

传智播客 C++ 课程讲义

传智扫地僧

1、C++ 对 C 的扩展

1 简单的 C++ 程序

1.1 求圆的周长和面积

数据描述：

半径，周长，面积均用实型数表示

数据处理：

输入半径 r ；

计算周长 $= 2 * \pi * r$ ；

计算面积 $= \pi * r^2$ ；

输出半径，周长，面积；

方法 1：用结构化方法编程，求圆的周长和面积

```
// count the girth and area of circle
#include<iostream.h>
using name std;
void main ()
{ double r, girth, area ;
  const double PI = 3.1415 ;
  cout << "Please input radius:\n" ; //操作符重载
  cin >> r ; //输入
  girth = 2 * PI * r ;
  area = PI * r * r ;
  cout << "radius = " << r << endl ;
  cout << "girth = " << girth << endl ;
  cout << "area = " << area << endl ;
}
```

方法 2：用面向对象方法编程，求圆的周长和面积

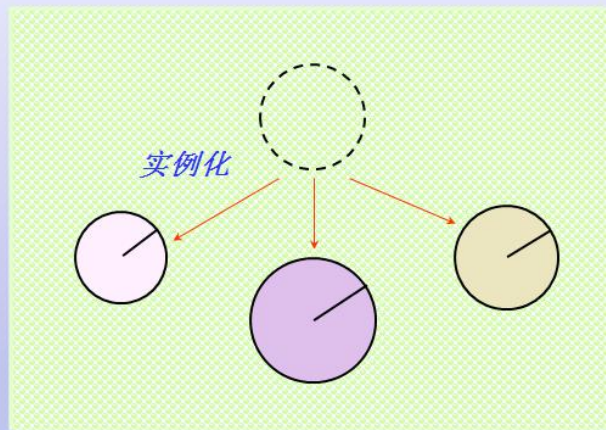
分析

“圆”是抽象的类类型

半径?

建立具体的圆 (对象)

圆的周长?
面积?



分析

圆类

成员变量

半径

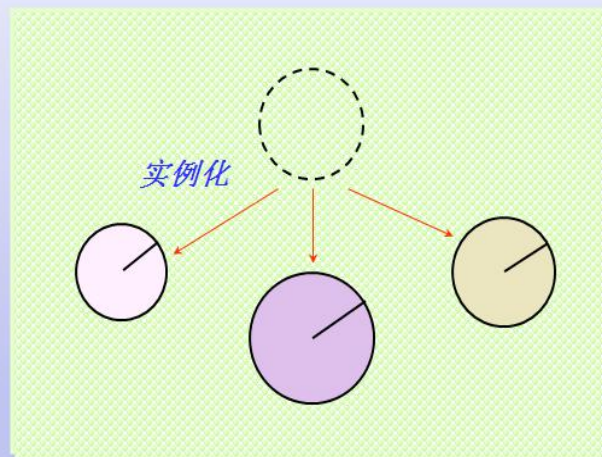
成员函数

置半径值

求圆的半径

求周长

求面积



```
#include<iostream.h>
using name std;
class Circle
{
    double radius ; //成员变量
public : //类的访问控制
    void Set_Radius( double r ) { radius = r ; } //成员函数
    double Get_Radius() { return  radius ; } //通过成员函数设置成员变量
    double Get_Girth()      { return  2 * 3.14f * radius ; } //通过成员函数获取成员变量
    double Get_Area()       { return  3.14f * radius * radius ; }
};

void main()
{
    Circle A, B ; //用类定义对象
    A.Set_Radius( 6.23 ) ; //类的调用
    cout << "A.Radius = " << A.Get_Radius() << endl ;
}
```

```
cout << "A.Girth = " << A.Get_Girth() << endl ;
cout << "A.Area = " << A.Get_Area() << endl ;
B.Set_Radius( 10.5 ) ;
cout << "B.radius = " << B.Get_Radius() << endl ;
cout << "B.Girth=" << B.Get_Girth() << endl ;
cout << "B.Area = " << B.Get_Area() << endl ;
}
```

总结：建立类、对象、成员变量、成员函数，输入输出流基本概念。

1.2 初学者易犯错误模型

```
// demo02_circle_err.cpp
#include<iostream>
using namespace std;//c++的命名空间
class circle
{
public:

    double r;

    double pi = 3.1415926;
    double area = pi*r*r;

};

int main()
{
    circle pi;
    cout << "请输入 area" << endl;
    cin >> pi.r;

    cout << pi.area << endl; //乱码

    system("pause");
    return 0;
}
```

总结： 从内存四区的角度，解释为什么会出现乱码
理解为什么需要成员函数

2 程序设计方法的发展历程

面向过程的结构化程序设计方法

- 设计思路
 - 自顶向下、逐步求精。采用模块分解与功能抽象，自顶向下、分而治之。
- 程序结构：
 - 按功能划分为若干个基本模块，形成一个树状结构。
 - 各模块间的关系尽可能简单，功能上相对独立；每一模块内部均是由顺序、选择和循环三种基本结构组成。
 - 其模块化实现的具体方法是使用子程序。
- 优点：

有效地将一个较复杂的程序系统设计任务分解成许多易于控制和处理的子任务，便于开发和维护。

- 缺点：可重用性差、数据安全性差、难以开发大型软件和图形界面的应用软件
 - 把数据和处理数据的过程分离为相互独立的实体。
 - 当数据结构改变时，所有相关的处理过程都要进行相应的修改。
 - 每一种相对于老问题的新方法都要带来额外的开销。
 - 图形用户界面的应用程序，很难用过程来描述和实现，开发和维护也都很难。

面向对象的方法

- 将数据及对数据的操作方法封装在一起，作为一个相互依存、不可分离的整体——对象。
- 对同类型对象抽象出其共性，形成类。
- 类通过一个简单的外部接口，与外界发生关系。
- 对象与对象之间通过消息进行通信。

