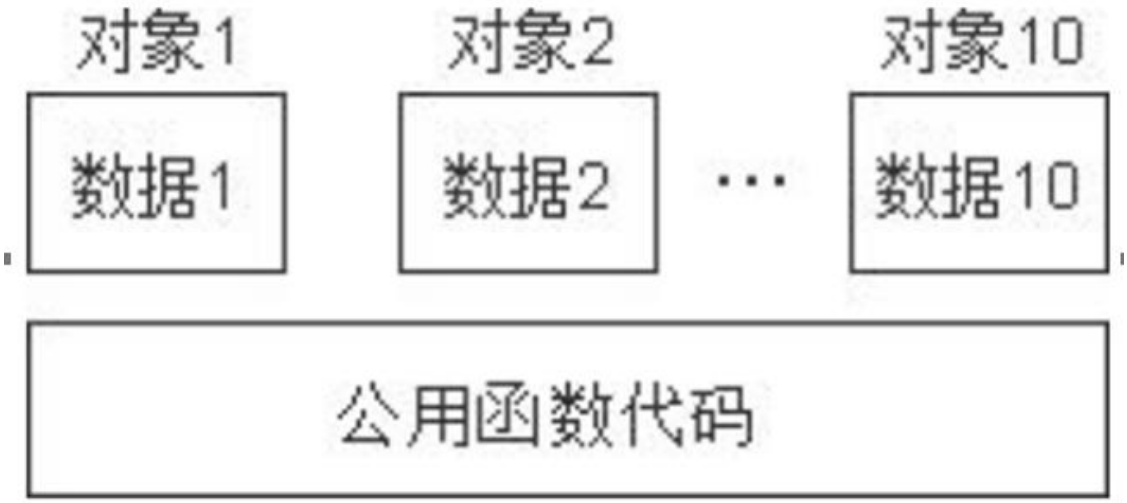
```
public:
	void printMyClass(){
		cout << "hello world!" << endl;
	}
	static void ShowMyClass(){
		cout << "hello world! " << endl;
	}
public:
	int mA;
	static int sB;
};

int main(){

	MyClass01 mclass01;
	MyClass02 mclass02;
	MyClass03 mclass03;
	MyClass04 mclass04;

	cout << "MyClass01:" << sizeof(mclass01) << endl; //4
	//静态数据成员并不保存在类对象中
	cout << "MyClass02:" << sizeof(mclass02) << endl; //4
	//非静态成员函数不保存在类对象中
	cout << "MyClass03:" << sizeof(mclass03) << endl; //4
	//静态成员函数也不保存在类对象中
	cout << "MyClass04:" << sizeof(mclass04) << endl; //4

	return EXIT_SUCCESS;
}
```

通过上面的案例，我们可以的得出：C++类对象中的变量和函数是分开存储。

4.4.2 this 指针

4.4.2.1 this 指针工作原理

通过上例我们知道，c++的数据和操作也是分开存储，并且每一个非内联成员函数(non-inline member function)只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码 那么问题是：这一块代码是如何区分那个对象调用自己的呢？



c++通过提供特殊的对象指针，this 指针，解决上述问题。this 指针指向被调用的成员函数所属的对象。 c++规定，this 指针是隐含在对象成员函数内的一种指针。当一个对象被创建后，它的每一个成员函数都含有一个系统自动生成的隐含指针 this，用以保存这个对象的地址，也就是说虽然我们没有写上 this 指针，编译器在编译的时候也是会加上的。因此 this 也称为“指向本对象的指针”，this 指针并不是对象的一部分，不会影响 sizeof(对象)的结果。 this 指针是 C++实现封装的一种机制，它将对象和该对象调用的成员函数连接在一起，在外部看来，每一个对象都拥有自己的函数成员。一般情况下，并不写 this，而是让系统进行默认设置。this 指针永远指向当前对象。

成员函数通过 this 指针即可知道操作的是那个对象的数据。This 指针是一种隐含指针，它隐含于每个类的非静态成员函数中。This 指针无需定义，直接使用即可。 注意：静态成员函数内部没有 this 指针，静态成员函数不能操作非静态成员变量。

c++编译器对普通成员函数的内部处理:

```
class Test{
public:
    Test(int a){
        m_a = a;
    }
    int getA(){
        return m_a;
    }
    static void print(){
        cout << "This is class Test!" << endl;
    }
private:
    int m_a; //4字节
};

Test a(10);

a.getA();

Test::print();
```

```
struct Test{
    int m_a;
};

void Test_initialize(Test* pThis, int i){
    pThis->m_a = i;
}

int Test_getA(Test* pThis){
    return pThis->m_a;
}

void Test_print(){
    cout << "hello world!" << endl;
}

Test a;
Test_initialize(&a,10);
Test_getA(&a);
Test_print();
```

4.4.2.2 this 指针的使用

当形参和成员变量同名时，可用 this 指针来区分 在类的非静态成员函数中返回对象本身，可使用 return *this.

```
class Person{
public:
    //1. 当形参名和成员变量名一样时，this 指针可用来区分
    Person(string name,int age){
        //name = name;
        //age = age; //输出错误
        this->name = name;
        this->age = age;
    }
    //2. 返回对象本身的引用
    //重载赋值操作符
    //其实也是两个参数，其中隐藏了一个 this 指针
    Person PersonPlusPerson(Person& person){
        string newname = this->name + person.name;
        int newage = this->age + person.age;
        Person newperson(newname, newage);
        return newperson;
    }
    void ShowPerson(){
        cout << "Name:" << name << " Age:" << age << endl;
    }
public:
    string name;
    int age;
```

```
};

//3. 成员函数和全局函数(Perosn 对象相加)
Person PersonPlusPerson(Person& p1,Person& p2){
	string newname = p1.name + p2.name;
	int newage = p1.age + p2.age;
	Person newperson(newname,newage);
	return newperson;
}

int main(){

	Person person("John",100);
	person.ShowPerson();

	cout << "---------" << endl;
	Person person1("John",20);
	Person person2("001", 10);
	//1.全局函数实现两个对象相加
	Person person3 = PersonPlusPerson(person1, person2);
	person1.ShowPerson();
	person2.ShowPerson();
	person3.ShowPerson();
	//2. 成员函数实现两个对象相加
	Person person4 = person1.PersonPlusPerson(person2);
	person4.ShowPerson();

	system("pause");
	return EXIT_SUCCESS;
}
```

4.4.2.3 const 修饰成员函数

用 const 修饰的成员函数时，const 修饰 this 指针指向的内存区域，成员函数体内不可以修改本类中的任何普通成员变量， 当成员变量类型符前用 mutable 修饰时例外。

```
//const 修饰成员函数
class Person{
public:
	Person(){
		this->mAge = 0;
		this->mID = 0;
	}
	//在函数括号后面加上 const,修饰成员变量不可修改,除了 mutable 变量
	void sonmeOperate() const{
		//this->mAge = 200; //mAge 不可修改
		this->mID = 10;
	}
```

```
	void ShowPerson(){
		cout << "ID:" << mID << " mAge:" << mAge << endl;
	}
private:
	int mAge;
	mutable int mID;
};

int main(){

	Person person;
	person.sonmeOperate();
	person.ShowPerson();

	system("pause");
	return EXIT_SUCCESS;
}
```

4.4.2.4 const 修饰对象(常对象) 常对象只能调用 const 的成员函数 常对象可访问 const 或非 const 数据成员，不能修改，除非成员用 mutable 修饰

```
class Person{
public:
	Person(){
		this->mAge = 0;
		this->mID = 0;
	}
	void ChangePerson() const{
		mAge = 100;
		mID = 100;
	}
	void ShowPerson(){
	  this->mAge = 1000;
		cout << "ID:" << this->mID << " Age:" << this->mAge << endl;
	}

public:
	int mAge;
	mutable int mID;
};

void test(){
	const Person person;
	//1. 可访问数据成员
	cout << "Age:" << person.mAge << endl;
	//person.mAge = 300; //不可修改
	person.mID = 1001; //但是可以修改mutable 修饰的成员变量
	//2. 只能访问 const 修饰的函数
	//person.ShowPerson();
```

```
	person.ChangePerson();
}
```

4.5 友元

类的主要特点之一是数据隐藏，即类的私有成员无法在类的外部(作用域之外)访问。但是，有时候需要在类的外部访问类的私有成员，怎么办？ 解决方法是使用友元函数，友元函数是一种特权函数，c++允许这个特权函数访问私有成员。这一点从现实生活中也可以很好的理解： 比如你的家，有客厅，有你的卧室，那么你的客厅是 Public 的，所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去，但是呢，你也可以允许你的闺蜜好基友进去。 程序员可以把一个全局函数、某个类中的成员函数、甚至整个类声明为友元。

4.5.1 友元语法

 friend 关键字只出现在声明处 其他类、类成员函数、全局函数都可声明为友元 友元函数不是类的成员，不带 this 指针 友元函数可访问对象任意成员属性，包括私有属性

```
class Building;
//友元类
class MyFriend{
public:
	//友元成员函数
	void LookAtBedRoom(Building& building);
	void PlayInBedRoom(Building& building);
};
class Building{
	//全局函数做友元函数
	friend void CleanBedRoom(Building& building);
#if 0
	//成员函数做友元函数
	friend void MyFriend::LookAtBedRoom(Building& building);
	friend void MyFriend::PlayInBedRoom(Building& building);
#else
	//友元类
	friend class MyFriend;
#endif
public:
	Building();
public:
	string mSittingRoom;
private:
	string mBedroom;
};

void MyFriend::LookAtBedRoom(Building& building){
```

```
	cout << "我的朋友参观" << building.mBedroom << endl;
}
void MyFriend::PlayInBedRoom(Building& building){
	cout << "我的朋友玩耍在" << building.mBedroom << endl;
}

//友元全局函数
void CleanBedRoom(Building& building){
	cout << "友元全局函数访问" << building.mBedroom << endl;
}

Building::Building(){
	this->mSittingRoom = "客厅";
	this->mBedroom = "卧室";
}

int main(){

	Building building;
	MyFriend myfriend;

	CleanBedRoom(building);
	myfriend.LookAtBedRoom(building);
	myfriend.PlayInBedRoom(building);

	system("pause");
	return EXIT_SUCCESS;
}
```

[友元类注意] 1．友元关系不能被继承。 2．友元关系是单向的，类 A 是类 B 的朋友，但类 B 不一定是类 A 的朋友。 3．友元关系不具有传递性。类 B 是类 A 的朋友，类 C 是类 B 的朋友，但类 C 不一定是类 A 的朋友。

思考: c++是纯面向对象的吗？ 如果一个类被声明为 friend,意味着它不是这个类的成员函数，却可以修改这个类的私有成员，而且必须列在类的定义中，因此他是一个特权函数。c++不是完全的面向对象语言，而只是一个混合产品。增加 friend 关键字只是用来解决一些实际问题，这也说明这种语言是不纯的。毕竟 c++设计的目的是为了实用性，而不是追求理想的抽象。

4.5.2 课堂练习

请编写电视机类，电视机有开机和关机状态，有音量，有频道，提供音量操作的方法，频道操作的方法。由于电视机只能逐一调整频道，不能指定频道，增加遥控类，遥控类除了拥有电视机已有的功能，再增加根据输入调台功能。

提示：遥控器可作为电视机类的友元类

```
class Remote;

class Television{
	friend class Remote;
public:
	enum{ On,Off }; //电视状态
	enum{ minVol,maxVol = 100 }; //音量从0到100
	enum{ minChannel = 1,maxChannel = 255 }; //频道从1到255
	Television(){
		mState = Off;
		mVolume = minVol;
		mChannel = minChannel;
	}

	//打开电视机
	void OnOrOff(){
		this->mState = (this->mState == On ? Off : On);
	}
	//调高音量
	void VolumeUp(){
		if (this->mVolume >= maxVol){
			return;
		}
		this->mVolume++;
	}
	//调低音量
	void VolumeDown(){
		if (this->mVolume <= minVol){
			return;
		}
		this->mVolume--;
	}
	//更换电视频道
	void ChannelUp(){
		if (this->mChannel >= maxChannel){
			return;
		}
		this->mChannel++;
	}
	void ChannelDown(){
		if (this->mChannel <= minChannel){
			return;
		}
		this->mChannel--;
	}
	//展示当前电视状态信息
	void ShowTeleState(){
		cout << "开机状态:" << (mState == On ? "已开机" : "已关机") <
< endl;
```

```
            if (mState == On){
                  cout << "当前音量:" << mVolume << endl;
                  cout << "当前频道:" << mChannel << endl;
            }
            cout << "-------------" << endl;
      }
private:
      int mState; //电视状态，开机，还是关机
      int mVolume; //电视机音量
      int mChannel; //电视频道
};

//电视机调台只能一个一个的调，遥控可以指定频道
//电视遥控器
class Remote{
public:
      Remote(Television* television){
            pTelevision = television;
      }
public:
      void OnOrOff(){
            pTelevision->OnOrOff();
      }
      //调高音量
      void VolumeUp(){
            pTelevision->VolumeUp();
      }
      //调低音量
      void VolumeDown(){
            pTelevision->VolumeDown();
      }
      //更换电视频道
      void ChannelUp(){
            pTelevision->ChannelUp();
      }
      void ChannelDown(){
            pTelevision->ChannelDown();
      }
      //设置频道 遥控新增功能
      void SetChannel(int channel){
            if (channel < Television::minChannel || channel > Televisio
n::maxChannel){
                  return;
            }
            pTelevision->mChannel = channel;
      }

      //显示电视当前信息
```

```
	void ShowTeleState(){
		pTelevision->ShowTeleState();
	}
private:
	Television* pTelevision;
};


//直接操作电视
void test01(){

	Television television;
	television.ShowTeleState();
	television.OnOrOff(); //开机
	television.VolumeUp(); //增加音量+1
	television.VolumeUp(); //增加音量+1
	television.VolumeUp(); //增加音量+1
	television.VolumeUp(); //增加音量+1
	television.ChannelUp(); //频道+1
	television.ChannelUp(); //频道+1
	television.ShowTeleState();
}

//通过遥控操作电视
void test02(){
	//创建电视
	Television television;
	//创建遥控
	Remote remote(&television);
	remote.OnOrOff();
	remote.ChannelUp();//频道+1
	remote.ChannelUp();//频道+1
	remote.ChannelUp();//频道+1
	remote.VolumeUp();//音量+1
	remote.VolumeUp();//音量+1
	remote.VolumeUp();//音量+1
	remote.VolumeUp();//音量+1
	remote.ShowTeleState();
}
```

4.5 强化训练(数组类封装)

MyArray.h

```
#ifndef MYARRAY_H
#define MYARRAY_H

class MyArray{
```

```
public:
	//无参构造函数，用户没有指定容量，则初始化为100
	MyArray();
	//有参构造函数，用户指定容量初始化
	explicit MyArray(int capacity);
	//用户操作接口
	//根据位置添加元素
	void SetData(int pos, int val);
	//获得指定位置数据
	int GetData(int pos);
	//尾插法
	void PushBack(int val);
	//获得长度
	int GetLength();
	//析构函数，释放数组空间
	~MyArray();
private:
	int mCapacity; //数组一共可容纳多少个元素
	int mSize; //当前有多少个元素
	int* pAdress; //指向存储数据的空间
};

#endif
```

MyArray.cpp

```
#include"MyArray.h"

MyArray::MyArray(){
	this->mCapacity = 100;
	this->mSize = 0;
	//在堆开辟空间
	this->pAdress = new int[this->mCapacity];
}
//有参构造函数，用户指定容量初始化
MyArray::MyArray(int capacity){
	this->mCapacity = capacity;
	this->mSize = 0;
	//在堆开辟空间
	this->pAdress = new int[capacity];
}
//根据位置添加元素
void MyArray::SetData(int pos, int val){
	if (pos < 0 || pos > mCapacity - 1){
		return;
	}
	pAdress[pos] = val;
}
```

```
//获得指定位置数据
int MyArray::GetData(int pos){
	return pAdress[pos];
}
//尾插法
void MyArray::PushBack(int val){
	if (mSize >= mCapacity){
		return;
	}
	this->pAdress[mSize] = val;
	this->mSize++;
}
//获得长度
int MyArray::GetLength(){
	return this->mSize;
}
//析构函数，释放数组空间
MyArray::~MyArray(){
	if (this->pAdress != nullptr){
		delete[] this->pAdress;
	}
}
```

TestMyArray.cpp

```
#include"MyArray.h"

void test(){
	//创建数组
	MyArray myarray(50);
	//数组中插入元素
	for (int i = 0; i < 50; i++){
		//尾插法
		myarray.PushBack(i);
		//myarray.SetData(i, i);
	}
	//打印数组中元素
	for (int i = 0; i < myarray.GetLength(); i++){
		cout << myarray.GetData(i) << " ";
	}
	cout << endl;
}
```