面向对象的基本概念

对象

- 一般意义上的对象：
 - 是现实世界中一个实际存在的事物。
 - 可以是有形的（比如一辆汽车），也可以是无形的（比如一项计划）。
 - 是构成世界的一个独立单位，具有
 - 静态特征：可以用某种数据来描述
 - 动态特征：对象所表现的行为或具有的功能
- 面向对象方法中的对象：
 - 是系统中用来描述客观事物的一个实体，它是用来构成系统的一个基本单位。对象由一组属性和一组行为构成。
 - 属性：用来描述对象静态特征的数据项。
 - 行为：用来描述对象动态特征的操作序列。

类

- 分类——人类通常的思维方法
- 分类所依据的原则——抽象
 - 忽略事物的非本质特征，只注意那些与当前目标有关的本质特征，从而找出

事物的共性，把具有共同性质的事物划分为一类，得出一个抽象的概念。

- 例如，石头、树木、汽车、房屋等都是人们在长期的生产和生活实践中抽象出的概念。

- 面向对象方法中的"类"

- 具有相同属性和服务的一组对象的集合
- 为属于该类的全部对象提供了抽象的描述，包括属性和行为两个主要部分。
- 类与对象的关系：
犹如模具与铸件之间的关系，一个属于某类的对象称为该类的一个实例。

封装

也就是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。

- 把对象的属性和服务结合成一个独立的系统单元。
- 尽可能隐蔽对象的内部细节。对外形成一个边界（或者说一道屏障），只保留有限的对外接口使之与外部发生联系。
- 继承对于软件复用有着重要意义，是面向对象技术能够提高软件开发效率的重要原因之一。
- 定义：特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。
- 例如：将轮船作为一个一般类，客轮便是一个特殊类。

多态

多态是指在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或行为在一般类及其各个特殊类中具有不同的语义。

面向对象的软件工程

- 面向对象的软件工程是面向对象方法在软件工程领域的全面应用。它包括：
 - 面向对象的分析（OOA）
 - 面向对象的设计（OOD）
 - 面向对象的编程（OOP）
 - 面向对象的测试（OOT）
 - 面向对象的软件维护（OOSM）

总结：

面向过程程序设计：数据结构 + 算法

主要解决科学计算问题，用户需求简单而固定

特点：

分析解决问题所需要的步骤

利用函数实现各个步骤

依次调用函数解决问题

问题：

软件可重用性差

软件可维护性差

构建的软件无法满足用户需求

面向对象程序设计：由现实世界建立软件模型

将现实世界中的事物直接映射到程序中，可直接满足用户需求

特点：

- 直接分析用户需求中涉及各个实体
- 在代码中描述现实世界中的实体
- 在代码中关联各个实体协同工作解决问题

优势：

- 构建的软件能够适应用户需求的不断变化
- 直接利用面向过程方法的优势而避开其劣势

3 C 语言和 C++ 语言关系

C 语言是在实践的过程中逐步完善起来的

- 没有深思熟虑的设计过程
- 使用时存在很多“灰色地带”
- 残留量过多低级语言的特征
- 直接利用指针进行内存操作

C 语言的目标是高效

- 最终程序执行效率的高效

当面向过程方法论暴露越来越多的缺陷的时候，业界开始考虑在工程项目中引入面向对象的设计方法，而第一个需要解决的问题就是：高效的面向对象语言，并且能够兼容已经存在的代码。

C 语言 + 面向对象方法论===》Objective C /C++

C 语言和 C++ 并不是对立的竞争关系

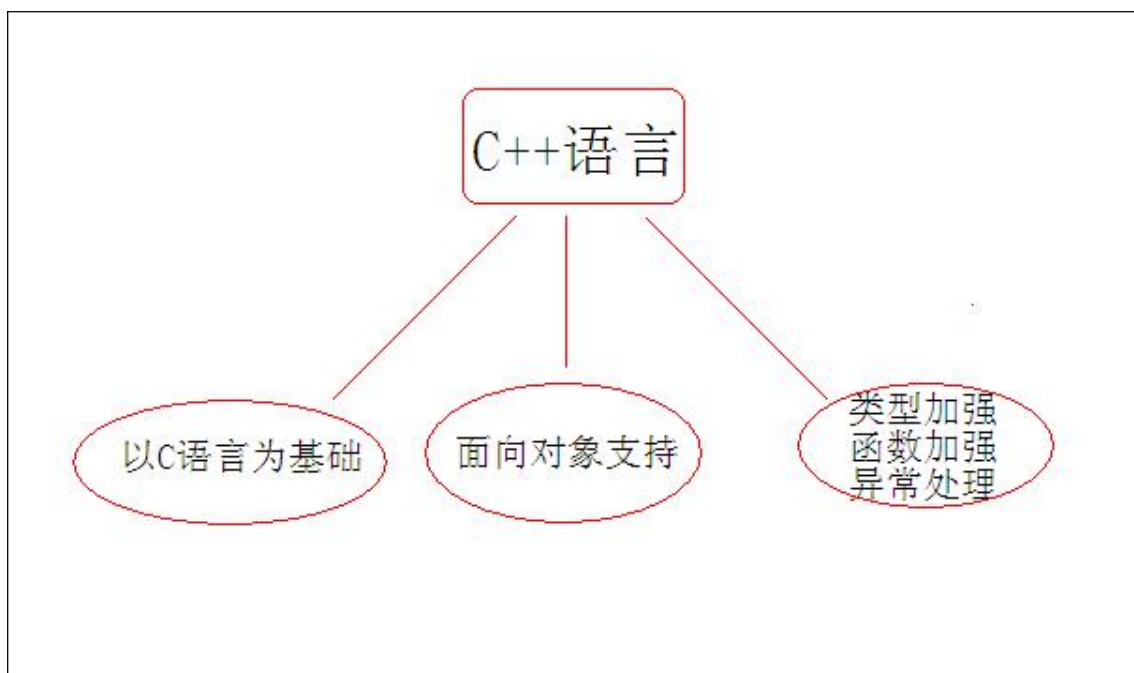
C++ 是 C 语言的加强，是一种更好的 C 语言

C++ 是以 C 语言为基础的，并且完全兼容 C 语言的特性

学习 C++ 并不会影响原有的 C 语言知识，相反会根据加深对 C 的认知；

学习 C++ 可以接触到更多的软件设计方法，并带来更多的机会。

- 1) C++ 是一种更强大的 C，通过学习 C++ 能够掌握更多的软件设计方法
- 2) C++ 是 Java/C#/D 等现代开发语言的基础，学习 C++ 后能够快速掌握这些语言
- 3) C++ 是各大知名软件企业挑选人才的标准之一



4 C++对 C 的加强

4.1 namespace 命名空间

1 C++命名空间基本常识

所谓 namespace，是**标识符的各种可见范围**。C++标准程序库中的所有标识符都被定义于一个名为 std 的 namespace 中。

一：<iostream>和<iostream.h>格式不一样，前者没有后缀，实际上，在你的编译器 include 文件夹里面可以看到，二者是两个文件，打开文件就会发现，里面的代码是不一样的。后缀为.h 的头文件 **c++标准已经明确提出不支持了**，早些的实现将标准库功能定义在全局空间里，声明在带.h 后缀的头文件里，c++标准为了和 C 区别开，也为了正确使用命名空间，规定头文件不使用后缀.h。因此，

1) 当使用<iostream.h>时，相当于在 c 中调用库函数，使用的是全局命名空间，也就是早期的 c++实现；

2) 当使用<iostream>的时候，该头文件没有定义全局命名空间，必须使用 **namespace std;** 这样才能正确使用 **cout**。

二：由于 namespace 的概念，使用 C++标准程序库的任何标识符时，可以有三种选择：

1、直接指定标识符。例如 std::ostream 而不是 ostream。完整语句如下： std::cout << std::hex << 3.4 << std::endl;

2、使用 using 关键字。 using std::cout; using std::endl; using std::cin; 以上程序可以写成 cout << std::hex << 3.4 << endl;

3、最方便的就是使用 using namespace std; 例如： using namespace std;这样命名空间 std 内定义的所有标识符都有效（曝光）。就好像它们被声明为全局变量一样。那么以上语句

可以如下写: `cout << hex << 3.4 << endl;` 因为标准库非常的庞大, 所以程序员在选择类的名称或函数名 时就很有可能和标准库中的某个名字相同。所以为了避免这种情况所造成的名字冲突, 就把标准库中的一切都被放在名字空间 `std` 中。但这又会带来了一个新问 题。无数原有的 C++ 代码都依赖于使用了多年的伪标准库中的功能, 他们都是在全局空间下的。所以就有了 `<iostream.h>` 和 `<iostream>` 等等这样的头文件, 一个是为了兼容以前的 C++ 代码, 一个是为了支持新的标准。命名空间 `std` 封装的是标准程序库的名称, 标准程序库为了和以前的头文件区别, 一般不加 ".h"

2 C++命名空间定义及使用语法

<pre>/* 在 C++ 中, 名称 (name) 可以是符号常量、变量、宏、函数、结构、枚举、类和对象等等。 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++ 引入了关键字 namespace (命名空间/名字空间/名称空间/名域), 可以更好地 控制标识符的作用域。 */</pre>
<pre>/* std 是 c++ 标准命名空间, c++ 标准程序库中的所有标识符都被定义在 std 中, 比如标准库中 的类 iostream、vector 等都定义在该命名空间中, 使用时要加上 using 声明 (using namespace std) 或 using 指示 (如 std::string、 std::vector<int>). */</pre>
<pre>/* C 中的命名空间 在 C 语言中只有一个全局作用域 C 语言中所有的全局标识符共享同一个作用域 标识符之间可能发生冲突 C++ 中提出了命名空间的概念 命名空间将全局作用域分成不同的部分 不同命名空间中的标识符可以同名而不会发生冲突 命名空间可以相互嵌套 全局作用域也叫默认命名空间 */</pre>
<pre>/* C++ 命名空间的定义: namespace name { ... } */</pre>
<pre>/* C++ 命名空间的使用: 使用整个命名空间: using namespace name; 使用命名空间中的变量: using name::variable;</pre>

使用默认命名空间中的变量：::variable

默认情况下可以直接使用默认命名空间中的所有标识符

*/

3 C++命名空间编程实践

```
namespace NameSpaceA
{
    int a = 0;
}

namespace NameSpaceB
{
    int a = 1;

    namespace NameSpaceC
    {
        struct Teacher
        {
            char name[10];
            int age;
        };
    }
}

int main()
{
    using namespace NameSpaceA;
    using NameSpaceB::NameSpaceC::Teacher;

    printf("a = %d\n", a);
    printf("a = %d\n", NameSpaceB::a);

    NameSpaceB::NameSpaceC::Teacher t2
    Teacher t1 = {"aaa", 3};

    printf("t1.name = %s\n", t1.name);
    printf("t1.age = %d\n", t1.age);

    system("pause");
    return 0;
}
```

4 结论

- 1) 当使用<iostream>的时候,该头文件没有定义全局命名空间,必须使用 namespace std; 这样才能正确使用 cout。若不引入 using namespace std ,需要这样做。std::cout。
- 2) c++标准为了和 C 区别开,也为了正确使用命名空间,规定头文件不使用后缀.h。
- 3) C++命名空间的定义: namespace name { ... }
- 4) using namespace NameSpaceA;
- 5) namespace 定义可嵌套。