4.6 运算符重载

4.6.1 运算符重载基本概念

运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。

运算符重载(operator overloading)只是一种”语法上的方便”,也就是它只是另一种函数调用的方式。

在 c++中，可以定义一个处理类的新运算符。这种定义很像一个普通的函数定义，只是函数的名字由关键字 operator 及其紧跟的运算符组成。差别仅此而已。它像任何其他函数一样也是一个函数，当编译器遇到适当的模式时，就会调用这个函数。 语法： 定义重载的运算符就像定义函数，只是该函数的名字是 operator@,这里的@代表了被重载的运算符。函数的参数中参数个数取决于两个因素。 运算符是一元(一个参数)的还是二元(两个参数)； 运算符被定义为全局函数(对于一元是一个参数，对于二元是两个参数)还是成员函数(对于一元没有参数，对于二元是一个参数-此时该类的对象用作左耳参数)

[两个极端] 有些人很容易滥用运算符重载。它确实是一个有趣的工具。但是应该注意，它仅仅是一种语法上的方便而已，是另外一种函数调用的方式。从这个角度来看，只有在能使涉及类的代码更易写，尤其是更易读时(请记住，读代码的机会比我们写代码多多了)才有理由重载运算符。如果不是这样，就改用其他更易用，更易读的方式。 对于运算符重载，另外一个常见的反应是恐慌：突然之间，C 运算符的含义变得不同寻常了，一切都变了，所有 C 代码的功能都要改变！并非如此，对于内置的数据类型的表达式的的运算符是不可能改变的。（例如想重载 int 类型数据的+号）

4.6.2 运算符重载碰上友元函数

友元函数是一个全局函数，和我们上例写的全局函数类似，只是友元函数可以访问某个类私有数据。 案例: 重载左移操作符(<<),使得 cout 可以输出对象。

```
class Person{
	friend ostream& operator<<(ostream& os, Person& person);
public:
	Person(int id,int age){
		mID = id;
		mAge = age;
	}
private:
	int mID;
	int mAge;
};

ostream& operator<<(ostream& os, Person& person){
	os << "ID:" << person.mID << " Age:" << person.mAge;
	return os;
}

int main(){

	Person person(1001, 30);
```

```
	//cout << person; //cout.operator+(person)
	cout << person << " | " << endl;

	return EXIT_SUCCESS;
}
```

4.6.3 可重载的运算符

几乎 C 中所有的运算符都可以重载，但运算符重载的使用时相当受限制的。特别是不能使用 C 中当前没有意义的运算符(例如用**求幂)不能改变运算符优先级，不能改变运算符的参数个数。这样的限制有意义，否则，所有这些行为产生的运算符只会混淆而不是澄清寓语意。

可以重载的操作符

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	->*	‘	->
[]	()	new	delete	new[]	delete[]			

不能重载的算符

.	::	.*	?:	sizeof

4.6.4 自增自减(++/--)运算符重载

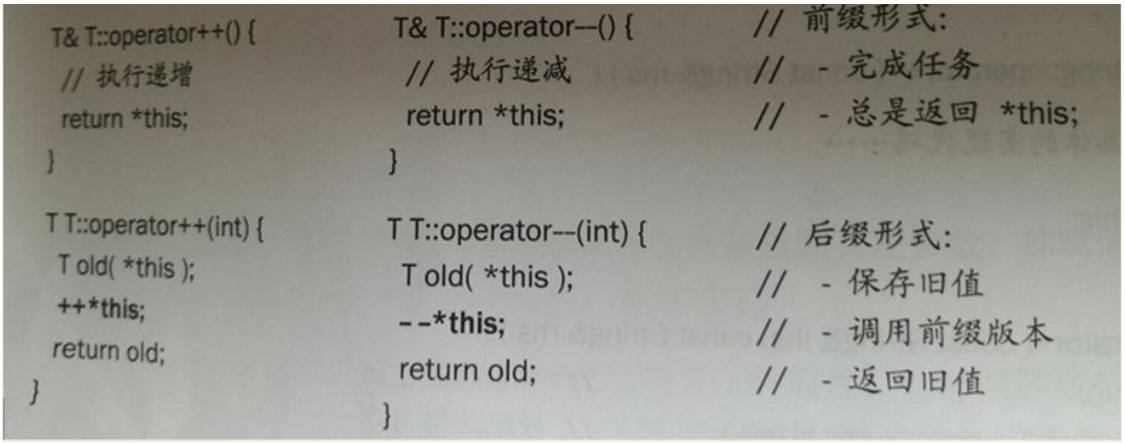
重载的++和--运算符有点让人不知所措，因为我们总是希望能根据它们出现在所作用对象的前面还是后面来调用不同的函数。解决办法很简单，例如当编译器看到++a(前置++)，它就调用 operator++(a),当编译器看到 a++（后置++），它就会去调用 operator++(a,int).

```
class Complex{
	friend ostream& operator<<(ostream& os,Complex& complex){
		os << "A:" << complex.mA << " B:" << complex.mB << endl;
		return os;
	}
public:
	Complex(){
		mA = 0;
		mB = 0;
```

```
	}
	//重载前置++
	Complex& operator++(){
		mA++;
		mB++;
		return *this;
	}
	//重载后置++
	Complex operator++(int){
		Complex temp;
		temp.mA = this->mA;
		temp.mB = this->mB;
		mA++;
		mB++;
		return temp;
	}
	//前置--
	Complex& operator--(){
		mA--;
		mB--;
		return *this;
	}
	//后置--
	Complex operator--(int){
		Complex temp;
		temp.mA = mA;
		temp.mB = mB;
		mA--;
		mB--;
		return temp;
	}
	void ShowComplex(){
		cout << "A:" << mA << " B:" << mB << endl;
	}
private:
	int mA;
	int mB;
};

void test(){
	Complex complex;
	complex++;
	cout << complex;
	++complex;
	cout << complex;

	Complex ret = complex++;
	cout << ret;
	cout << complex;
```

```
        cout << "------" << endl;
        ret--;
        --ret;
        cout << "ret:" << ret;
        complex--;
        --complex;
        cout << "complex:" << complex;
}
```

优先使用++和--的标准形式，优先调用前置++。 如果定义了++c，也要定义 c++，递增操作符比较麻烦，因为他们都有前缀和后缀形式，而两种语义略有不同。重载 operator++和 operator--时应该模仿他们对应的内置操作符。 对于++和--而言，后置形式是先返回，然后对象++或者--，返回的是对象的原值。前置形式，对象先++或--，返回当前对象，返回的是新对象。其标准形式为:



```
T& T::operator++() {
 // 执行递增
 return *this;
}

T T::operator++(int) {
 T old( *this );
 ++*this;
 return old;
}
```

```
T& T::operator--() {
 // 执行递减
 return *this;
}

T T::operator--(int) {
 T old( *this );
 --*this;
 return old;
}
```

```
// 前缀形式:
//  - 完成任务
//  - 总是返回 *this;

// 后缀形式:
//  - 保存旧值
//  - 调用前缀版本
//  - 返回旧值
```

调用代码时候，要优先使用前缀形式，除非确实需要后缀形式返回的原值，前缀和后缀形式语义上是等价的，输入工作量也相当，只是效率经常会略高一些，由于前缀形式少创建了一个临时对象。

4.6.5 指针运算符(*、->)重载

```
class Person{
public:
        Person(int param){
                this->mParam = param;
        }
        void PrintPerson(){
                cout << "Param:" << mParam << endl;
        }
private:
        int mParam;
};
```

```
class SmartPointer{
public:
	SmartPointer(Person* person){
		this->pPerson = person;
	}
	//重载指针的->、*操作符
	Person* operator->(){
		return pPerson;
	}
	Person& operator*(){
		return *pPerson;
	}
	~SmartPointer(){
		if (pPerson != NULL){
			delete pPerson;
		}
	}
public:
	Person* pPerson;
};

void test01(){

	//Person* person = new Person(100);
	//如果忘记释放，那么就会造成内存泄漏

	SmartPointer pointer(new Person(100));
	pointer->PrintPerson();
}
```

4.6.6 赋值(=)运算符重载

赋值符常常初学者的混淆。这是毫无疑问的，因为'='在编程中是最基本的运算符，可以进行赋值操作，也能引起拷贝构造函数的调用。

```
class Person{
	friend ostream& operator<<(ostream& os,const Person& person){
		os << "ID:" << person.mID << " Age:" << person.mAge << endl;
		return os;
	}
public:
	Person(int id,int age){
		this->mID = id;
		this->mAge = age;
	}
	//重载赋值运算符
	Person& operator=(const Person& person){
		this->mID = person.mID;
		this->mAge = person.mAge;
```

```
            return *this;
        }
private:
        int mID;
        int mAge;
};

//1. =号混淆的地方
void test01(){
        Person person1(10, 20);
        Person person2 = person1; //调用拷贝构造
        //如果一个对象还没有被创建，则必须初始化，也就是调用构造函数
        //上述例子由于person2 还没有初始化，所以会调用构造函数
        //由于person2 是从已有的person1 来创建的，所以只有一个选择
        //就是调用拷贝构造函数
        person2 = person1; //调用operator=函数
        //由于person2 已经创建，不需要再调用构造函数，这时候调用的是重载的赋值
运算符
}
//2. 赋值重载案例
void test02(){
        Person person1(20, 20);
        Person person2(30, 30);
        cout << "person1:" << person1;
        cout << "person2:" << person2;
        person2 = person1;
        cout << "person2:" << person2;
}
//常见错误，当准备给两个相同对象赋值时，应该首先检查一下这个对象是否对自身赋值
了
//对于本例来讲，无论如何执行这些赋值运算都是无害的，但如果对类的实现进行修改，
那么将会出现差异;
//3. 类中指针
class Person2{
        friend ostream& operator<<(ostream& os, const Person2& person){
                os << "Name:" << person.pName << " ID:" << person.mID << "
Age:" << person.mAge << endl;
                return os;
        }
public:
        Person2(char* name,int id, int age){
                this->pName = new char[strlen(name) + 1];
                strcpy(this->pName, name);
                this->mID = id;
                this->mAge = age;
        }
#if 1
        //重载赋值运算符
```

```
	Person2& operator=(const Person2& person){

		//注意:由于当前对象已经创建完毕，那么就有可能 pName 指向堆内存
		//这个时候如果直接赋值，会导致内存没有及时释放
		if (this->pName != NULL){
			delete[] this->pName;
		}

		this->pName = new char[strlen(person.pName) + 1];
		strcpy(this->pName,person.pName);
		this->mID = person.mID;
		this->mAge = person.mAge;
		return *this;
	}
#endif
	//析构函数
	~Person2(){
		if (this->pName != NULL){
			delete[] this->pName;
		}
	}
private:
	char* pName;
	int mID;
	int mAge;
};

void test03(){
	Person2 person1("John",20, 20);
	Person2 person2("Edward",30, 30);
	cout << "person1:" << person1;
	cout << "person2:" << person2;
	person2 = person1;
	cout << "person2:" << person2;
}
```

如果没有重载赋值运算符，编译器会自动创建默认的赋值运算符重载函数。行为类似默认拷贝构造，进行简单值拷贝。

4.6.7 等于和不等于(==、!=)运算符重载

```
class Complex{
public:
	Complex(char* name,int id,int age){
		this->pName = new char[strlen(name) + 1];
		strcpy(this->pName, name);
		this->mID = id;
		this->mAge = age;
	}
	//重载==号操作符
```

```
	bool operator==(const Complex& complex){
		if (strcmp(this->pName,complex.pName) == 0 &&
		    this->mID == complex.mID &&
		      this->mAge == complex.mAge){
		      return true;
		}
		return false;
	}
	//重载!=操作符
	bool operator!=(const Complex& complex){
		if (strcmp(this->pName, complex.pName) != 0 ||
		    this->mID != complex.mID ||
		      this->mAge != complex.mAge){
		      return true;
		}
		return false;
	}
	~Complex(){
		if (this->pName != NULL){
			delete[] this->pName;
		}
	}
private:
	char* pName;
	int mID;
	int mAge;
};
void test(){
	Complex complex1("aaa", 10, 20);
	Complex complex2("bbb", 10, 20);
	if (complex1 == complex2){ cout << "相等!" << endl; }
	if (complex1 != complex2){ cout << "不相等!" << endl; }
}
```

4.6.8 函数调用符号()重载

```
class Complex{
public:
	int Add(int x,int y){
		return x + y;
	}
	int operator()(int x,int y){
		return x + y;
	}
};
void test01(){
	Complex complex;
	cout << complex.Add(10,20) << endl;
	//对象当做函数来调用
```

```
	cout << complex(10, 20) << endl;
}
```

4.6.9 不要重载&&、||

不能重载 operator&& 和 operator|| 的原因是，无法在这两种情况下实现内置操作符的完整语义。说得更具体一些，内置版本版本特殊之处在于：内置版本的&&和||首先计算左边的表达式，如果这完全能够决定结果，就无需计算右边的表达式了--而且能够保证不需要。我们都已经习惯这种方便的特性了。 我们说操作符重载其实是另一种形式的函数调用而已，对于函数调用总是在函数执行之前对所有参数进行求值。

```
class Complex{
public:
	Complex(int flag){
		this->flag = flag;
	}
	Complex& operator+=(Complex& complex){
		this->flag = this->flag + complex.flag;
		return *this;
	}
	bool operator&&(Complex& complex){
		return this->flag && complex.flag;
	}
public:
	int flag;
};
int main(){

	Complex complex1(0);  //flag 0
	Complex complex2(1);  //flag 1

	//原来情况，应该从左往右运算，左边为假，则退出运算，结果为假
	//这边却是，先运算（complex1+complex2），导致，complex1 的flag 变为complex1+complex2 的值， complex1.a = 1
	// 1 && 1
	//complex1.operator&&(complex1.operator+=(complex2))
	if (complex1 && (complex1 += complex2)){
	  //complex1.operator+=(complex2)
		cout << "真!" << endl;
	}
	else{
		cout << "假!" << endl;
	}

	return EXIT_SUCCESS;
}
```

根据内置&&的执行顺序，我们发现这个案例中执行顺序并不是从左向右，而是先右猴左，这就是不满足我们习惯的特性了。由于 complex1 += complex2 先执行，导致 complex1 本身发生了变化，初始值是 0，现在经过+=运算变成 1,1 && 1 输出了真。

4.6.10 符号重载总结

=, [], () 和 -> 操作符只能通过成员函数进行重载　<< 和 >>只能通过全局函数配合友元函数进行重载　不要重载 && 和 || 操作符，因为无法实现短路规则 常规建议

运算符	建议使用
所有的一元运算符	成员
= () [] -> ->*	必须是成员
+= -= /= *= ^= &= != %= >>= <<=	成员
其它二员运算符	非成员

4.6.10 强化训练_字符串类封装

MyString.h

```
#define _CRT_SECURE_NO_WARNINGS
#pragma once
#include <iostream>
using namespace std;

class MyString
{
	friend ostream& operator<< (ostream  & out, MyString& str);
	friend istream& operator>>(istream& in, MyString& str);

public:
	MyString(const char *);
	MyString(const MyString&);
	~MyString();

	char& operator[](int index); //[]重载

	//=号重载
	MyString& operator=(const char * str);
	MyString& operator=(const MyString& str);

	//字符串拼接 重载+号
	MyString operator+(const char * str );
```

```
	MyString operator+(const MyString& str);

	//字符串比较
	bool operator== (const char * str);
	bool operator== (const MyString& str);
private:
	char * pString; //指向堆区空间
	int m_Size; //字符串长度 不算'\0'
};
```

MyString.cpp

```
#include "MyString.h"

//左移运算符
ostream& operator<< (ostream & out, MyString& str)
{
	out << str.pString;
	return out;
}
//右移运算符
istream& operator>>(istream& in, MyString& str)
{
	//先将原有的数据释放
	if (str.pString != NULL)
	{
		delete[] str.pString;
		str.pString = NULL;
	}
	char buf[1024]; //开辟临时的字符数组，保存用户输入内容
	in >> buf;

	str.pString = new char[strlen(buf) + 1];
	strcpy(str.pString, buf);
	str.m_Size = strlen(buf);

	return in;
}

//构造函数
MyString::MyString(const char * str)
{
	this->pString = new char[strlen(str) + 1];
	strcpy(this->pString, str);
	this->m_Size = strlen(str);
}

//拷贝构造
MyString::MyString(const MyString& str)
```

```
{
	this->pString = new char[strlen(str.pString) + 1];
	strcpy(this->pString, str.pString);
	this->m_Size = str.m_Size;
}
//析构函数
MyString::~MyString()
{
	if (this->pString!=NULL)
	{
		delete[]this->pString;
		this->pString = NULL;
	}
}

char& MyString::operator[](int index)
{
	return this->pString[index];
}

MyString& MyString::operator=(const char * str)
{
	if (this->pString != NULL){
		delete[] this->pString;
		this->pString = NULL;
	}
	this->pString = new char[strlen(str) + 1];
	strcpy(this->pString, str);
	this->m_Size = strlen(str);
	return *this;
}

MyString& MyString::operator=(const MyString& str)
{
	if (this->pString != NULL){
		delete[] this->pString;
		this->pString = NULL;
	}
	this->pString = new char[strlen(str.pString) + 1];
	strcpy(this->pString, str.pString);
	this->m_Size = str.m_Size;
	return *this;
}


MyString MyString::operator+(const char * str)
{
	int newsize = this->m_Size + strlen(str) + 1;
	char *temp = new char[newsize];
```

```
	memset(temp, 0, newsize);
	strcat(temp, this->pString);
	strcat(temp, str);

	MyString newstring(temp);
	delete[] temp;

	return newstring;
}

MyString MyString::operator+(const MyString& str)
{
	int newsize = this->m_Size + str.m_Size + 1;
	char *temp = new char[newsize];
	memset(temp, 0, newsize);
	strcat(temp, this->pString);
	strcat(temp, str.pString);

	MyString newstring(temp);
	delete[] temp;
	return newstring;
}

bool MyString::operator==(const char * str)
{
	if (strcmp(this->pString, str) == 0 && strlen(str) == this->m_Siz
e){
		return true;
	}

	return false;
}

bool MyString::operator==(const MyString& str)
{
	if (strcmp(this->pString, str.pString) == 0 && str.m_Size == this
->m_Size){
		return true;
	}

	return false;
}
```

TestMyString.cpp

```
void test01()
{
	MyString str("hello World");
```

```
	cout << str << endl;

	//cout << "请输入 MyString 类型字符串：" << endl;
	//cin >> str;

	//cout << "字符串为：  " << str << endl;

	//测试[]
	cout << "MyString 的第一个字符为：" << str[0] << endl;

	//测试 =
	MyString str2 = "^_^";
	MyString str3 = "";
	str3 = "aaaa";
	str3 = str2;
	cout << "str2 = " << str2 << endl;
	cout << "str3 = " << str3 << endl;

	//测试 +
	MyString str4 = "我爱";
	MyString str5 = "北京";
	MyString str6 = str4 + str5;
	MyString str7 = str6 + "天安门";

	cout << str7 << endl;

	//测试 ==
	if (str6 == str7)
	{
		cout << "s6 与 s7 相等" << endl;
	}
	else
	{
		cout << "s6 与 s7 不相等" << endl;
	}
}
```

4.6.11 附录：运算符和结合性

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	--
	()	圆括号	(表达式）/函数名(形参表)		--
	.	成员选择（对象）	对象.成员名		--
	->	成员选择（指针）	对象指针->成员名		--
2	-	负号运算符	-表达式	右到左	单目运算符
	~	按位取反运算符	~表达式		
	++	自增运算符	++变量名/变量名++		
	--	自减运算符	--变量名/变量名--		
	*	取值运算符	*指针变量		
	&	取地址运算符	&变量名		
	!	逻辑非运算符	!表达式		
	(类型)	强制类型转换	(数据类型)表达式		--
	sizeof	长度运算符	sizeof(表达式)		--
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		
	%	余数（取模）	整型表达式%整型表达式		
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		

6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		
	<	小于	表达式<表达式		
	<=	小于等于	表达式<=表达式		
7	==	等于	表达式==表达式	左到右	双目运算符
	! =	不等于	表达式!= 表达式		

8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符

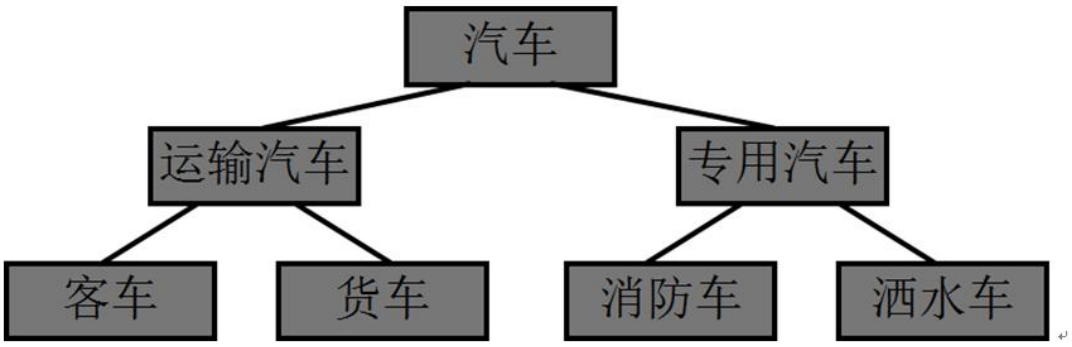
14	=	赋值运算符	变量=表达式	右到左	--
	/=	除后赋值	变量/=表达式		--
	*=	乘后赋值	变量*=表达式		--
	%=	取模后赋值	变量%=表达式		--
	+=	加后赋值	变量+=表达式		--

	-=	减后赋值	变量-=表达式		--
	<<=	左移后赋值	变量<<=表达式		--
	>>=	右移后赋值	变量>>=表达式		--
	&=	按位与后赋值	变量&=表达式		--
	^=	按位异或后赋值	变量^=表达式		--
	|=	按位或后赋值	变量|=表达式		--
15	,	逗号运算符	表达式,表达式,...	左到右	--

4.7 继承和派生

4.7.1 继承概述

4.7.1.1 为什么需要继承




```
//person(个人)类
class person
{
private:
    char name[10];
    int age;
    char sex;

public:
    void print();
};
```

```
//employee(职工)类
class employee
{
private:
    char name[10];
    int age;
    char sex;

    char department[20];
    float salary;
public:
    void print();
};
```

直接定义employee类，代码重复比较严重。

网页类

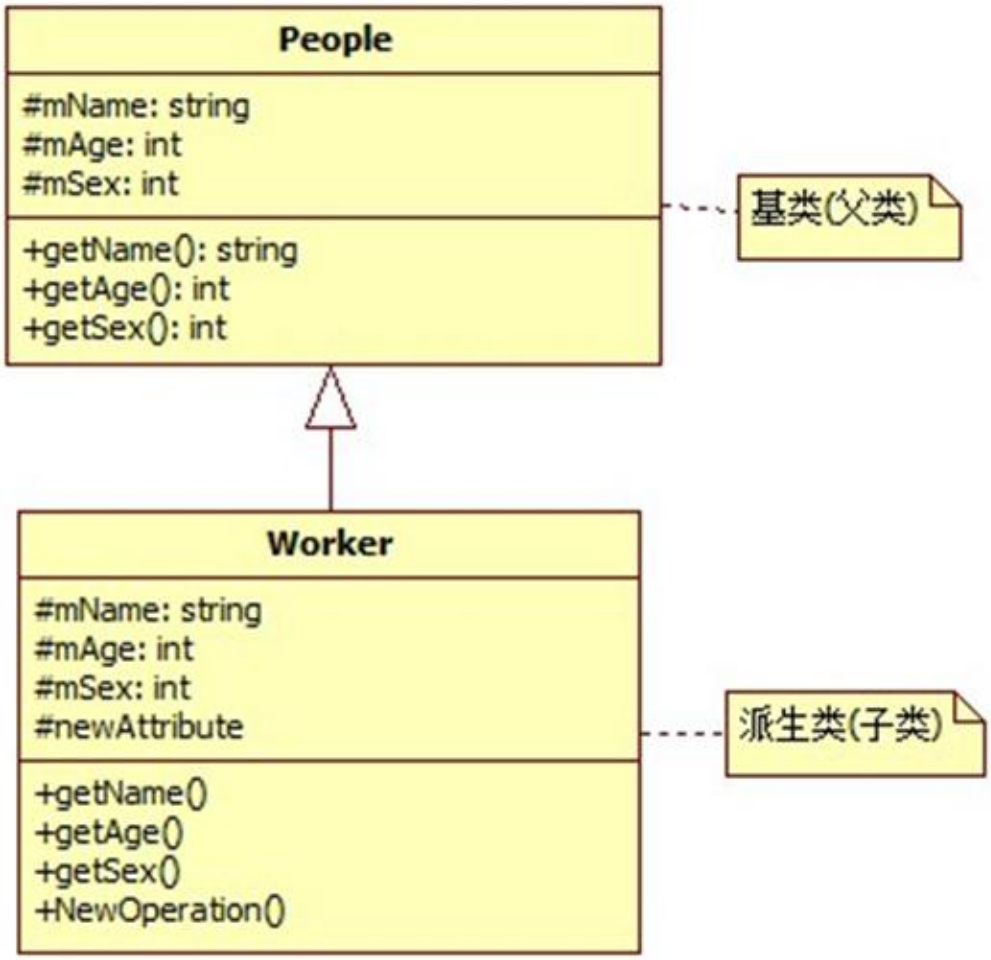
```
class IndexPage{
public:
    //网页头部
```

```
	void Header(){
		cout << "网页头部!" << endl;
	}
	//网页左侧菜单
	void LeftNavigation(){
		cout << "左侧导航菜单!" << endl;
	}
	//网页主体部分
	void MainBody(){
		cout << "首页网页主题内容!" << endl;
	}
	//网页底部
	void Footer(){
		cout << "网页底部!" << endl;
	}
private:
	string mTitle; //网页标题
};

#if 0
//如果不使用继承，那么定义新闻页类，需要重新写一遍已经有的代码
class NewsPage{
public:
	//网页头部
	void Header(){
		cout << "网页头部!" << endl;
	}
	//网页左侧菜单
	void LeftNavigation(){
		cout << "左侧导航菜单!" << endl;
	}
	//网页主体部分
	void MainBody(){
		cout << "新闻网页主体内容!" << endl;
	}
	//网页底部
	void Footer(){
		cout << "网页底部!" << endl;
	}
private:
	string mTitle; //网页标题
};

void test(){
	NewsPage* newspage = new NewsPage;
	newspage->Header();
	newspage->MainBody();
	newspage->LeftNavigation();
```

```
	newspage->Footer();
}
#else
//使用继承，可以复用已有的代码，新闻业除了主体部分不一样，其他都是一样的
class NewsPage : public IndexPage{
public:
	//网页主体部分
	void MainBody(){
		cout << "新闻网页主主体内容!" << endl;
	}
};
void test(){
	NewsPage* newspage = new NewsPage;
	newspage->Header();
	newspage->MainBody();
	newspage->LeftNavigation();
	newspage->Footer();
}
#endif
int main(){

	test();

	return EXIT_SUCCESS;
}
```

4.7.1.2 继承基本概念

c++最重要的特征是代码重用，通过继承机制可以利用已有的数据类型来定义新的数据类型，新的类不仅拥有旧类的成员，还拥有新定义的成员。 一个 B 类继承于 A 类，或称从类 A 派生类 B。这样的话，类 A 成为基类（父类）， 类 B 成为派生类（子类）。 派生类中的成员，包含两大部分： 一类是从基类继承过来的，一类是自己增加的成员。 从基类继承过过来的表现其共性，而新增的成员体现了其个性。



4.7.1.3 派生类定义

派生类定义格式：

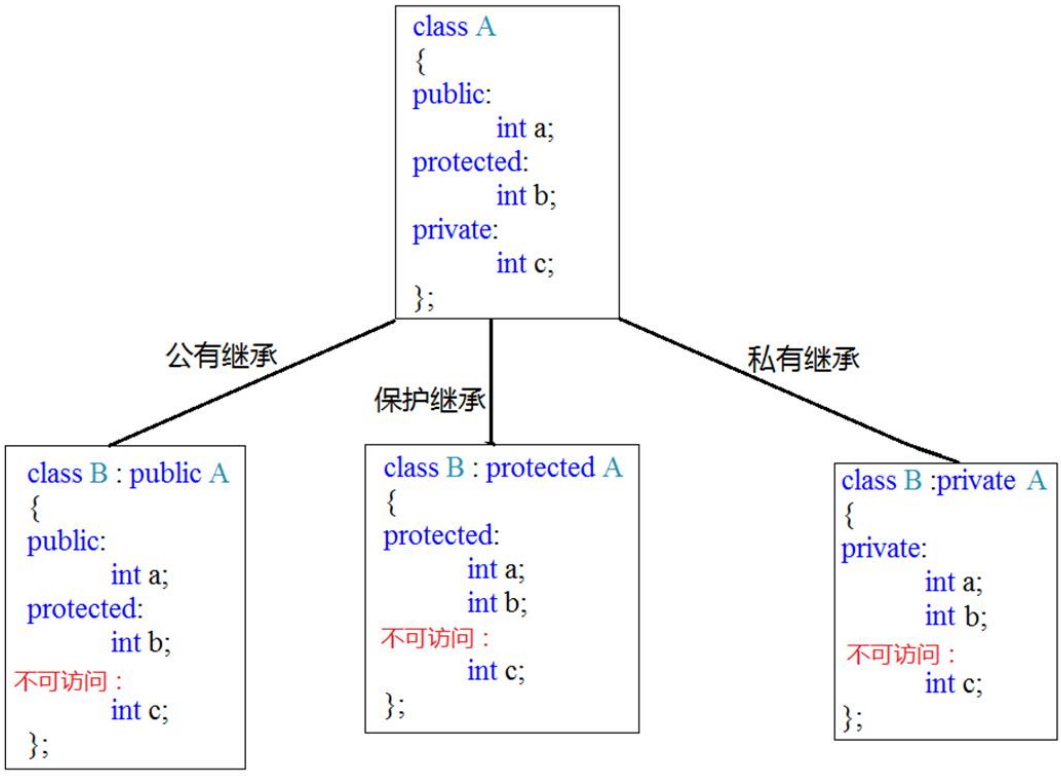
```
Class 派生类名 ： 继承方式 基类名{
        //派生类新增的数据成员和成员函数
}
```

三种继承方式： public： 公有继承 private： 私有继承 protected： 保护继承
从继承源上分： 单继承：指每个派生类只直接继承了一个基类的特征 多继承：指多个基类派生出一个派生类的继承关系,多继承的派生类直接继承了不止一个基类的特征

4.7.2 派生类访问控制

派生类继承基类，派生类拥有基类中全部成员变量和成员方法（除了构造和析构之外的成员方法），但是在派生类中，继承的成员并不一定能直接访问，不同的继承方式会导致不同的访问权限。 派生类的访问权限规则如下：

公有派生		私有派生		保护派生	
基类属性	派生类权限	基类属性	派生类权限	基类属性	派生类权限
私有	不能访问	私有	不能访问	私有	不能访问
保护	保护	保护	私有	保护	保护
公有	公有	公有	私有	公有	保护




```
//基类
class A{
public:
	int mA;
protected:
	int mB;
private:
	int mC;
};

//1. 公有(public)继承
class B : public A{
public:
	void PrintB(){
		cout << mA << endl; //可访问基类 public 属性
		cout << mB << endl; //可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
	}
};
class SubB : public B{
	void PrintSubB(){
		cout << mA << endl; //可访问基类 public 属性
		cout << mB << endl; //可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
```

```
	}
};
void test01(){

	B b;
	cout << b.mA << endl; //可访问基类 public 属性
	//cout << b.mB << endl; //不可访问基类 protected 属性
	//cout << b.mC << endl; //不可访问基类 private 属性
}

//2. 私有(private)继承
class C : private A{
public:
	void PrintC(){
		cout << mA << endl; //可访问基类 public 属性
		cout << mB << endl; //可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
	}
};
class SubC : public C{
	void PrintSubC(){
		//cout << mA << endl; //不可访问基类 public 属性
		//cout << mB << endl; //不可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
	}
};
void test02(){
	C c;
	//cout << c.mA << endl; //不可访问基类 public 属性
	//cout << c.mB << endl; //不可访问基类 protected 属性
	//cout << c.mC << endl; //不可访问基类 private 属性
}
//3. 保护(protected)继承
class D : protected A{
public:
	void PrintD(){
		cout << mA << endl; //可访问基类 public 属性
		cout << mB << endl; //可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
	}
};
class SubD : public D{
	void PrintD(){
		cout << mA << endl; //可访问基类 public 属性
		cout << mB << endl; //可访问基类 protected 属性
		//cout << mC << endl; //不可访问基类 private 属性
	}
```

```
};
void test03(){
	D d;
	//cout << d.mA << endl; //不可访问基类 public 属性
	//cout << d.mB << endl; //不可访问基类 protected 属性
	//cout << d.mC << endl; //不可访问基类 private 属性
}
```

4.7.3 继承中的构造和析构

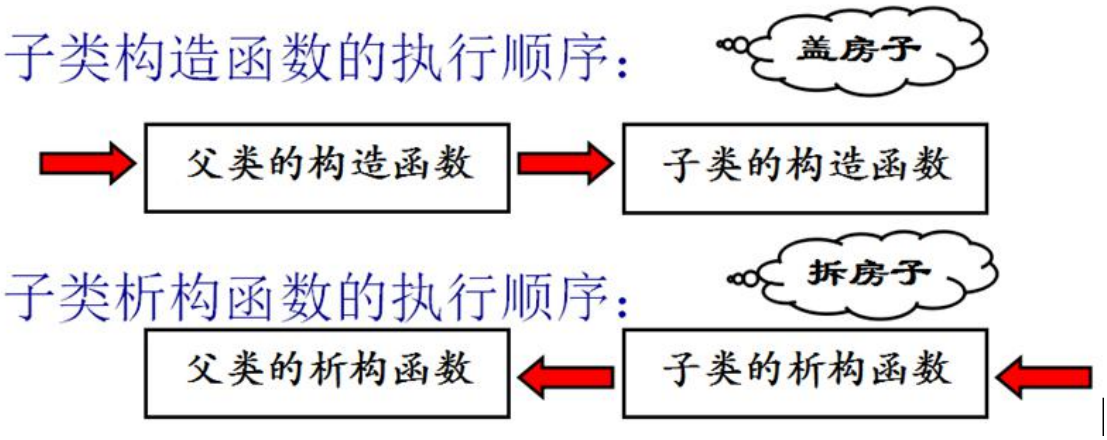
4.7.3.1 继承中的对象模型

在 C++编译器的内部可以理解为结构体，子类是由父类成员叠加子类新成员而成：

```
class Aclass{
public:
	int mA;
	int mB;
};
class Bclass : public Aclass{
public:
	int mC;
};
class Cclass : public Bclass{
public:
	int mD;
};
void test(){
	cout << "A size:" << sizeof(Aclass) << endl;
	cout << "B size:" << sizeof(Bclass) << endl;
	cout << "C size:" << sizeof(Cclass) << endl;
}
```