4.2 “实用性” 增加

```
#include "iostream"
using namespace std;

//C 语言中的变量都必须在作用域开始的位置定义!!
//C++中更强调语言的“实用性”,所有的变量都可以在需要使用时再定义。

int main11()
{
    int i = 0;

    printf("ddd");
    int k;
    system("pause");
    return 0;
}
```

4.3 register 关键字增强

```
//register 关键字 请求编译器让变量 a 直接放在寄存器里面,速度快
//在 c 语言中 register 修饰的变量 不能取地址,但是在 c++里面做了内容

/*
//1
register 关键字的变化
register 关键字请求“编译器”将局部变量存储于寄存器中
C 语言中无法取得 register 变量地址
在 C++中依然支持 register 关键字
C++编译器有自己的优化方式,不使用 register 也可能做优化
C++中可以取得 register 变量的地址
```

```
//2
```

C++编译器发现程序中需要取 `register` 变量的地址时，`register` 对变量的声明变得无效。

```
//3
```

早期 C 语言编译器不会对代码进行优化，因此 `register` 变量是一个很好的补充。

```
*/
```

```
int main22()
```

```
{
```

```
    register int a = 0;
```

```
    printf("&a = %x\n", &a);
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

其他补充：请阅读《`register` 关键字常识课外阅读.docx》

4.4 变量检测增强

```
/*
```

在 C 语言中，重复定义多个同名的全局变量是合法的

在 C++中，不允许定义多个同名的全局变量

C 语言中多个同名的全局变量最终会被链接到全局数据区的同一个地址空间上

```
int g_var;
```

```
int g_var = 1;
```

C++直接拒绝这种二义性的做法。

```
*/
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    printf("g_var = %d\n", g_var);
```

```
    return 0;
```

```
}
```

4.5 struct 类型加强

struct 类型的加强：

C 语言的 struct 定义了一组变量的集合，C 编译器并不认为这是一种新的类型

C++中的 struct 是一个新类型的定义声明

```
struct Student
```

```
{
    char name[100];
    int age;
};

int main(int argc, char *argv[])
{
    Student s1 = {"wang", 1};
    Student s2 = {"wang2", 2};
    return 0;
}
```

4.6 C++中所有的变量和函数都必须有类型

```
/*
C++中所有的变量和函数都必须有类型
    C语言中的默认类型在C++中是不合法的

函数f的返回值是什么类型，参数又是什么类型？
函数g可以接受多少个参数？
*/

//更换成.cpp 试试

f(i)
{
    printf("i = %d\n", i);
}

g()
{
    return 5;
}

int main(int argc, char *argv[])
{

    f(10);

    printf("g() = %d\n", g(1, 2, 3, 4, 5));
}
```

```
    getchar();  
    return 0;  
}
```

总结:

/*

在 C 语言中

int f(); 表示返回值为 int，接受任意参数的函数

int f(void); 表示返回值为 int 的无参函数

在 C++ 中

int f(); 和 int f(void) 具有相同的意义，都表示返回值为 int 的无参函数

*/

C++ 更加强调类型，任意的程序元素都必须显示指明类型

4.2-4.6 属于语法级别的增强。

4.7 新增 Bool 类型关键字

/*

C++ 中的布尔类型

C++ 在 C 语言的基本类型系统之上增加了 bool

C++ 中的 bool 可取的值只有 true 和 false

理论上 bool 只占用一个字节，

如果多个 bool 变量定义在一起，可能会各占一个 bit，这取决于编译器的实现

true 代表真值，编译器内部用 1 来表示

false 代表非真值，编译器内部用 0 来表示

bool 类型只有 true（非 0）和 false（0）两个值

C++ 编译器会在赋值时将非 0 值转换为 true，0 值转换为 false

*/

```
int main(int argc, char *argv[])  
{  
    int a;  
    bool b = true;  
    printf("b = %d, sizeof(b) = %d\n", b, sizeof(b));  
  
    b = 4;  
    a = b;  
    printf("a = %d, b = %d\n", a, b);  
}
```

```
b = -4;
a = b;
printf("a = %d, b = %d\n", a, b);

a = 10;
b = a;
printf("a = %d, b = %d\n", a, b);

b = 0;
printf("b = %d\n", b);

system("pause");
return 0;
}
```

4.8 三目运算符功能增强

1 三目运算符在 C 和 C++ 编译器的表现

```
int main()
{
    int a = 10;
    int b = 20;

    //返回一个最小数 并且给最小数赋值成 3
    //三目运算符是一个表达式，表达式不可能做左值
    (a < b ? a : b) = 30;

    printf("a = %d, b = %d\n", a, b);

    system("pause");

    return 0;
}
```

2 结论

- 1) C 语言返回变量的值 C++ 语言是返回变量本身
C 语言中的三目运算符返回的是变量值，不能作为左值使用
C++ 中的三目运算符可直接返回变量本身，因此可以出现在程序的任何地方
- 2) 注意：三目运算符可能返回的值中如果有一个是常量值，则不能作为左值使用

```
(a < b ? 1 : b) = 30;
```

3) C 语言如何支持类似 C++ 的特性呢？

====> 当左值的条件：要有内存空间；C++ 编译器帮助程序员取了一个地址而已

思考：如何让 C 中的三目运算符当左值呢？

5 C/C++ 中的 const

1 const 基础知识（用法、含义、好处）

```
int main()
{
    const int a;
    int const b;

    const int *c;
    int * const d;
    const int * const e;

    return 0;
}
```

```
int func1(const )
```

初级理解：const 是定义常量==》const 意味着只读

含义：

//第一个第二个意思一样 代表一个常整形数

//第三个 c 是一个指向常整形数的指针(所指向的内存数据不能被修改，但是本身可以修改)

//第四个 d 常指针（指针变量不能被修改，但是它所指向内存空间可以被修改）

//第五个 e 一个指向常整形的常指针（指针和它所指向的内存空间，均不能被修改）

Const 好处

//合理的利用 const，

//1 指针做函数参数，可以有效的提高代码可读性，减少 bug；

//2 清楚的分清参数的输入和输出特性

```
int setTeacher_err( const Teacher *p)
```

Const 修改形参的时候，在利用形参不能修改指针所指向的内存空间

2 C 中“冒牌货”

```
int main()
{
    const int a = 10;
```

```

int *p = (int*)&a;
printf("a==>%d\n", a);
*p = 11;
printf("a==>%d\n", a);

printf("Hello.....\n");
return 0;
}

```

解释:

C++编译器对 `const` 常量的处理

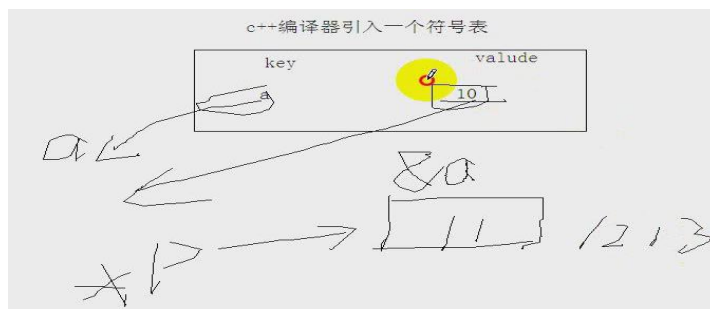
当碰见**常量声明**时，在符号表中放入常量 ==> 问题：那有如何解释取地址

编译过程中若发现使用常量则直接以符号表中的值替换

编译过程中若发现对 `const` 使用了 `extern` 或者 `&` 操作符，则给对应的常量分配存储空间（兼容 C）

? 联想： `int &a = 1(err)` & `const int &a = 10(ok)?`

C++中 `const` 符号表原理图



注意:

C++编译器虽然可能为 `const` 常量分配空间，但不会使用其存储空间中的值。

结论:

C 语言中的 `const` 变量

C 语言中 `const` 变量是只读变量，有自己的存储空间

C++中的 `const` 常量

可能分配存储空间,也可能不分配存储空间

当 `const` 常量为全局，并且需要在其它文件中使用

当使用 `&` 操作符取 `const` 常量的地址

3 `const` 和 `#define` 相同之处

```

//练习 解释为什么
//#define N 10
int main()
{
    const int a = 1;
    const int b = 2;
}

```



```
int array[a + b] = {0};
int i = 0;

for(i=0; i<(a+b); i++)
{
    printf("array[%d] = %d\n", i, array[i]);
}

getchar();

return 0;
}
```

C++中的 `const` 修饰的，是一个真正的常量，而不是 C 中变量（只读）。在 `const` 修饰的常量编译期间，就已经确定下来了。

4 `const` 和 `#define` 的区别

对比加深

C++中的 `const` 常量类似于宏定义

`const int c = 5;` \approx `#define c 5`

C++中的 `const` 常量与宏定义不同

`const` 常量是由编译器处理的，提供类型检查和作用域检查

宏定义由预处理器处理，单纯的文本替换

//在 `func1` 定义 `a`，在 `func2` 中能使用吗？

//在 `func1` 中定义的 `b`，在 `func2` 中能使用吗？

练习

```
void fun1()
{
    #define a 10
    const int b = 20;
    // #undef a    # undef
}

void fun2()
{
    printf("a = %d\n", a);
    // printf("b = %d\n", b);
}

int main()
{
    fun1();
}
```

```
fun2();  
return 0;  
}
```

5 结论

C 语言中的 const 变量

C 语言中 const 变量是只读变量，有自己的存储空间

C++ 中的 const 常量

可能分配存储空间,也可能不分配存储空间

当 const 常量为全局，并且需要在其它文件中使用，会分配存储空间

当使用&操作符，取 const 常量的地址时，会分配存储空间

当 const int &a = 10; const 修饰引用时，也会分配存储空间

6 引用专题讲座

1 引用（普通引用）

变量名回顾

变量名实质上是一段连续存储空间的别名，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

问题 1：对一段连续的内存空间只能取一个别名吗？

1 引用概念

- a) 在C++中新增加了引用的概念
- b) 引用可以看作一个已定义变量的别名
- c) 引用的语法：Type& name = var;
- d) 引用做函数参数那？（引用作为函数参数声明时不进行初始化）

```
void main01()  
{  
    int a = 10; //c编译器分配4个字节内存。。。a内存空间的别名  
    int &b = a;  //b就是a的别名。。。  
    a = 11; //直接赋值  
    {  
        int *p = &a;  
        *p = 12;
```

```
        printf("a %d \n",a);
    }
    b = 14;
    printf("a:%d b:%d", a, b);
    system("pause");
}
```

2 引用是 C++ 的概念

属于C++编译器对C的扩展

问题：C中可以编译通过吗？

```
int main()
{
    int a = 0;
    int &b = a; //int * const b = &a
    b = 11;    //*b = 11;

    return 0;
}
```

结论：请不要用C的语法考虑 b=11

3 引用做函数参数

普通引用在声明时必须用其它的变量进行初始化，
引用作为函数参数声明时不进行初始化

//05复杂数据类型 的引用