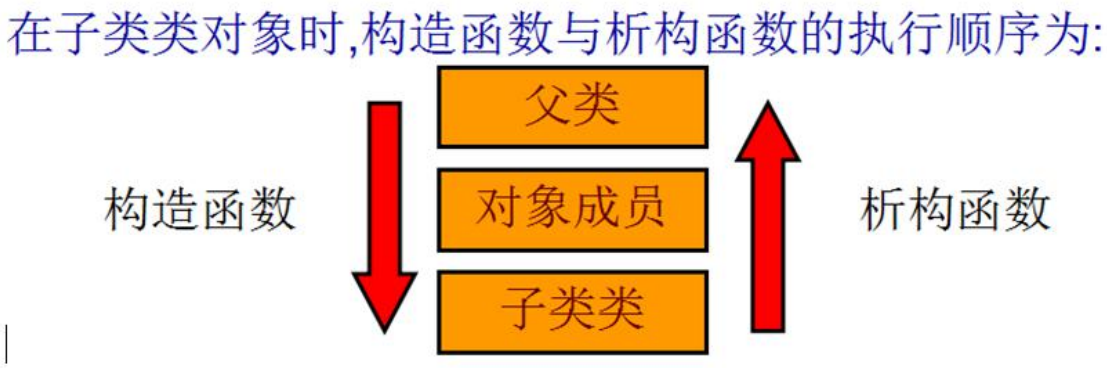
4.7.3.2 对象构造和析构的调用原则

继承中的构造和析构 子类对象在创建时会首先调用父类的构造函数 父类构造函数执行完毕后，才会调用子类的构造函数 当父类构造函数有参数时，需要在子类初始化列表(参数列表)中显示调用父类构造函数 析构函数调用顺序和构造函数相反




```
class A{
public:
	A(){
		cout << "A 类构造函数!" << endl;
	}
	~A(){
		cout << "A 类析构函数!" << endl;
	}
};

class B : public A{
public:
	B(){
		cout << "B 类构造函数!" << endl;
	}
	~B(){
		cout << "B 类析构函数!" << endl;
	}
};

class C : public B{
public:
	C(){
		cout << "C 类构造函数!" << endl;
	}
	~C(){
		cout << "C 类析构函数!" << endl;
	}
};

void test(){
	C c;
}
```

继承与组合混搭的构造和析构




```
class D{
public:
	D(){
		cout << "D 类构造函数!" << endl;
	}
	~D(){
		cout << "D 类析构函数!" << endl;
	}
};
class A{
public:
	A(){
		cout << "A 类构造函数!" << endl;
	}
	~A(){
		cout << "A 类析构函数!" << endl;
	}
};
class B : public A{
public:
	B(){
		cout << "B 类构造函数!" << endl;
	}
	~B(){
		cout << "B 类析构函数!" << endl;
	}
};
class C : public B{
public:
	C(){
		cout << "C 类构造函数!" << endl;
	}
```

```
	~C(){
		cout << "C 类析构函数!" << endl;
	}
public:
	D c;
};
void test(){
	C c;
}
```

4.7.4 继承中同名成员的处理方法

当子类成员和父类成员同名时，子类依然从父类继承同名成员 如果子类有成员和父类同名，子类访问其成员默认访问子类的成员(本作用域，就近原则) 在子类通过作用域::进行同名成员区分(在派生类中使用基类的同名成员，显示使用类名限定符)

```
class Base{
public:
	Base():mParam(0){}
	void Print(){ cout << mParam << endl; }
public:
	int mParam;
};

class Derived : public Base{
public:
	Derived():mParam(10){}
	void Print(){
		//在派生类中使用和基类的同名成员,显示使用类名限定符
		cout << Base::mParam << endl;
		cout << mParam << endl;
	}
	//返回基类重名成员
	int& getBaseParam(){ return  Base::mParam; }
public:
	int mParam;
};

int main(){

	Derived derived;
	//派生类和基类成员属性重名，子类访问成员默认是子类成员
	cout << derived.mParam << endl; //10
	derived.Print();
	//类外如何获得基类重名成员属性
	derived.getBaseParam() = 100;
	cout << "Base:mParam:" << derived.getBaseParam() << endl;
```

```
	return EXIT_SUCCESS;
}
```

注意: 如果重新定义了基类中的重载函数，将会发生什么？

```
class Base{
public:
	void func1(){
		cout << "Base::void func1()" << endl;
	};
	void func1(int param){
		cout << "Base::void func1(int param)" << endl;
	}
	void myfunc(){
		cout << "Base::void myfunc()" << endl;
	}
};
class Derived1 : public Base{
public:
	void myfunc(){
		cout << "Derived1::void myfunc()" << endl;
	}
};
class Derived2 : public Base{
public:
	//改变成员函数的参数列表
	void func1(int param1, int param2){
		cout << "Derived2::void func1(int param1,int param2)" << en
dl;
	};
};
class Derived3 : public Base{
public:
	//改变成员函数的返回值
	int func1(int param){
		cout << "Derived3::int func1(int param)" << endl;
		return 0;
	}
};
int main(){

	Derived1 derived1;
	derived1.func1();
	derived1.func1(20);
	derived1.myfunc();
	cout << "-------------" << endl;
	Derived2 derived2;
	//derived2.func1();  //func1 被隐藏
	//derived2.func1(20); //func2 被隐藏
```

```
	derived2.func1(10,20); //重载func1之后，基类的函数被隐藏
	derived2.myfunc();
	cout << "-------------" << endl;
	Derived3 derived3;
	//derived3.func1();  没有重新定义的重载版本被隐藏
	derived3.func1(20);
	derived3.myfunc();

	return EXIT_SUCCESS;
}
```

Derive1 重定义了 Base 类的 myfunc 函数，derive1 可访问 func1 及其重载版本的函数。 Derive2 通过改变函数参数列表的方式重新定义了基类的 func1 函数，则从基类中继承来的其他重载版本被隐藏，不可访问 Derive3 通过改变函数返回类型的方式重新定义了基类的 func1 函数，则从基类继承来的没有重新定义的重载版本的函数将被隐藏。

任何时候重新定义基类中的一个重载函数，在新类中所有的其他版本将被自动隐藏.

4.7.5 非自动继承的函数

不是所有的函数都能自动从基类继承到派生类中。构造函数和析构函数用来处理对象的创建和析构操作，构造和析构函数只知道对它们的特定层次的对象做什么，也就是说构造函数和析构函数不能被继承，必须为每一个特定的派生类分别创建。另外 operator=也不能被继承，因为它完成类似构造函数的行为。也就是说尽管我们知道如何由=右边的对象如何初始化=左边的对象的所有成员，但是这个并不意味着对其派生类依然有效。 在继承的过程中，如果没有创建这些函数，编译器会自动生成它们。

4.7.6 继承中的静态成员特性

静态成员函数和非静态成员函数的共同点:

1. 他们都可以被继承到派生类中。
2. 如果重新定义一个静态成员函数，所有在基类中的其他重载函数会被隐藏。
3. 如果我们改变基类中一个函数的特征，所有使用该函数名的基类版本都会被隐藏。

```
class Base{
public:
	static int getNum(){ return sNum; }
	static int getNum(int param){
		return sNum + param;
	}
public:
```

```
	static int sNum;
};
int Base::sNum = 10;

class Derived : public Base{
public:
	static int sNum; //基类静态成员属性将被隐藏
#if 0
	//重定义一个函数，基类中重载的函数被隐藏
	static int getNum(int param1, int param2){
		return sNum + param1 + param2;
	}
#else
	//改变基类函数的某个特征，返回值或者参数个数，将会隐藏基类重载的函数
	static void getNum(int param1, int param2){
		cout <<  sNum + param1 + param2 << endl;
	}
#endif
};
int Derived::sNum = 20;
```

4.7.6 多继承

4.7.6.1 多继承概念

我们可以从一个类继承，我们也可以能同时从多个类继承，这就是多继承。但是由于多继承是非常受争议的，从多个类继承可能会导致函数、变量等同名导致较多的歧义。

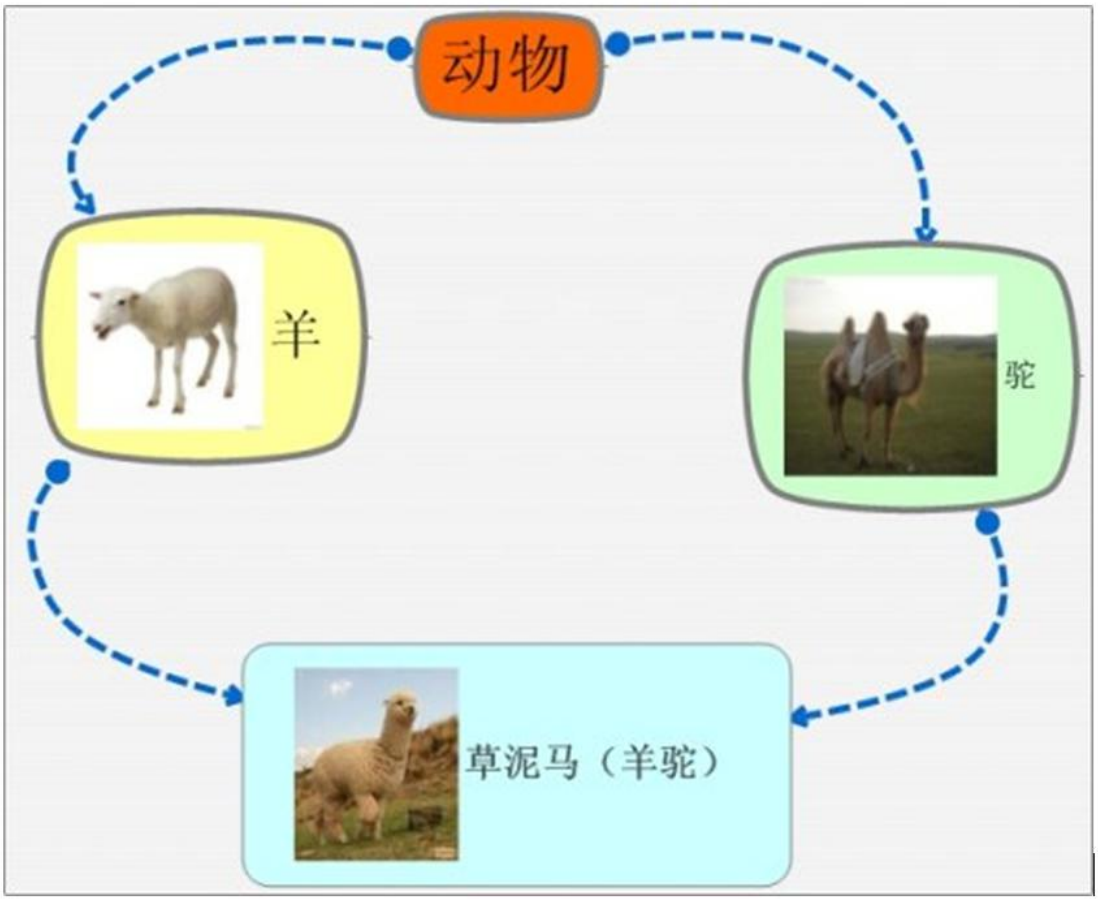
```
class Base1{
public:
	void func1(){ cout << "Base1::func1" << endl; }
};
class Base2{
public:
	void func1(){ cout << "Base2::func1" << endl; }
	void func2(){ cout << "Base2::func2" << endl; }
};
//派生类继承Base1、Base2
class Derived : public Base1, public Base2{};
int main(){

	Derived derived;
	//func1 是从 Base1 继承来的还是从 Base2 继承来的？
	//derived.func1();
	derived.func2();

	//解决歧义:显示指定调用那个基类的func1
	derived.Base1::func1();
```

```
        derived.Base2::func1();

        return EXIT_SUCCESS;
}
```

多继承会带来一些二义性的问题， 如果两个基类中有同名的函数或者变量，那么通过派生类对象去访问这个函数或变量时就不能明确到底调用从基类 1 继承的版本还是从基类 2 继承的版本？ 解决方法就是显示指定调用那个基类的版本。

4.7.6.2 菱形继承和虚继承

两个派生类继承同一个基类而又有某个类同时继承者两个派生类，这种继承被称为菱形继承，或者钻石型继承。



这种继承所带来的问题：

1. 羊继承了动物的数据和函数，鸵同样继承了动物的数据和函数，当草泥马调用函数或者数据时，就会产生二义性。

2. 草泥马继承自动物的函数和数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

```
class BigBase{
public:
	BigBase(){ mParam = 0; }
	void func(){ cout << "BigBase::func" << endl; }
public:
	int mParam;
};

class Base1 : public BigBase{};
class Base2 : public BigBase{};
class Derived : public Base1, public Base2{};

int main(){

	Derived derived;
	//1. 对“func”的访问不明确
	//derived.func();
	//cout << derived.mParam << endl;
	cout << "derived.Base1::mParam:" << derived.Base1::mParam << endl;
	cout << "derived.Base2::mParam:" << derived.Base2::mParam << endl;

	//2. 重复继承
	cout << "Derived size:" << sizeof(Derived) << endl; //8

	return EXIT_SUCCESS;
}
```

上述问题如何解决？对于调用二义性，那么可通过指定调用那个基类的方式来解决，那么重复继承怎么解决？ 对于这种菱形继承所带来的两个问题，c++为我们提供了一种方式，采用虚基类。那么我们采用虚基类方式将代码修改如下：

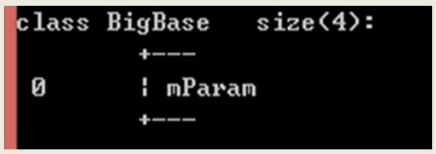
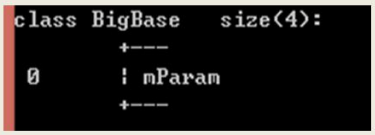
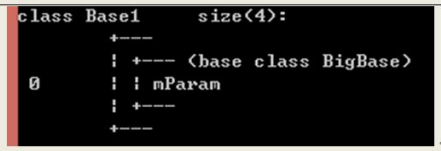
```
class BigBase{
public:
	BigBase(){ mParam = 0; }
	void func(){ cout << "BigBase::func" << endl; }
public:
	int mParam;
};

class Base1 : virtual public BigBase{};
class Base2 : virtual public BigBase{};
class Derived : public Base1, public Base2{};

int main(){

	Derived derived;
```

```
        //二义性问题解决
        derived.func();
        cout << derived.mParam << endl;
        //输出结果:12
        cout << "Derived size:" << sizeof(Derived) << endl;

        return EXIT_SUCCESS;
}
```

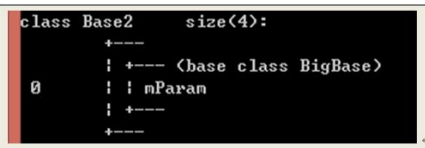
以上程序 Base1，Base2 采用虚继承方式继承 BigBase,那么 BigBase 被称为虚基类。通过虚继承解决了菱形继承所带来的二义性问题。但是虚基类是如何解决二义性的呢？并且 derived 大小为 12 字节，这是怎么回事？

4.7.6.3 虚继承实现原理

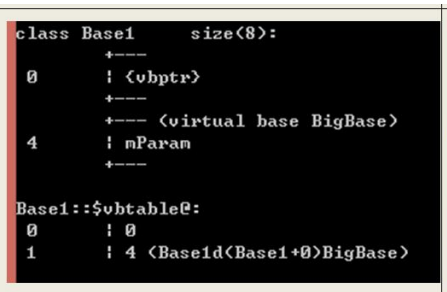
```
class BigBase{
public:
        BigBase(){ mParam = 0; }
        void func(){ cout << "BigBase::func" << endl; }
public: int mParam;
};
#if 0 //虚继承
class Base1 : virtual public BigBase{};
class Base2 : virtual public BigBase{};
#else //普通继承
class Base1 :  public BigBase{};
class Base2 :  public BigBase{};
#endif
class Derived : public Base1, public Base2{};
```

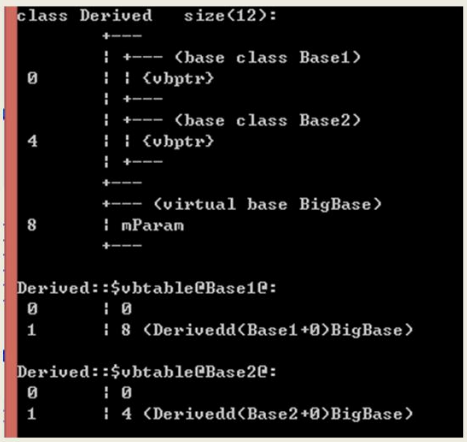
	普通继承	虚继承
BigBase：		
Base1：		

Base2：



Derived：





通过内存图，我们发现普通继承和虚继承的对象内存图是不一样的。我们也可以猜测到编译器肯定对我们编写的程序做了一些手脚。

BigBase 菱形最顶层的类，内存布局图没有发生改变。 Base1 和 Base2 通过虚继承的方式派生自 BigBase,这两个对象的布局图中可以看出编译器为我们的对象中增加了一个 vbptr (virtual base pointer),vbptr 指向了一张表，这张表保存了当前的虚指针相对于虚基类的首地址的偏移量。 Derived 派生于 Base1 和 Base2,继承了两个基类的 vbptr 指针，并调整了 vbptr 与虚基类的首地址的偏移量。

由此可知编译器帮我们做了一些幕后工作，使得这种菱形问题在继承时候能只继承一份数据，并且也解决了二义性的问题。现在模型就变成了 Base1 和 Base2 Derived 三个类对象共享了一份 BigBase 数据。

当使用虚继承时，虚基类是被共享的，也就是在继承体系中无论被继承多少次，对象内存模型中均只会出现一个虚基类的子对象（这和多继承是完全不同的）。即使共享虚基类，但是必须要有一个类来完成基类的初始化（因为所有的对象都必须被初始化，哪怕是默认的），同时还不能够重复进行初始化，那到底谁应该负责完成初始化呢？C++标准中选择在每一次继承子类中都必须书写初始化语句（因为每一次继承子类可能都会用来定义对象），但是虚基类的初始化是由最后的子类完成，其他的初始化语句都不会调用。

```
class BigBase{
public:
	BigBase(int x){mParam = x;}
	void func(){cout << "BigBase::func" << endl;}
public:
	int mParam;
};
class Base1 : virtual public BigBase{
public:
	Base1() :BigBase(10){} //不调用BigBase 构造
};
class Base2 : virtual public BigBase{
public:
	Base2() :BigBase(10){} //不调用BigBase 构造
};

class Derived : public Base1, public Base2{
public:
	Derived() :BigBase(10){} //调用BigBase 构造
};
//每一次继承子类中都必须书写初始化语句
int main(){
	Derived derived;
	return EXIT_SUCCESS;
}
```

注意： 虚继承只能解决具备公共祖先的多继承所带来的二义性问题，不能解决没有公共祖先的多继承的.

工程开发中真正意义上的多继承是几乎不被使用，因为多重继承带来的代码复杂性远多于其带来的便利，多重继承对代码维护性上的影响是灾难性的，在设计方法上，任何多继承都可以用单继承代替。

4.8 多态

4.8.1 多态基本概念

多态是面向对象程序设计语言中数据抽象和继承之外的第三个基本特征。 多态性(polymorphism)提供接口与具体实现之间的另一层隔离，从而将”what”和”how”分离开来。多态性改善了代码的可读性和组织性，同时也使创建的程序具有可扩展性，项目不仅在最初创建时期可以扩展，而且当项目在需要有新的功能时也能扩展。 c++支持编译时多态(静态多态)和运行时多态(动态多态)，运算符重载和函数重载就是编译时多态，而派生类和虚函数实现运行时多态。 静态多态和动态多态的区别就是函数地址是早绑定(静态联编)还是晚绑定(动态联编)。如果函数的调用，在编译阶段就可以确定函数的调用地址，并产生代码，就是静态多态(编译时

多态)，就是说地址是早绑定的。而如果函数的调用地址不能编译不能在编译期间确定，而需要在运行时才能决定，这这就属于晚绑定(动态多态,运行时多态)。

```
//计算器
class Caculator{
public:
	void setA(int a){
		this->mA = a;
	}
	void setB(int b){
		this->mB = b;
	}
	void setOperator(string oper){
		this->mOperator = oper;
	}
	int getResult(){

		if (this->mOperator == "+"){
			return mA + mB;
		}
		else if (this->mOperator == "-"){
			return mA - mB;
		}
		else if (this->mOperator == "*"){
			return mA * mB;
		}
		else if (this->mOperator == "/"){
			return mA / mB;
		}
	}
private:
	int mA;
	int mB;
	string mOperator;
};

//这种程序不利于扩展，维护困难，如果修改功能或者扩展功能需要在源代码基础上修改
//面向对象程序设计一个基本原则:开闭原则(对修改关闭，对扩展开放)

//抽象基类
class AbstractCaculator{
public:
	void setA(int a){
		this->mA = a;
	}
	virtual void setB(int b){
		this->mB = b;
	}
	virtual int getResult() = 0;
```

```
protected:
	int mA;
	int mB;
	string mOperator;
};

//加法计算器
class PlusCaculator : public AbstractCaculator{
public:
	virtual int getResult(){
		return mA + mB;
	}
};

//减法计算器
class MinusCaculator : public AbstractCaculator{
public:
	virtual int getResult(){
		return mA - mB;
	}
};

//乘法计算器
class MultipliesCaculator : public AbstractCaculator{
public:
	virtual int getResult(){
		return mA * mB;
	}
};

void DoBussiness(AbstractCaculator* caculator){
	int a = 10;
	int b = 20;
	caculator->setA(a);
	caculator->setB(b);
	cout << "计算结果: " << caculator->getResult() << endl;
	delete caculator;
}
```

4.8.2 向上类型转换及问题

4.8.2.1 问题抛出

对象可以作为自己的类或者作为它的基类的对象来使用。还能通过基类的地址来操作它。取一个对象的地址(指针或引用)，并将其作为基类的地址来处理，这种称为向上类型转换。 也就是说：父类引用或指针可以指向子类对象，通过父类指针或引用来操作子类对象。

```
class Animal{
public:
	void speak(){
		cout << "动物在唱歌..." << endl;
	}
};

class Dog : public Animal{
public:
	void speak(){
		cout << "小狗在唱歌..." << endl;
	}
};

void DoBussiness(Animal& animal){
	animal.speak();
}

void test(){
	Dog dog;
	DoBussiness(dog);
}
```

运行结果: 动物在唱歌 问题抛出: 我们给 DoBussiness 传入的对象是 dog，而不是 animal 对象，输出的结果应该是 Dog::speak。

问题解决思路

解决这个问题，我们需要了解下绑定(捆绑,binding)概念。 把函数体与函数调用相联系称为绑定(捆绑，binding)

当绑定在程序运行之前(由编译器和连接器)完成时，称为早绑定(early binding).C 语言中只有一种函数调用方式，就是早绑定。 上面的问题就是由于早绑定引起的，因为编译器在只有 Animal 地址时并不知道要调用的正确函数。编译是根据指向对象的指针或引用的类型来选择函数调用。这个时候由于 DoBussiness 的参数类型是 Animal&,编译器确定了应该调用的 speak 是 Animal::speak 的，而不是真正传入的对象 Dog::speak。 解决方法就是迟绑定(迟捆绑,动态绑定,运行时绑定，late binding),意味着绑定要根据对象的实际类型，发生在运行。 C++语言要实现这种动态绑定，必须有某种机制来确定运行时对象的类型并调用合适的成员函数。对于一种编译语言，编译器并不知道实际的对象类型（编译器并不知道 Animal 类型的指针或引用指向的实际的对象类型）。

问题解决方案(虚函数,vitual function)

C++动态多态性是通过虚函数来实现的，虚函数允许子类（派生类）重新定义父类（基类）成员函数，而子类（派生类）重新定义父类（基类）虚函数的做法称为覆盖(override)，或者称为重写。 对于特定的函数进行动态绑定，c++要求在基类中

声明这个函数的时候使用 virtual 关键字,动态绑定也就对 virtual 函数起作用. 为创建一个需要动态绑定的虚成员函数，可以简单在这个函数声明前面加上 virtual 关键字，定义时候不需要. 如果一个函数在基类中被声明为 virtual，那么在所有派生类中它都是 virtual 的. 在派生类中 virtual 函数的重定义称为重写(override). Virtual 关键字只能修饰成员函数. 构造函数不能为虚函数

注意: 仅需要在基类中声明一个函数为 virtual.调用所有匹配基类声明行为的派生类函数都将使用虚机制。虽然可以在派生类声明前使用关键字 virtual(这也是无害的)，但这个样会使得程序显得冗余和杂乱。(我建议写上)

```
class Animal{
public:
	virtual void speak(){
		cout << "动物在唱歌..." << endl;
	}
};
class Dog : public Animal{
public:
	virtual void speak(){
		cout << "小狗在唱歌..." << endl;
	}
};
void DoBussiness(Animal& animal){
	animal.speak();
}
void test(){
	Dog dog;
	DoBussiness(dog);
}
```

4.8.3 C++如何实现动态绑定

动态绑定什么时候发生？所有的工作都是由编译器在幕后完成。当我们告诉通过创建一个 virtual 函数来告诉编译器要进行动态绑定，那么编译器就会根据动态绑定机制来实现我们的要求， 不会再执行早绑定。

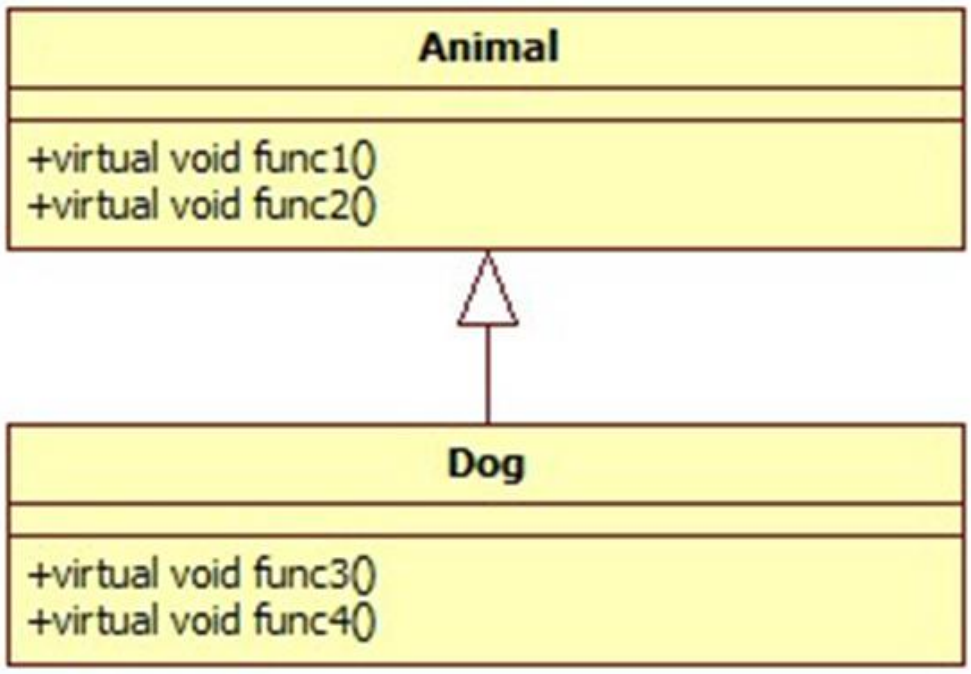
问题:C++的动态捆绑机制是怎么样的？ 首先，我们看看编译器如何处理虚函数。当编译器发现我们的类中有虚函数的时候，编译器会创建一张虚函数表，把虚函数的函数入口地址放到虚函数表中，并且在类中秘密增加一个指针，这个指针就是 vpointer(缩写 vptr)，这个指针是指向对象的虚函数表。在多态调用的时候，根据 vptr 指针，找到虚函数表来实现动态绑定。

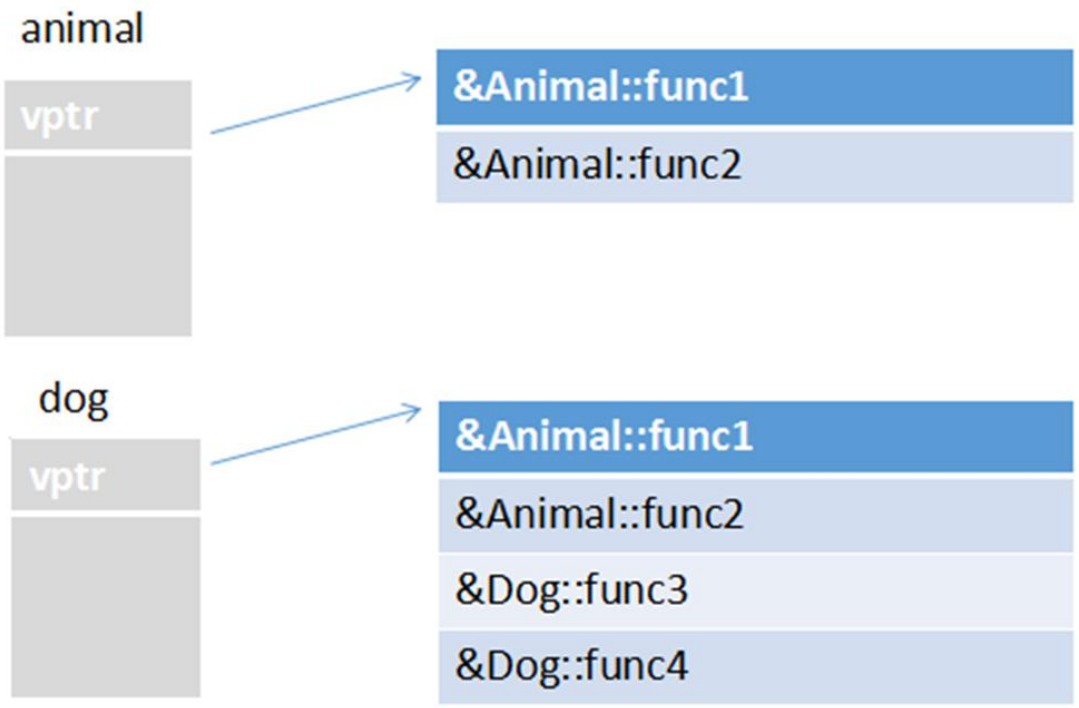
验证对象中的虚指针：

```
class A{
public:
	virtual void func1(){}
	virtual void func2(){}
```

```
};

//B 类为空，那么大小应该是 1 字节，实际情况是这样吗？
class B : public A{};

void test(){
      cout << "A size:" << sizeof(A) << endl;
      cout << "B size:" << sizeof(B) << endl;
}
```

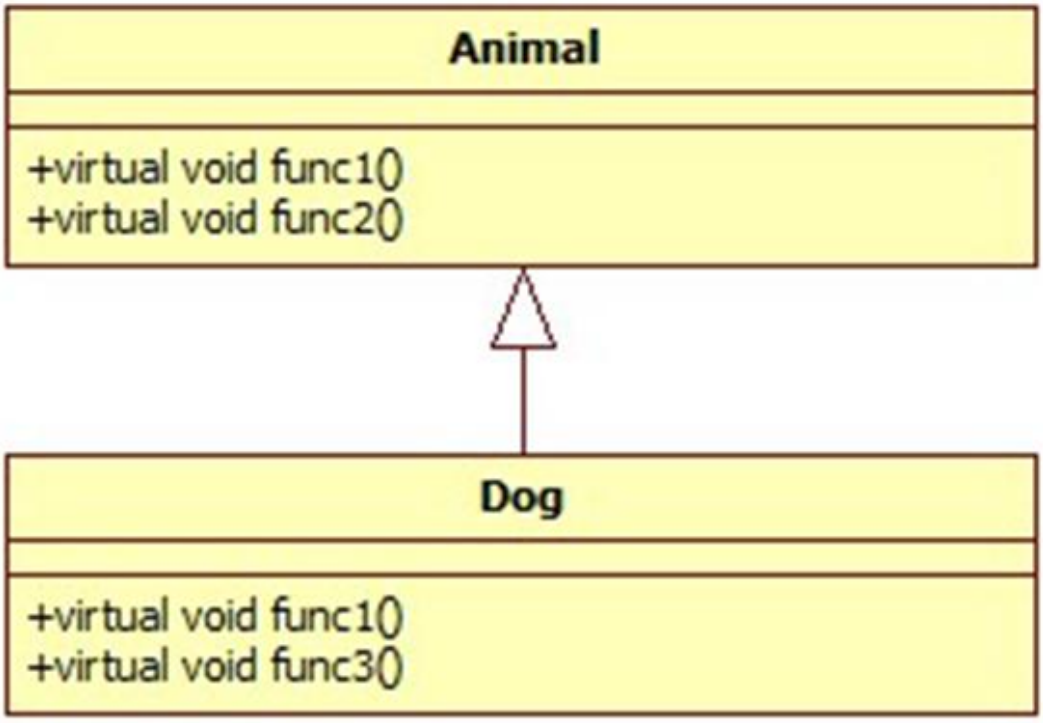
在编译阶段，编译器秘密增加了一个 vptr 指针，但是此时 vptr 指针并没有初始化指向虚函数表(vtable),什么时候 vptr 才会指向虚函数表？在对象构建的时候，也就是在对象初始化调用构造函数的时候。编译器首先默认会在我们所编写的每一个构造函数中，增加一些 vptr 指针初始化的代码。如果没有提供构造函数，编译器会提供默认的构造函数，那么就会在默认构造函数里做此项工作，初始化 vptr 指针，使之指向本对象的虚函数表。 起初，子类继承基类，子类继承了基类的 vptr 指针，这个 vptr 指针是指向基类虚函数表，当子类调用构造函数，使得子类的 vptr 指针指向了子类的虚函数表。 当子类无重写基类虚函数时:

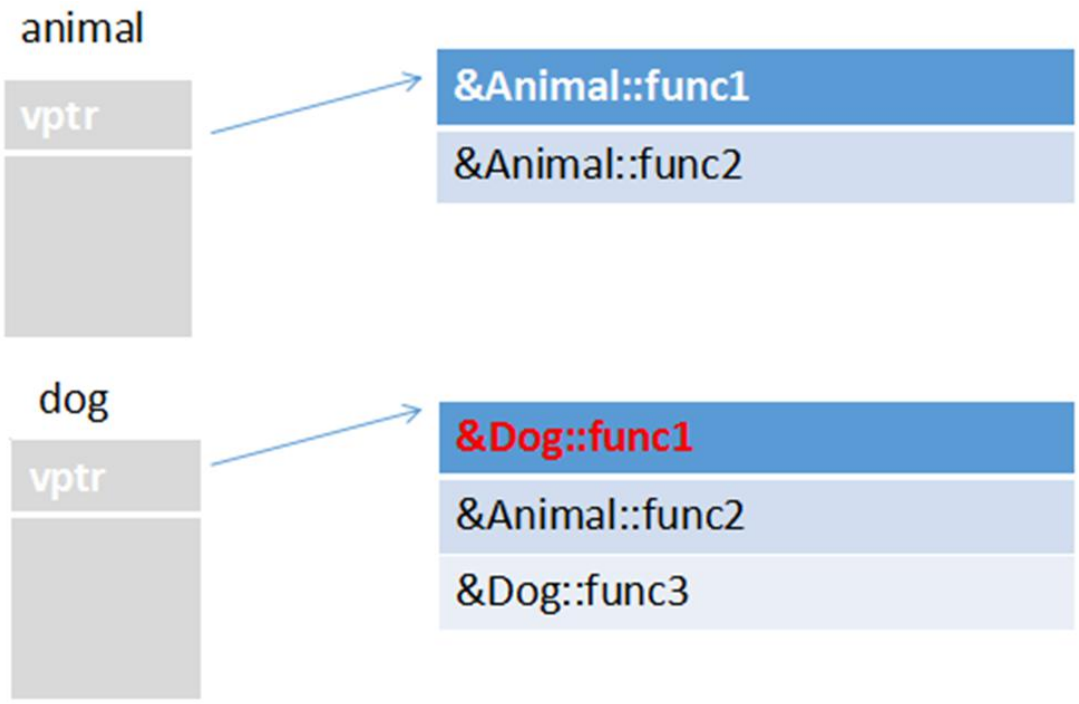




过程分析: Animal* animal = new Dog; animal->fun1(); 当程序执行到这里，会去 animal 指向的空间中寻找 vptr 指针，通过 vptr 指针找到 func1 函数，此时由于子类并没有重写也就是覆盖基类的 func1 函数，所以调用 func1 时，仍然调用的是基类的 func1. 执行结果: 我是基类的 func1 测试结论: 无重写基类的虚函数，无意义

当子类重写基类虚函数时:


Animal
+virtual void func1()
+virtual void func2()
Dog
+virtual void func1()
+virtual void func3()


animal
vptr
&Animal::func1
&Animal::func2
dog
vptr
&Dog::func1
&Animal::func2
&Dog::func3

过程分析: Animal* animal = new Dog; animal->fun1(); 当程序执行到这里，会去 animal 指向的空间中寻找 vptr 指针，通过 vptr 指针找到 func1 函数，由于子类重写基类的 func1 函数，所以调用 func1 时，调用的是子类的 func1. 执行结果: 我是子类的 func1 测试结论: 无重写基类的虚函数，无意义

多态的成立条件： 有继承 子类重写父类虚函数函数 a) 返回值，函数名字，函数参数，必须和父类完全一致(析构函数除外) b) 子类中 virtual 关键字可写可不写，建议写 类型兼容，父类指针，父类引用 指向 子类对象

4.8.4 抽象基类和纯虚函数(pure virtual function)

在设计时，常常希望基类仅仅作为其派生类的一个接口。这就是说，仅想对基类进行向上类型转换，使用它的接口，而不希望用户实际的创建一个基类的对象。同时创建一个纯虚函数允许接口中放置成员原函数，而不一定要提供一段可能对这个函数毫无意义的代码。

做到这点，可以在基类中加入至少一个纯虚函数(pure virtual function),使得基类称为抽象类(abstract class). 纯虚函数使用关键字 virtual，并在其后面加上=0。如果试图去实例化一个抽象类，编译器则会阻止这种操作。 当继承一个抽象类的时候，必须实现所有的纯虚函数，否则由抽象类派生的类也是一个抽象类。 Virtual void fun() = 0;告诉编译器在 vtable 中为函数保留一个位置，但在这个特定位置不放地址。

建立公共接口目的是为了将子类公共的操作抽象出来，可以通过一个公共接口来操纵一组类，且这个公共接口不需要事先(或者不需要完全实现)。可以创建一个公共类.

案例: 模板方法模式

1. 煮水
2. 冲泡咖啡
3. 倒入杯中
4. 加糖和牛奶

冲咖啡

1. 煮水
2. 冲泡茶叶
3. 倒入杯中
4. 加柠檬

冲茶叶

```
//抽象制作饮品
class AbstractDrinking{
public:
	//烧水
	virtual void Boil() = 0;
	//冲泡
	virtual void Brew() = 0;
	//倒入杯中
	virtual void PourInCup() = 0;
	//加入辅料
	virtual void PutSomething() = 0;
	//规定流程
	void MakeDrink(){
		this->Boil();
		Brew();
		PourInCup();
		PutSomething();
	}
};

//制作咖啡
class Coffee : public AbstractDrinking{
```

```
public:
	//烧水
	virtual void Boil(){
		cout << "煮农夫山泉!" << endl;
	}
	//冲泡
	virtual void Brew(){
		cout << "冲泡咖啡!" << endl;
	}
	//倒入杯中
	virtual void PourInCup(){
		cout << "将咖啡倒入杯中!" << endl;
	}
	//加入辅料
	virtual void PutSomething(){
		cout << "加入牛奶!" << endl;
	}
};

//制作茶水
class Tea : public AbstractDrinking{
public:
	//烧水
	virtual void Boil(){
		cout << "煮自来水!" << endl;
	}
	//冲泡
	virtual void Brew(){
		cout << "冲泡茶叶!" << endl;
	}
	//倒入杯中
	virtual void PourInCup(){
		cout << "将茶水倒入杯中!" << endl;
	}
	//加入辅料
	virtual void PutSomething(){
		cout << "加入食盐!" << endl;
	}
};

//业务函数
void DoBussiness(AbstractDrinking* drink){
	drink->MakeDrink();
	delete drink;
}

void test(){
	DoBussiness(new Coffee);
```

```
	cout << "--------------" << endl;
	DoBussiness(new Tea);
}
```

4.8.5 纯虚函数和多继承

多继承带来了一些争议，但是接口继承可以说一种毫无争议的运用了。 绝大数面向对象语言都不支持多继承，但是绝大数面向对象对象语言都支持接口的概念，c++中没有接口的概念，但是可以通过纯虚函数实现接口。

接口类中只有函数原型定义，没有任何数据定义。

多重继承接口不会带来二义性和复杂性问题。接口类只是一个功能声明，并不是功能实现，子类需要根据功能说明定义功能实现。 注意:除了析构函数外，其他声明都是纯虚函数。

4.8.6 虚析构函数

4.8.6.1 虚析构函数作用

虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。

```
class People{
public:
	People(){
		cout << "构造函数 People!" << endl;
	}
	virtual void showName() = 0;
	virtual ~People(){
		cout << "析构函数 People!" << endl;
	}
};

class Worker : public People{
public:
	Worker(){
		cout << "构造函数 Worker!" << endl;
		pName = new char[10];
	}
	virtual void showName(){
		cout << "打印子类的名字!" << endl;
	}
	~Worker(){
		cout << "析构函数 Worker!" << endl;
		if (pName != NULL){
			delete pName;
		}
```

```
	}
private:
	char* pName;
};

void test(){

	People* people = new Worker;
	people->~People();
}
```

4.8.6.2 纯虚析构函数

纯虚析构函数在 c++中是合法的，但是在使用的时候有一个额外的限制：必须为纯虚析构函数提供一个函数体。 那么问题是：如果给虚析构函数提供函数体了，那怎么还能称作纯虚析构函数呢？ 纯虚析构函数和非纯析构函数之间唯一的不同之处在于纯虚析构函数使得基类是抽象类，不能创建基类的对象。

```
//非纯虚析构函数
class A{
public:
	virtual ~A();
};

A::~A(){}

//纯析构函数
class B{
public:
	virtual ~B() = 0;
};

B::~B(){}

void test(){
	A a; //A 类不是抽象类，可以实例化对象
	B b; //B 类是抽象类，不可以实例化对象
}
```

如果类的目的不是为了实现多态，作为基类来使用，就不要声明虚析构函数，反之，则应该为类声明虚析构函数。

4.8.7 重写 重载 重定义

重载，同一作用域的同名函数

1. 同一个作用域

2. 参数个数，参数顺序，参数类型不同
3. 和函数返回值，没有关系
4. const 也可以作为重载条件 //do(const Teacher& t){} do(Teacher& t)

   重定义（隐藏）
5. 有继承
6. 子类（派生类）重新定义父类（基类）的同名成员（非 virtual 函数） 重写（覆盖）
7. 有继承
8. 子类（派生类）重写父类（基类）的 virtual 函数
9. 函数返回值，函数名字，函数参数，必须和基类中的虚函数一致

```
class A{
public:
	//同一作用域下，func1 函数重载
	void func1(){}
	void func1(int a){}
	void func1(int a,int b){}
	void func2(){}
	virtual void func3(){}
};

class B : public A{
public:
	//重定义基类的 func2,隐藏了基类的 func2 方法
	void func2(){}
	//重写基类的 func3 函数，也可以覆盖基类 func3
	virtual void func3(){}
};
```

5.C++模板

5.1 模板概论

c++提供了函数模板(function template.)所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不具体制定，用一个虚拟的类型来代表。这个通用函数就成为函数模板。凡是函数体相同的函数都可以用这个模板代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现不同函数的功能。 c++提供两种模板机制:函数模板和类模板 类属 - 类型参数化，又称参数模板 总结： 模板把函数或类要处理的数据类型

参数化，表现为参数的多态性，成为类属。 模板用于表达逻辑结构相同，但具体数据元素类型不同的数据对象的通用行为。

5.2 函数模板

5.2.1 什么是函数模板？

```
//交换 int 数据
void SwapInt(int& a,int& b){
	int temp = a;
	a = b;
	b = temp;
}

//交换 char 数据
void SwapChar(char& a,char& b){
	char temp = a;
	a = b;
	b = temp;
}
//问题：如果我要交换 double 类型数据，那么还需要些一个 double 类型数据交换的函数
//繁琐，写的函数越多，当交换逻辑发生变化的时候，所有的函数都需要修改，无形当中增加了代码的维护难度

//如果能把类型作为参数传递进来就好了，传递 int 就是 Int 类型交换，传递 char 就是 char 类型交换
//我们有一种技术，可以实现类型的参数化---函数模板


//class 和 typename 都是一样的，用哪个都可以
template<class T>
void MySwap(T& a,T& b){
	T temp = a;
	a = b;
	b = temp;
}

void test01(){

	int a = 10;
	int b = 20;
	cout << "a:" << a << " b:" << b << endl;
	//1. 这里有个需要注意点，函数模板可以自动推导参数的类型
	MySwap(a,b);
	cout << "a:" << a << " b:" << b << endl;

	char c1 = 'a';
```

```
	char c2 = 'b';
	cout << "c1:" << c1 << " c2:" << c2 << endl;
	//2. 函数模板可以自动类型推导，那么也可以显式指定类型
	MySwap<char>(c1, c2);
	cout << "c1:" << c1 << " c2:" << c2 << endl;
}
```

用模板是为了实现泛型，可以减轻编程的工作量，增强函数的重用性。

5.2.2 课堂练习

使用函数模板实现对 char 和 int 类型数组进行排序？

```
//模板打印函数
template<class T>
void PrintArray(T arr[],int len){
	for (int i = 0; i < len;i++){
		cout << arr[i] << " ";
	}
	cout << endl;
}

//模板排序函数
template<class T>
void MySort(T arr[],int len){

	for (int i = 0; i < len;i++){
		for (int j = len - 1; j > i;j--){
			if (arr[j] > arr[j - 1]){
				T temp = arr[j - 1];
				arr[j - 1] = arr[j];
				arr[j] = temp;
			}
		}
	}

}

void test(){

	//char 数组
	char tempChar[] = "aojtifysn";
	int charLen = strlen(tempChar);

	//int 数组
	int tempInt[] = {7,4,2,9,8,1};
	int intLen = sizeof(tempInt) / sizeof(int);

	//排序前 打印函数
```

```
	PrintArray(tempChar, charLen);
	PrintArray(tempInt, intLen);
	//排序
	MySort(tempChar, charLen);
	MySort(tempInt, intLen);
	//排序后打印
	PrintArray(tempChar, charLen);
	PrintArray(tempInt, intLen);
}
```

5.3 函数模板和普通函数区别

函数模板不允许自动类型转化 普通函数能够自动进行类型转化

```
//函数模板
template<class T>
T MyPlus(T a, T b){
	T ret = a + b;
	return ret;
}

//普通函数
int MyPlus(int a,char b){
	int ret = a + b;
	return ret;
}

void test02(){

	int a = 10;
	char b = 'a';

	//调用函数模板，严格匹配类型
	MyPlus(a, a);
	MyPlus(b, b);
	//调用普通函数
	MyPlus(a, b);
	//调用普通函数  普通函数可以隐式类型转换
	MyPlus(b, a);

	//结论：
	//函数模板不允许自动类型转换，必须严格匹配类型
	//普通函数可以进行自动类型转换
}
```

5.4 函数模板和普通函数在一起调用规则

c++编译器优先考虑普通函数 可以通过空模板实参列表的语法限定编译器只能通过模板匹配 函数模板可以像普通函数那样可以被重载 如果函数模板可以产生一个更好的匹配，那么选择模板

```
//函数模板
template<class T>
T MyPlus(T a, T b){
	T ret = a + b;
	return ret;
}

//普通函数
int MyPlus(int a, int b){
	int ret = a + b;
	return ret;
}

void test03(){
	int a = 10;
	int b = 20;
	char c = 'a';
	char d = 'b';
	//如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
	cout << MyPlus(a, b) << endl;
	//如果我必须要调用函数模板,那么怎么办?
	cout << MyPlus<>(a, b) << endl;
	//此时普通函数也可以匹配，因为普通函数可以自动类型转换
	//但是此时函数模板能够有更好的匹配
	//如果函数模板可以产生一个更好的匹配，那么选择模板
	cout << MyPlus(c,d);
}

//函数模板重载
template<class T>
T MyPlus(T a, T b, T c){
	T ret = a + b + c;
	return ret;
}

void test04(){

	int a = 10;
	int b = 20;
	int c = 30;
	cout << MyPlus(a, b, c) << endl;
```

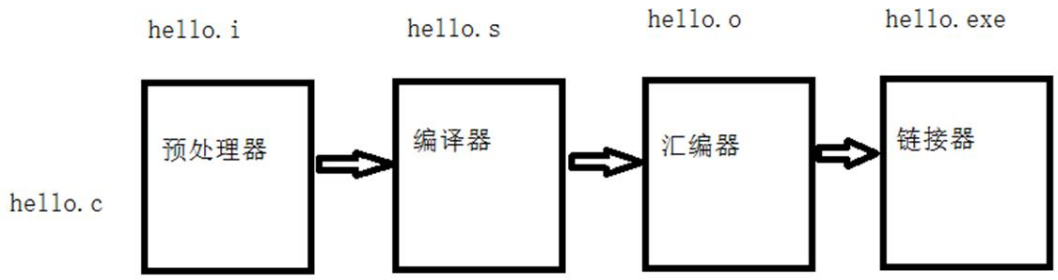
```
    //如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
}
```

5.5 模板机制剖析

思考:为什么函数模板可以和普通函数放在一起?c++编译器是如何实现函数模板机制的？

5.5.1 编译过程

hello.cpp 程序是高级 c 语言程序，这种程序易于被人读懂。为了在系统上运行 hello.c 程序，每一条 c 语句都必须转化为低级的机器指令。然后将这些机器指令打包成可执行目标文件格式，并以二进制形式存储于磁盘中。 预处理(Pre-processing) -> 编译(Compiling) ->汇编(Assembling) -> 链接(Linking)



hello.c经过预处理器，将宏展开，生成的文件hello.i
hello.i经过编译器，将文件编译成汇编语言，生成文件为hello.s
hello.s经过汇编器，将文件编译成目标文件hello.o（win下为hello.obj）
hello.o经过链接器，将文件编译成可执行文件