```
struct Teacher
```

```
{
    char name[64];
    int age ;
};
```

```
void printfT(Teacher *pT)
```

```
{
    cout<<pT->age<<endl;
}
```

//pT是t1的别名 ,相当于修改了t1

```
void printfT2(Teacher &pT)
```

```
{
    //cout<<pT.age<<endl;
    pT.age = 33;
```

```
}

//pT和t1的是两个不同的变量
void printfT3(Teacher pT)
{
    cout<<pT.age<<endl;
    pT.age = 45; //只会修改pT变量 ,不会修改t1变量
}

void main()
{
    Teacher t1;
    t1.age = 35;

    printfT(&t1);

    printfT2(t1); //pT是t1的别名
    printf("t1.age:%d \n", t1.age); //33

    printfT3(t1) ;// pT是形参 ,t1 copy一份数据 给pT      //---> pT = t1
    printf("t1.age:%d \n", t1.age); //35

    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

4 引用的意义

- 1) 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
- 2) 引用相对于指针来说具有更好的可读性和实用性

```
int swap1(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
    return 0;
}
```

```
int swap2(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
    return 0;
}
```

5 引用本质思考

思考1: C++编译器背后做了什么工作?

```
int main()
{
    int a = 10;
    int &b = a;
    //b是a的别名, 请问c++编译器后面做了什么工作?
    b = 11;
    cout<<"b---"><<a<<endl;
    printf("a:%d\n", a);
    printf("b:%d\n", b);
    printf("&a:%d\n", &a);
    printf("&b:%d\n", &b); //请思考: 对同一内存空间可以取好几个名字吗?
    system("pause");
    return 0;
}
```

单独定义的引用时, 必须初始化; 说明很像一个常量

思考2: 普通引用有自己的空间吗?

```
struct Teacher {
    int &a;
    int &b;
};

int main()
{
    printf("sizeof(Teacher) %d\n", sizeof(Teacer));
    system("pause");
    return 0;
}
```

引用是一个有地址, 引用是常量。。。。。

char *const p

6 引用的本质

1) 引用在C++中的内部实现是一个常指针

Type& name \longleftrightarrow Type* const name

2) 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。

3) 从使用的角度, 引用会让人误会其只是一个别名, 没有自己的存储空间。这是C++为了实用性而做出的细节隐藏

```
void func(int &a)
{
    a = 5;
}
```

```
void func(int *const a)
{
    *a = 5;
}
```

```
int main()
{
    int x = 10;
    func(x);
}
```

4) 请仔细对比间接赋值成立的三个条件

- 1定义两个变量 （一个实参一个形参）
- 2建立关联 实参取地址传给形参
- 3*p形参去间接的修改实参的值

7 引用结论

1) 引用在实现上，只不过是把：间接赋值成立的三个条件的后两步和二为一

//当实参传给形参引用的时候，只不过是c++编译器帮我们程序员手工取了一个实参地址，传给了形参引用（常量指针）

2) 当我们使用引用语法的时，我们不去关心编译器引用是怎么做的

当我们分析奇怪的语法现象的时，我们才去考虑c++编译器是怎么做的

8 函数返回值是引用(引用当左值)

C++引用使用时的难点：

当函数返回值为引用时

若返回栈变量

不能成为其它引用的初始值

不能作为左值使用

若返回静态变量或全局变量

可以成为其他引用的初始值

即可作为右值使用，也可作为左值使用

C++链式编程中，经常用到引用，运算符重载专题

返回值是基础类型，当引用

```
int getAA1()
{
    int a;
    a = 10;
    return a;
}
```

//基础类型a返回的时候，也会有一个副本

```
int& getAA2()
```

```
{  
    int a;  
    a = 10;  
    return a;  
}
```

```
int* getAA3()
```

```
{  
    int a;  
    a = 10;  
    return &a;  
}
```

返回值是static变量，当引用

//static修饰变量的时候，变量是一个状态变量

```
int j()
```

```
{  
    static int a = 10;  
    a ++;  
    printf("a:%d \n", a);  
    return a;  
}
```

```
int& j1()
```

```
{  
    static int a = 10;  
    a ++;  
    printf("a:%d \n", a);  
    return a;  
}
```

```
int *j2()
```

```
{  
    static int a = 10;  
    a ++;  
    printf("a:%d \n", a);  
    return &a;  
}
```

```
void main22()
```

```
{
    // j()的运算结果是一个数值，没有内存地址，不能当左值。。。。
    //11 = 100;
    //*(a>b?&a:&b) = 111;
    //当被调用的函数当左值的时候，必须返回一个引用。。。。
    j1() = 100; //编译器帮我们打造了环境
    j1();
    *(j2()) = 200; //相当于我们程序员手工的打造 做左值的条件
    j2();
    system("pause");
}
```

返回值是形参，当引用

```
int g1(int *p)
{
    *p = 100;
    return *p;
}

int& g2(int *p) //
{
    *p = 100;
    return *p;
}
```

//当我们使用引用语法的时候，我们不去关心编译器引用是怎么做的
//当我们分析乱码这种现象的时候，我们才去考虑c++编译器是怎么做的。。。

```
void main23()
{
    int a1 = 10;
    a1 = g2(&a1);

    int &a2 = g2(&a1); //用引用去接受函数的返回值，是不是乱码，关键是看返回的内存空间是不是被编译器回收了。。。
    printf("a1:%d \n", a1);
    printf("a2:%d \n", a2);

    system("pause");
}
```


返回值非基础类型

```
struct Teacher
```

```
{
```

```
    char name[64];
```

```
    int age;
```

```
};
```

//如果返回引用不是基础类型，是一个类，那么情况非常赋值。。涉及到copy构造函数和=操作重载，抛砖。。。

```
struct Teacher
```

```
{
```

```
    char name[64];
```

```
    int age;
```

```
};
```

//如果返回引用不是基础类型，是一个类，那么情况非常赋值。。涉及到copy构造函数和=操作重载，抛砖。。。

```
struct Teacher & OpTeacher(struct Teacher &t1)
```

```
{
```

```
}
```

9 指针引用

```
#include "iostream"
```

```
using namespace std;
```

```
struct Teacher
```

```
{
```

```
    char name[64];
```

```
    int age;
```

```
};
```

```
int getTe(Teacher **myp )
```

```
{
```

```
    Teacher *p = (Teacher *)malloc(sizeof(Teacher));
```

```
    if (p ==NULL)
```

```
    {
```

```
        return -1;
```

```
    }
```

```
    memset(p, 0, sizeof(Teacher));
```

```
    p->age = 33;
```

```
*myp = p; //
return 0;
}

//指针的引用而已
int getTe2(Teacher* &myp)
{
    myp = (Teacher *)malloc(sizeof(Teacher));
    myp->age = 34;

    return 0;
}

void main333()
{
    Teacher *p = NULL;
    //getTe(&p);
    getTe2(p);

    printf("age:%d \n", p->age);
    system("pause");
}
```

2 常引用

下面开始进入const引用难点

1 使用变量初始化 const 引用

思考const int &a = b PK const int &a = 10;

???? 问题: const引用,

在 C++中可以声明 const 引用

const Type& name = var;

const 引用让变量拥有只读属性

案例1:

```
int main()
{
    int a = 10;
    const int &b = a;

    //int *p = (int *)&b;
    b = 11; //err
    //*p = 11; //只能用指针来改变了
```

```
cout<<"b---"<<a<<endl;
printf("a:%d\n", a);
printf("b:%d\n", b);
printf("&a:%d\n", &a);
printf("&b:%d\n", &b);
system("pause");
return 0;
}
```

案例2:

```
void main41()
{
    int a = 10;

    const int &b = a; //const引用 使用变量a初始化
    a = 11;
    //b = 12; //通过引用修改a, 对不起修改不了
    system("pause");
}
```

```
struct Teacher1
{
    char name[64];
    int age;
};
```

```
void printTe2(const Teacher1 *const pt)
{
}
```

//const引用让变量(所指内存空间)拥有只读属性

```
void printTe(const Teacher1 &t)
{
    //t.age = 11;
}
```

```
void main42()
{
    Teacher1 t1;
    t1.age = 33;
    printTe(t1);
    system("pause");
}
```

2 使用字面量常量初始化 const 引用

思考：

- 1、用变量对const引用初始化，const引用分配内存空间了吗？
- 2、用常量对const引用初始化，const引用分配内存空间了吗？

```
void main()
{
    const int b = 10;
    printf("b:%d", &b);

    //int &a1 = 19; 如果不加const编译失败
    const int &a = 19;
    printf("&a:%d \n", &a);

    system("pause");
}
```

3 综合案例

```
void main()
{
    //普通引用
    int a = 10;
    int &b = a;

    //常量引用：让变量引用只读属性
    const int &c = a;

    //常量引用初始化 分为两种
    //1 用变量 初始化 常量引用
    {
        int x = 20;
        const int& y = x;
        printf("y:%d \n", y);
    }

    //2 用常量 初始化 常量引用
    {
        //int &m = 10; //引用是内存空间的别名 字面量 10 没有内存空间 没有方法做引用
        const int &m = 10;
    }

    cout<<"hello..."<<endl;
}
```

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

3 const 引用结论

- 1) Const & int e 相当于 const int * const e
- 2) 普通引用 相当于 int *const e1
- 3) 当使用常量（字面量）对const引用进行初始化时，C++编译器会为常量值分配空间，并将引用名作为这段空间的别名
- 4) 使用字面量对const引用初始化后，将生成一个只读变量

4const 修饰类

后续课程介绍

5 综合练习

```
int& j()  
{  
    static int a = 0;  
    return a;  
}  
  
int& g()  
{  
    int a = 0;  
    return a;  
}  
  
int main()  
{  
    int a = g();  
    int& b = g();  
    j() = 10;  
    printf("a = %d\n", a);  
    printf("b = %d\n", b);  
    printf("f() = %d\n", f());  
    system("pause");  
    return 0;  
}
```

7C++对 C 的函数扩展

1 inline 内联函数

<p>C++中的 const 常量可以替代宏常数定义，如：</p> <pre>const int A = 3; #define A 3</pre> <p>C++中是否有解决方案替代宏代码片段呢？（替代宏代码片段就可以避免宏的副作用！）</p>
<p>C++中推荐使用内联函数替代宏代码片段</p> <p>C++中使用 inline 关键字声明内联函数</p>
<p>内联函数声明时 inline 关键字必须和函数定义结合在一起，否则编译器会直接忽略内联请求。</p> <p>//宏替换和函数调用区别</p>
<pre>#include "iostream" using namespace std; #define MYFUNC(a, b) ((a) < (b) ? (a) : (b)) inline int myfunc(int a, int b) { return a < b ? a : b; } int main() { int a = 1; int b = 3; //int c = myfunc(++a, b); //头疼系统 int c = MYFUNC(++a, b); printf("a = %d\n", a); printf("b = %d\n", b); printf("c = %d\n", c); system("pause"); return 0; }</pre>
<p>说明 1:</p> <p>必须 inline int myfunc(int a, int b)和函数体的实现，写在一块</p>
<p>说明 2</p>
<p>C++编译器可以将一个函数进行内联编译</p> <p>被 C++编译器内联编译的函数叫做内联函数</p> <p>内联函数在最终生成的代码中是没有定义的</p> <p>C++编译器直接将函数体插入在函数调用的地方</p> <p>内联函数没有普通函数调用时的额外开销(压栈，跳转，返回)</p>

说明 3: C++编译器不一定准许函数的内联请求!