5.5.2 模板实现机制

函数模板机制结论： 编译器并不是把函数模板处理成能够处理任何类型的函数 函数模板通过具体类型产生不同的函数 编译器会对函数模板进行两次编译，在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的 代码进行编译。

1.6 模板的局限性

假设有如下模板函数：

```
template<class T>
void f(T a, T b)
{ … }
```

如果代码实现时定义了赋值操作 a = b，但是 T 为数组，这种假设就不成立了 同样，如果里面的语句为判断语句 if(a>b),但 T 如果是结构体，该假设也不成立，另外如果是传入的数组，数组名为地址，因此它比较的是地址，而这也不是我们所希望的操作。 总之，编写的模板函数很可能无法处理某些类型，另一方面，有时候通用化是有意义的，但 C++语法不允许这样做。为了解决这种问题，可以提供模板的重载，为这些特定的类型提供具体化的模板。

```
class Person
{
public:
	Person(string name, int age)
	{
		this->mName = name;
		this->mAge = age;
	}
	string mName;
	int mAge;
};

//普通交换函数
template <class T>
void mySwap(T &a,T &b)
{
	T temp = a;
	a = b;
	b = temp;
}
//第三代具体化，显示具体化的原型和定意思以 template<>开头，并通过名称来指出类型
//具体化优先于常规模板
template<>void  mySwap<Person>(Person &p1, Person &p2)
{
	string nameTemp;
	int ageTemp;

	nameTemp = p1.mName;
	p1.mName = p2.mName;
	p2.mName = nameTemp;

	ageTemp = p1.mAge;
	p1.mAge = p2.mAge;
	p2.mAge = ageTemp;

}

void test()
{
	Person P1("Tom", 10);
```

```
	Person P2("Jerry", 20);

	cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << en
dl;
	cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << en
dl;
	mySwap(P1, P2);
	cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << en
dl;
	cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << en
dl;
}
```

5.7 类模板

5.7.1 类模板基本概念

类模板和函数模板的定义和使用类似，我们已经进行了介绍。有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同。 类模板用于实现类所需数据的类型参数化

```
template<class NameType, class AgeType>
class Person
{
public:
	Person(NameType name, AgeType age)
	{
		this->mName = name;
		this->mAge = age;
	}
	void showPerson()
	{
		cout << "name: " << this->mName << " age: " << this->mAge <
< endl;
	}
public:
	NameType mName;
	AgeType mAge;
};

void test01()
{
	//Person P1("德玛西亚",18); // 类模板不能进行类型自动推导
	Person<string, int>P1("德玛西亚", 18);
	P1.showPerson();
}
```

5.7.2 类模板做函数参数

```
/类模板
template<class NameType, class AgeType>
class Person{
public:
	Person(NameType name, AgeType age){
		this->mName = name;
		this->mAge = age;
	}
	void PrintPerson(){
		cout << "Name:" << this->mName << " Age:" << this->mAge <<
endl;
	}
public:
	NameType mName;
	AgeType mAge;
};

//类模板做函数参数
void DoBussiness(Person<string,int>& p){
	p.mAge += 20;
	p.mName += "_vip";
	p.PrintPerson();
}

int main(){

	Person<string, int> p("John", 30);
	DoBussiness(p);

	system("pause");
	return EXIT_SUCCESS;
}
```

5.7.3 类模板派生普通类

```
//类模板
template<class T>
class MyClass{
public:
	MyClass(T property){
		this->mProperty = property;
	}
public:
	T mProperty;
};

//子类实例化的时候需要具体化的父类，子类需要知道父类的具体类型是什么样的
//这样 c++编译器才能知道给子类分配多少内存
```

```
//普通派生类
class SubClass : public MyClass<int>{
public:
	SubClass(int b) : MyClass<int>(20){
		this->mB = b;
	}
public:
	int mB;
};
```

1.7.4 类模板派生类模板

```
//父类类模板
template<class T>
class Base
{
	T m;
};
template<class T >
class Child2 : public Base<double>  //继承类模板的时候，必须要确定基类的大小
{
public:
	T mParam;
};

void test02()
{
	Child2<int> d2;
}
```

5.7.5 类模板类内实现

```
template<class NameType, class AgeType>
class Person
{
public:
	Person(NameType name, AgeType age)
	{
		this->mName = name;
		this->mAge = age;
	}
	void showPerson()
	{
		cout << "name: " << this->mName << " age: " << this->mAge << endl;
	}
public:
	NameType mName;
```

```
	AgeType mAge;
};

void test01()
{
	//Person P1("德玛西亚",18); // 类模板不能进行类型自动推导
	Person<string, int>P1("德玛西亚", 18);
	P1.showPerson();
}
```

5.7.6 类模板类外实现

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
using namespace std;

template<class T1, class T2>
class Person{
public:
	Person(T1 name, T2 age);
	void showPerson();

public:
	T1 mName;
	T2 mAge;
};


//类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age){
	this->mName = name;
	this->mAge = age;
}


template<class T1, class T2>
void Person<T1, T2>::showPerson(){
	cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
}

void test()
{
	Person<string, int> p("Obama", 20);
	p.showPerson();
}

int main(){
```

```
	test();

	system("pause");
	return EXIT_SUCCESS;
}
```

5.7.7 类模板头文件和源文件分离问题

Person.hpp

```
#pragma once

template<class T1,class T2>
class Person{
public:
	Person(T1 name,T2 age);
	void ShowPerson();
public:
	T1 mName;
	T2 mAge;
};

template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age){
	this->mName = name;
	this->mAge = age;
}

template<class T1, class T2>
void Person<T1, T2>::ShowPerson(){
	cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
}
```

main.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include<string>
#include"Person.hpp"

//模板二次编译
//编译器编译源码 逐个编译单元编译的

int main(){

	Person<string, int> p("Obama", 20);
	p.ShowPerson();
```

```
	system("pause");
	return EXIT_SUCCESS;
}
```

结论: 案例代码在 qt 编译器顺利通过编译并执行，但是在 Linux 和 vs 编辑器下如果只包含头文件，那么会报错链接错误，需要包含 cpp 文件，但是如果类模板中有友元类，那么编译失败！ 解决方案: 类模板的声明和实现放到一个文件中，我们把这个文件命名为.hpp(这个是个约定的规则，并不是标准，必须这么写). 原因：类模板需要二次编译，在出现模板的地方编译一次，在调用模板的地方再次编译。C++编译规则为独立编译。

5.7.8 模板类碰到友元函数

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include <string>

template<class T1, class T2> class Person;
//告诉编译器这个函数模板是存在
template<class T1, class T2> void PrintPerson2(Person<T1, T2>& p);

//友元函数在类内实现
template<class T1, class T2>
class Person{
	//1. 友元函数在类内实现
	friend void PrintPerson(Person<T1, T2>& p){
		cout << "Name:" << p.mName << " Age:" << p.mAge << endl;
	}

	//2.友元函数类外实现
	//告诉编译器这个函数模板是存在
	friend void PrintPerson2<>(Person<T1, T2>& p);

	//3. 类模板碰到友元函数模板
	template<class U1, class U2>
	friend void PrintPerson(Person<U1, U2>& p);

public:
	Person(T1 name, T2 age){
		this->mName = name;
		this->mAge = age;
	}
	void showPerson(){
		cout << "Name:" << this->mName << " Age:" << this->mAge <<
endl;
	}
```

```
private:
	T1 mName;
	T2 mAge;
};

void test01()
{
	Person <string, int>p("Jerry", 20);
	PrintPerson(p);
}


// 类模板碰到友元函数
//友元函数类外实现  加上<>空参数列表，告诉编译去匹配函数模板
template<class T1 , class T2>
void PrintPerson2(Person<T1, T2>& p)
{
	cout << "Name2:" << p.mName << " Age2:" << p.mAge << endl;
}

void  test02()
{
	Person <string, int>p("Jerry", 20);
	PrintPerson2(p);   //不写可以编译通过，写了之后，会找 PrintPerson2 的普通函数调用，因为写了普通函数 PrintPerson2 的声明
}

int main(){

	//test01();
	test02();
	system("pause");
	return EXIT_SUCCESS;
}
```

5.8 类模板的应用

设计一个数组模板类(MyArray),完成对不同类型元素的管理

```
#pragma once
template<class T>
class MyArray
{
public:
	explicit MyArray(int capacity)
	{
		this->m_Capacity = capacity;
		this->m_Size = 0;
		// 如果 T 是对象，那么这个对象必须提供默认的构造函数
```

```
        pAddress = new T[this->m_Capacity];
}

//拷贝构造
MyArray(const MyArray & arr)
{
        this->m_Capacity = arr.m_Capacity;
        this->m_Size = arr.m_Size;
        this->pAddress = new T[this->m_Capacity];
        for (int i = 0; i < this->m_Size;i++)
        {
                this->pAddress[i] = arr.pAddress[i];
        }
}

//重载[] 操作符  arr[0]
T& operator [](int index)
{
        return this->pAddress[index];
}
//尾插法
void Push_back(const T & val)
{
        if (this->m_Capacity == this->m_Size)
        {
                return;
        }
        this->pAddress[this->m_Size] = val;
        this->m_Size++;
}
void Pop_back()
{
        if (this->m_Size == 0)
        {
                return;
        }
        this->m_Size--;
}
int   getSize()
{
        return this->m_Size;
}
//析构
~MyArray()
{
        if (this->pAddress != NULL)
        {
                delete[] this->pAddress;
                this->pAddress = NULL;
```

```
                this->m_Capacity = 0;
                this->m_Size = 0;
            }
        }

private:
        T * pAddress;  //指向一个堆空间，这个空间存储真正的数据
        int m_Capacity; //容量
        int m_Size;   // 大小
};
```

测试代码：

```
class Person{
public:
        Person(){}
        Person(string name, int age){
                this->mName = name;
                this->mAge = age;
        }
public:
        string mName;
        int mAge;
};


void PrintMyArrayInt(MyArray<int>& arr){
        for (int i = 0; i < arr.getSize(); i++){
                cout << arr[i] << " ";
        }
        cout << endl;
}

void PrintMyPerson(MyArray<Person>& personArr)
{
        for (int i = 0; i < personArr.getSize(); i++){
                cout << "姓名: " << personArr[i].mName << " 年龄:  " << pers
onArr[i].mAge << endl;
        }

}
void test01()
{
        MyArray<int> myArrayInt(10);
        for (int i = 0; i < 9; i++)
        {
                myArrayInt.Push_back(i);
        }
```

```
        myArrayInt.Push_back(100);
        PrintMyArrayInt(myArrayInt);
        MyArray<Person> myArrayPerson(10);
        Person p1("德玛西亚", 30);
        Person p2("提莫", 20);
        Person p3("孙悟空",18);
        Person p4("赵信", 15);
        Person p5("赵云", 24);
        myArrayPerson.Push_back(p1);
        myArrayPerson.Push_back(p2);
        myArrayPerson.Push_back(p3);
        myArrayPerson.Push_back(p4);
        myArrayPerson.Push_back(p5);
}
```

6.C++类型转换

类型转换(cast)是将一种数据类型转换成另一种数据类型。例如，如果将一个整型值赋给一个浮点类型的变量，编译器会暗地里将其转换成浮点类型。 转换是非常有用的，但是它也会带来一些问题，比如在转换指针时，我们很可能将其转换成一个比它更大的类型，但这可能会破坏其他的数据。 应该小心类型转换，因为转换也就相当于对编译器说：忘记类型检查，把它看做其他的类型。 一般情况下，尽量少的去使用类型转换，除非用来解决非常特殊的问题。

无论什么原因，任何一个程序如果使用很多类型转换都值得怀疑.

标准 c++提供了一个显示的转换的语法，来替代旧的 C 风格的类型转换。 使用 C 风格的强制转换可以把想要的任何东西转换成我们需要的类型。那为什么还需要一个新的 C++类型的强制转换呢？ 新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。C++风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。程序员只要扫一眼这样的代码，就能立即知道一个强制转换的目的。

6.1 静态转换(static_cast)

用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的； 进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的。 用于基本数据类型之间的转换，如把 int 转换成 char，把 char 转换成 int。这种转换的安全性也要开发人员来保证

```
class Animal{};
class Dog : public Animal{};
class Other{};
```

```
//基础数据类型转换
void test01(){
	char a = 'a';
	double b = static_cast<double>(a);
}

//继承关系指针互相转换
void test02(){
	//继承关系指针转换
	Animal* animal01 = NULL;
	Dog* dog01 = NULL;
	//子类指针转成父类指针,安全
	Animal* animal02 = static_cast<Animal*>(dog01);
	//父类指针转成子类指针，不安全
	Dog* dog02 = static_cast<Dog*>(animal01);
}

//继承关系引用相互转换
void test03(){

	Animal ani_ref;
	Dog dog_ref;
	//继承关系指针转换
	Animal& animal01 = ani_ref;
	Dog& dog01 = dog_ref;
	//子类指针转成父类指针,安全
	Animal& animal02 = static_cast<Animal&>(dog01);
	//父类指针转成子类指针，不安全
	Dog& dog02 = static_cast<Dog&>(animal01);
}

//无继承关系指针转换
void test04(){

	Animal* animal01 = NULL;
	Other* other01 = NULL;

	//转换失败
	//Animal* animal02 = static_cast<Animal*>(other01);
}
```

6.2 动态转换(dynamic_cast)

dynamic*cast 主要用于类层次间的上行转换和下行转换； 在类层次间进行上行转换时，dynamic*cast 和 static*cast 的效果是一样的； 在进行下行转换时，dynamic*cast 具有类型检查的功能，比 static_cast 更安全；

```
class Animal {
public:
	virtual void ShowName() = 0;
};
class Dog : public Animal{
	virtual void ShowName(){
		cout << "I am a dog!" << endl;
	}
};
class Other {
public:
	void PrintSomething(){
		cout << "我是其他类!" << endl;
	}
};

//普通类型转换
void test01(){

	//不支持基础数据类型
	int a = 10;
	//double a = dynamic_cast<double>(a);
}

//继承关系指针
void test02(){

	Animal* animal01 = NULL;
	Dog* dog01 = new Dog;

	//子类指针转换成父类指针 可以
	Animal* animal02 = dynamic_cast<Animal*>(dog01);
	animal02->ShowName();
	//父类指针转换成子类指针 不可以
	//Dog* dog02 = dynamic_cast<Dog*>(animal01);
}

//继承关系引用
void test03(){

	Dog dog_ref;
	Dog& dog01 = dog_ref;

	//子类引用转换成父类引用 可以
	Animal& animal02 = dynamic_cast<Animal&>(dog01);
	animal02.ShowName();
}
```

```
//无继承关系指针转换
void test04(){

	Animal* animal01 = NULL;
	Other* other = NULL;

	//不可以
	//Animal* animal02 = dynamic_cast<Animal*>(other);
}
```

6.3 常量转换(const_cast)

该运算符用来修改类型的 const 属性。。 常量指针被转化成非常量指针，并且仍然指向原来的对象； 常量引用被转换成非常量引用，并且仍然指向原来的对象；

注意:不能直接对非指针和非引用的变量使用 const_cast 操作符去直接移除它的 const.

```
//常量指针转换成非常量指针
void test01(){

	const int* p = NULL;
	int* np = const_cast<int*>(p);

	int* pp = NULL;
	const int* npp = const_cast<const int*>(pp);

	const int a = 10;  //不能对非指针或非引用进行转换
	//int b = const_cast<int>(a); }

//常量引用转换成非常量引用
void test02(){

int num = 10;
	int & refNum = num;

	const int& refNum2 = const_cast<const int&>(refNum);

}
```

6.4 重新解释转换(reinterpret_cast)

这是最不安全的一种转换机制，最有可能出问题。 主要用于将一种数据类型从一种类型转换为另一种类型。它可以将一个指针转换成一个整数，也可以将一个整数转换成一个指针.

7.C++异常

7.1 异常基本概念

Bjarne Stroustrup 说：提供异常的基本目的就是为了处理上面的问题。基本思想是：让一个函数在发现了自己无法处理的错误时抛出（throw）一个异常，然后它的（直接或者间接）调用者能够处理这个问题。也就是《C++ primer》中说的：将问题检测和问题处理相分离。 一种思想：在所有支持异常处理的编程语言中（例如 java），要认识到的一个思想：在异常处理过程中，由问题检测代码可以抛出一个对象给问题处理代码，通过这个对象的类型和内容，实际上完成了两个部分的通信，通信的内容是“出现了什么错误”。当然，各种语言对异常的具体实现有着或多或少的区别，但是这个通信的思想是不变的。

一句话：异常处理就是处理程序中的错误。所谓错误是指在程序运行的过程中发生的一些异常事件（如：除 0 溢出，数组下标越界，所要读取的文件不存在,空指针，内存不足等等）。

回顾一下：我们以前编写程序是如何处理异常？ 在 C 语言的世界中，对错误的处理总是围绕着两种方法：一是使用整型的返回值标识错误；二是使用 errno 宏（可以简单的理解为一个全局整型变量）去记录错误。当然 C++中仍然是可以用这两种方法的。 这两种方法最大的缺陷就是会出现不一致问题。例如有些函数返回 1 表示成功，返回 0 表示出错；而有些函数返回 0 表示成功，返回非 0 表示出错。 还有一个缺点就是函数的返回值只有一个，你通过函数的返回值表示错误代码，那么函数就不能返回其他的值。当然，你也可以通过指针或者 C++的引用来返回另外的值，但是这样可能会令你的程序略微晦涩难懂。

c++异常机制相比 C 语言异常处理的优势？ 函数的返回值可以忽略，但异常不可忽略。如果程序出现异常，但是没有被捕获，程序就会终止，这多少会促使程序员开发出来的程序更健壮一点。而如果使用 C 语言的 error 宏或者函数返回值，调用者都有可能忘记检查，从而没有对错误进行处理，结果造成程序莫名其面的终止或出现错误的结果。 整型返回值没有任何语义信息。而异常却包含语义信息，有时你从类名就能够体现出来。 整型返回值缺乏相关的上下文信息。异常作为一个类，可以拥有自己的成员，这些成员就可以传递足够的信息。 异常处理可以在调用跳级。这是一个代码编写时的问题：假设在有多个函数的调用栈中出现了某个错误，使用整型返回码要求你在每一级函数中都要进行处理。而使用异常处理的栈展开机制，只需要在一处进行处理就可以了，不需要每级函数都处理。

/如果判断返回值，那么返回值是错误码还是结果？

```
//如果不判断返回值，那么 b==0 时候，程序结果已经不正确
//A 写的代码
int A_MyDivide(int a,int b){
	if (b == 0){
		return -1;
	}
```

```
	return a / b;
}

//B 写的代码
int B_MyDivide(int a,int b){

	int ba = a + 100;
	int bb = b;

	int ret = A_MyDivide(ba, bb);  //由于B 没有处理异常，导致B 结果运算错误

	return ret;
}

//C 写的代码
int C_MyDivide(){

	int a = 10;
	int b = 0;

	int ret = B_MyDivide(a, b); //更严重的是，由于B 没有继续抛出异常，导致C 的代码没有办法捕获异常
	if (ret == -1){
		return -1;
	}
	else{
		return ret;
	}
}

//所以,我们希望:
//1.异常应该捕获，如果你捕获，可以，那么异常必须继续抛给上层函数,你不处理，不代表你的上层不处理
//2.这个例子，异常没有捕获的结果就是运行结果错的一塌糊涂，结果未知，未知的结果程序没有必要执行下去
```

7.2 异常语法

7.2.1 异常基本语法

```
int A_MyDivide(int a, int b){
	if (b == 0){
		throw 0;
	}

	return a / b;
```

```
}

//B 写的代码 B 写代码比较粗心，忘记处理异常
int B_MyDivide(int a, int b){

	int ba = a;
	int bb = b;

	int ret = A_MyDivide(ba, bb) + 100;  //由于B 没有处理异常，导致B 结果运算错误

	return ret;
}

//C 写的代码
int C_MyDivide(){

	int a = 10;
	int b = 0;

	int ret = 0;

//没有处理异常，程序直接中断执行
#if 1
	ret = B_MyDivide(a, b);

//处理异常
#else
	try{
		ret = B_MyDivide(a, b); //更严重的是，由于B 没有继续抛出异常，导致C 的代码没有办法捕获异常
	}
	catch (int e){
		cout << "C_MyDivide Call B_MyDivide 除数为:" << e << endl;
	}
#endif

	return ret;
}

int main(){

	C_MyDivide();

	system("pause");
	return EXIT_SUCCESS;
}
```

总结: 若有异常则通过 throw 操作创建一个异常对象并抛出。 将可能抛出异常的程序段放到 try 块之中。 如果在 try 段执行期间没有引起异常，那么跟在 try 后面的 catch 字句就不会执行。 catch 子句会根据出现的先后顺序被检查，匹配的 catch 语句捕获并处理异常(或继续抛出异常) 如果匹配的处理未找到，则运行函数 terminate 将自动被调用，其缺省功能调用 abort 终止程序。 处理不了的异常，可以在 catch 的最后一个分支，使用 throw，向上抛。

c++异常处理使得异常的引发和异常的处理不必在一个函数中，这样底层的函数可以着重解决具体问题，而不必过多的考虑异常的处理。上层调用者可以在适当的位置设计对不同类型异常的处理。

7.2.2 异常严格类型匹配

异常机制和函数机制互不干涉,但是捕捉方式是通过严格类型匹配。

```
void TestFunction(){

	cout << "开始抛出异常..." << endl;
	//throw 10; //抛出 int 类型异常
	//throw 'a'; //抛出 char 类型异常
	//throw "abcd"; //抛出 char*类型异常
	string ex = "string exception!";
	throw ex;

}

int main(){

	try{
		TestFunction();
	}
	catch (int){
		cout << "抛出 Int 类型异常!" << endl;
	}
	catch (char){
		cout << "抛出 Char 类型异常!" << endl;
	}
	catch (char*){
		cout << "抛出 Char*类型异常!" << endl;
	}
	catch (string){
		cout << "抛出 string 类型异常!" << endl;
	}
	//捕获所有异常
	catch (...){
		cout << "抛出其他类型异常!" << endl;
	}
```

```
	system("pause");
	return EXIT_SUCCESS;
}
```

7.2.3 栈解旋(unwinding)

异常被抛出后，从进入 try 块起，到异常被抛掷前，这期间在栈上构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反，这一过程称为栈的解旋(unwinding).

```
public:
	Person(string name){
		mName = name;
		cout << mName << "对象被创建!" << endl;
	}
	~Person(){
		cout << mName << "对象被析构!" << endl;
	}
public:
	string mName;
};

void TestFunction(){

	Person p1("aaa");
	Person p2("bbb");
	Person p3("ccc");

	//抛出异常
	throw 10;
}

int main(){

	try{
		TestFunction();
	}
	catch (...){
		cout << "异常被捕获!" << endl;
	}

	system("pause");
	return EXIT_SUCCESS;
}
```

7.2.4 异常接口声明

为了加强程序的可读性，可以在函数声明中列出可能抛出异常的所有类型，例如：void func() throw(A,B,C);这个函数 func 能够且只能抛出类型 A,B,C 及其子类型的异常。 如果在函数声明中没有包含异常接口声明，则此函数可以抛任何类型的异常，例如:void func() 一个不抛任何类型异常的函数可声明为:void func() throw() 如果一个函数抛出了它的异常接口声明所不允许抛出的异常,unexcepted 函数会被调用，该函数默认行为调用 terminate 函数中断程序。

```
//可抛出所有类型异常
void TestFunction01(){
	throw 10;
}

//只能抛出 int char char*类型异常
void TestFunction02() throw(int,char,char*){
	string exception = "error!";
	throw exception;
}

//不能抛出任何类型异常
void TestFunction03() throw(){
	throw 10;
}

int main(){

	try{
		//TestFunction01();
		//TestFunction02();
		//TestFunction03();
	}
	catch (...){
		cout << "捕获异常!" << endl;
	}

	system("pause");
	return EXIT_SUCCESS;
}
```

请分别在 qt vs linux 下做测试! Qt and Linux 正确!

7.2.5 异常变量生命周期

throw 的异常是有类型的，可以是数字、字符串、类对象。 throw 的异常是有类型的，catch 需严格匹配异常类型。

```
class MyException
{
public:
	MyException(){
		cout << "异常变量构造" << endl;
	};
	MyException(const MyException & e)
	{
		cout << "拷贝构造" << endl;
	}
	~MyException()
	{
		cout << "异常变量析构" << endl;
	}
};
void DoWork()
{
	throw new MyException(); //test1 2 都用 throw MyExecption();
}

void test01()
{
	try
	{
		DoWork();
	}
	catch (MyException e)
	{
		cout << "捕获 异常" << endl;
	}
}
void test02()
{
	try
	{
		DoWork();
	}
	catch (MyException &e)
	{
		cout << "捕获 异常" << endl;
	}
}

void test03()
{
	try
	{
		DoWork();
```

```
	}
	catch (MyException *e)
	{
		cout << "捕获 异常" << endl;
		delete e;
	}
}
```

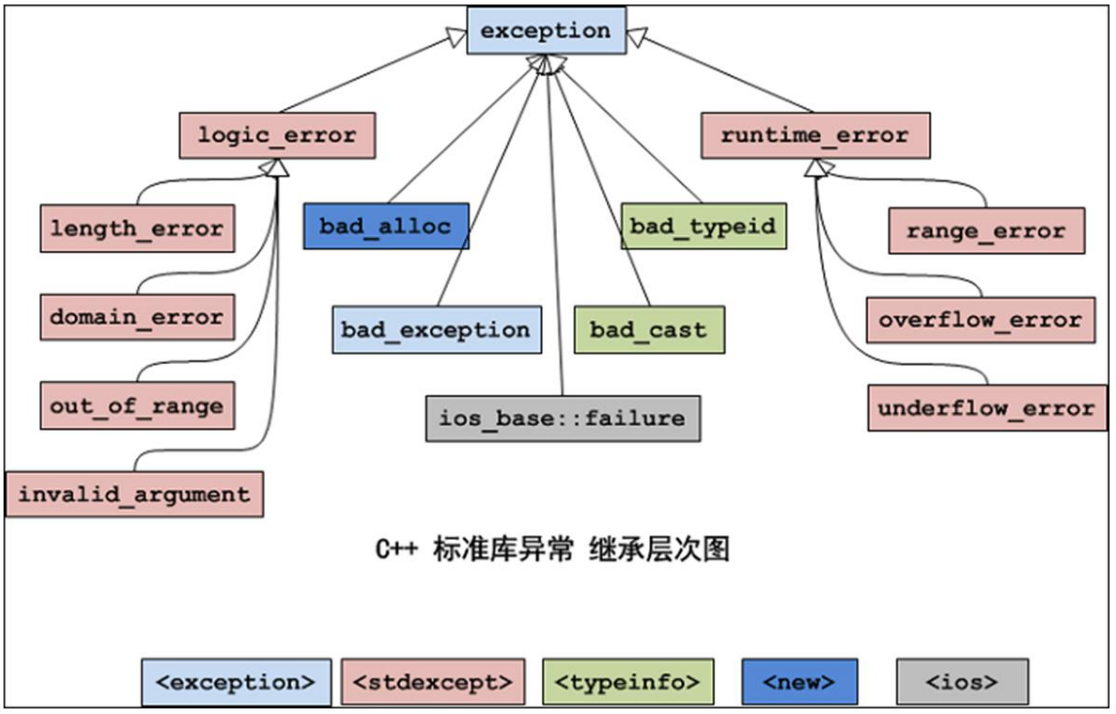
7.2.6 异常的多态使用

```
//异常基类
class BaseException{
public:
	virtual void printError(){};
};

//空指针异常
class NullPointerException : public BaseException{
public:
	virtual void printError(){
		cout << "空指针异常!" << endl;
	}
};
//越界异常
class OutOfRangeException : public BaseException{
public:
	virtual void printError(){
		cout << "越界异常!" << endl;
	}
};

void doWork(){

	throw NullPointerException();
}

void test()
{
	try{
		doWork();
	}
	catch (BaseException& ex){
		ex.printError();
	}
}
```

7.3 C++标准异常库

7.3.1 标准库介绍

标准库中也提供了很多的异常类，它们是通过类继承组织起来的。异常类继承层级结构图如下：



每个类所在的头文件在图下方标识出来。 标准异常类的成员： ① 在上述继承体系中，每个类都有提供了构造函数、复制构造函数、和赋值操作符重载。 ② logic*error 类及其子类*、*runtime*error 类及其子类，它们的构造函数是接受一个 string 类型的形式参数，用于异常信息的描述 ③ 所有的异常类都有一个 what()方法，返回 const char* 类型（C 风格字符串）的值，描述异常信息。

标准异常类的具体描述：

异常名称	描述
exception	所有标准异常类的父类
bad_alloc	当 operator new and operator new[]，请求分配内存失败时
bad_exception	这是个特殊的异常，如果函数的异常抛出列表里声明了 bad*exception 异常，当函数内部抛出了异常抛出列表中没有的*

	*异常，这是调用的 unexpected 函数中若抛出异常，不论什么类型，都会被替换为 bad*exception 类型
bad_typeid	使用 typeid 操作符，操作一个 NULL 指针，而该指针是带有虚函数的类，这时抛出 bad_typeid 异常
bad_cast	使用 dynamic_cast 转换引用失败的时候
ios_base::failure	io 操作过程出现错误
logic_error	逻辑错误，可以在运行前检测的错误
runtime_error	运行时错误，仅在运行时才可以检测的错误

logic_error 的子类：

异常名称	描述
length_error	试图生成一个超出该类型最大长度的对象时，例如 vector 的 resize 操作
domain_error	参数的值域错误，主要用在数学函数中。例如使用一个负值调用只能操作非负数的函数
out*of*range	超出有效范围
invalid_argument	参数不合适。在标准库中，当利用 string 对象构造 bitset 时，而 string 中的字符不是'0'或'1'的时候，抛出该异常

runtime_error 的子类：

异常名称	描述
range_error	计算结果超出了有意义的值域范围
overflow_error	算术计算上溢
underflow_error	算术计算下溢
invalid_argument	参数不合适。在标准库中，当利用 string 对象构造 bitset 时，而 string 中的字符不是'0'或'1'的时候，抛出该异常

```
#include<stdexcept>
class Person{
public:
	Person(int age){
		if (age < 0 || age > 150){
			throw out_of_range("年龄应该在 0-150 岁之间!");
```

```
		}
	}
public:
	int mAge;
};

int main(){

	try{
		Person p(151);
	}
	catch (out_of_range& ex){
		cout << ex.what() << endl;
	}

	system("pause");
	return EXIT_SUCCESS;
}
```

7.3.2 编写自己的异常类

① 标准库中的异常是有限的；② 在自己的异常类中，可以添加自己的信息。（标准库中的异常类值允许设置一个用来描述异常的字符串）。

如何编写自己的异常类？① 建议自己的异常类要继承标准异常类。因为 C++中可以抛出任何类型的异常，所以我们的异常类可以不继承自标准异常，但是这样可能会导致程序混乱，尤其是当我们多人协同开发时。② 当继承标准异常类时，应该重载父类的 what 函数和虚析构函数。③ 因为栈展开的过程中，要复制异常类型，那么要根据你在类中添加的成员考虑是否提供自己的复制构造函数。

```
//自定义异常类
class MyOutOfRange:public exception
{
public:
	MyOutOfRange(const string  errorInfo)
	{
		this->m_Error = errorInfo;
	}

	MyOutOfRange(const char * errorInfo)
	{
		this->m_Error = string( errorInfo);
	}

	virtual  ~MyOutOfRange()
	{

	}
```

```
	virtual const char *  what() const
	{
		return this->m_Error.c_str() ;
	}

	string m_Error;

};

class Person
{
public:
	Person(int age)
	{
		if (age <= 0 || age > 150)
		{
			//抛出异常 越界
			//cout << "越界" << endl;
			//throw  out_of_range("年龄必须在0~150之间");

			//throw length_error("长度异常");
			throw MyOutOfRange(("我的异常 年龄必须在0~150之间"));
		}
		else
		{
			this->m_Age = age;
		}

	}

	int m_Age;
};


void test01()
{
	try
	{
		Person p(151);
	}
	catch ( out_of_range & e )
	{
		cout << e.what() << endl;
	}
	catch (length_error & e)
	{
		cout << e.what() << endl;
	}
	catch (MyOutOfRange e)
```

```
	{
		cout << e.what() << endl;
	}
}

//cin.get
void test01(){
#if 0
	char ch = cin.get();
	cout << ch << endl;

	cin.get(ch);
	cout << ch << endl;


	//链式编程
	char char1, char2, char3, char4;
	cin.get(char1).get(char2).get(char3).get(char4);

	cout << char1 << " " << char2 << "" << char3 <<  " " << char4 <<
" ";
#endif

	char buf[1024] = { 0 };
	//cin.get(buf.1024);
	cin.getline(buf,1024);
	cout << buf;
}

//cin.ignore
void test02(){

	char buf[1024] = { 0 };
	cin.ignore(2); //忽略缓冲区当前字符
	cin.get(buf,1024);
	cout << buf << endl;
}

//cin.putback 将数据放回缓冲区
void test03(){

	//从缓冲区取走一个字符
	char ch = cin.get();
	cout << "从缓冲区取走的字符:" << ch << endl;
	//将数据再放回缓冲区
	cin.putback(ch);
	char buf[1024] = { 0 };
	cin.get(buf,1024);
	cout << buf << endl;
```

```
}

//cin.peek 偷窥
void test04(){

	//偷窥下缓冲区的数据
	char ch = cin.peek();
	cout << "偷窥缓冲区数据:" << ch << endl;
	char buf[1024] = { 0 };
	cin.get(buf, 1024);
	cout << buf << endl;
}

//练习  作业 使用cin.get 和putback 完成类似功能
void test05(){

	cout << "请输入一个数字或者字符串:" << endl;
	char ch = cin.peek();
	if(ch >= '0' && ch <= '9'){
		int number;
		cin >> number;
		cout << "数字:" << number << endl;
	}
	else{
		char buf[64] = { 0 };
		cin.getline(buf, 64);
		cout << "字符串:" <<  buf << endl;
	}
}
```