说明 4

内联函数是一种特殊的函数, 具有普通函数的特征 (参数检查, 返回类型等)

内联函数是对编译器的一种请求, 因此编译器可能拒绝这种请求

内联函数由 **编译器处理**, 直接将编译后的函数体插入调用的地方

宏代码片段 **由预处理器处理**, 进行简单的文本替换, 没有任何编译过程

说明 5:

现代 C++编译器能够进行编译优化, 因此一些函数即使没有 `inline` 声明, 也可能被编译器内联编译

另外, 一些现代 C++编译器提供了扩展语法, 能够对函数进行强制内联

如: g++中的 `__attribute__((always_inline))` 属性

说明 6:

C++中内联编译的限制:

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

函数内联声明必须在调用语句之前

编译器对于内联函数的限制并不是绝对的, 内联函数相对于普通函数的优势只是省去了函数调用时压栈, 跳转和返回的开销。

因此, 当函数体的执行开销远大于压栈, 跳转和返回所用的开销时, 那么内联将无意义。

结论:

1) **内联函数在编译时直接将函数体插入函数调用的地方**

2) `inline` 只是一种请求, 编译器不一定允许这种请求

3) 内联函数省去了普通函数调用时压栈, 跳转和返回的开销

2 默认参数

/*1

C++中可以在函数声明时为参数提供一个默认值,

当函数调用时没有指定这个参数的值, 编译器会自动用默认值代替

*/

```
void myPrint(int x = 3)
```

```
{
```

```
    printf("x:%d", x);
```

```
}
```

/*2

函数默认参数的规则

只有参数列表后面部分的参数才可以提供默认参数值

一旦在一个函数调用中开始使用默认参数值, 那么这个参数后的所有参数都必须使用默认参数值

*/

```
//默认参数
void printAB(int x = 3)
{
    printf("x:%d\n", x);
}

//在默认参数规则，如果默认参数出现，那么右边的都必须有默认参数
void printABC(int a, int b, int x = 3, int y=4, int z = 5)
{
    printf("x:%d\n", x);
}

int main62(int argc, char *argv[])
{
    printAB(2);
    printAB();
    system("pause");
    return 0;
}
```

3 函数占位参数

```
/*
函数占位参数
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
*/
```

```
int func(int a, int b, int )
{
    return a + b;
}

int main01()
{
    //func(1, 2); //可以吗?
    printf("func(1, 2, 3) = %d\n", func(1, 2, 3));

    getchar();
    return 0;
}
```


4 默认参数和占位参数

```
/*
可以将占位参数与默认参数结合起来使用
    意义
    为以后程序的扩展留下线索
    兼容 C 语言程序中可能出现的不规范写法
*/
//C++可以声明占位符参数，占位符参数一般用于程序扩展和对 C 代码的兼容

int func2(int a, int b, int = 0)
{
    return a + b;
}

void main()
{
    //如果默认参数和占位参数在一起，都能调用起来
    func2(1, 2);
    func2(1, 2, 3);
    system("pause");
}
```

结论：//如果默认参数和占位参数在一起，都能调用起来

5 函数重载（Overload）

函数重载概念

```
1 函数重载概念
函数重载(Function Overload)
    用同一个函数名定义不同的函数
    当函数名和不同的参数搭配时函数的含义不同

2 函数重载的判断标准
/*
函数重载至少满足下面的一个条件：
    参数个数不同
    参数类型不同
    参数顺序不同
*/

3 函数返回值不是函数重载的判断标准
实验 1：调用情况分析；实验 2：判断标准

//两个难点：重载函数和默认函数参数混搭 重载函数和函数指针
/*
```

```
int func(int x)
{
    return x;
}

int func(int a, int b)
{
    return a + b;
}

int func(const char* s)
{
    return strlen(s);
}

int main()
{
    int c = 0;

    c = func(1);

    printf("c = %d\n", c);

    c = func(1, 2);

    printf("c = %d\n", c);

    c = func("12345");

    printf("c = %d\n", c);

    printf("Press enter to continue ...");
    getchar();
    return 0;
}
*/
```

函数重载的调用准则

```
/*
编译器调用重载函数的准则
    将所有同名函数作为候选者
    尝试寻找可行的候选函数
*/
```

精确匹配实参

通过默认参数能够匹配实参

通过默认类型转换匹配实参

匹配失败

最终寻找到的可行候选函数不唯一，则出现二义性，编译失败。

无法匹配所有候选者，函数未定义，编译失败。

*/

/*

函数重载的注意事项

重载函数在本质上是相互独立的不同函数（静态链编）

重载函数的函数类型是不同的

函数返回值不能作为函数重载的依据

函数重载是由函数名和参数列表决定的。

*/

函数重载是发生在一个类中里面

函数重载遇上函数默认参数

//当函数默认参数遇上函数重载会发生什么

/*

int func(int a, int b, int c = 0)

{

return a * b * c;

}

int func(int a, int b)

{

return a + b;

}

//1 个参数的允许吗

int func(int a)

{

return a + b;

}

int main()

{

int c = 0;

c = func(1, 2); // 存在二义性，调用失败，编译不能通过

```
printf("c = %d\n", c);

printf("Press enter to continue ...");
getchar();
return 0;
}
*/
```

函数重载和函数指针结合

```
/*
函数重载与函数指针
    当使用重载函数名对函数指针进行赋值时
    根据重载规则挑选与函数指针参数列表一致的候选者
    严格匹配候选者的函数类型与函数指针的函数类型
*/
/*
int func(int x) // int(int a)
{
    return x;
}

int func(int a, int b)
{
    return a + b;
}

int func(const char* s)
{
    return strlen(s);
}

typedef int(*PFUNC)(int a); // int(int a)

int main()
{
    int c = 0;
    PFUNC p = func;

    c = p(1);
}
```

```
printf("c = %d\n", c);

printf("Press enter to continue ...");
getchar();
return 0;
}
*/
```

函数重载、重写、重定义

后续课程。

8 附录

附录 1：C++ 语言对 C 语言扩充和增强的几点具体体现

在 C 语言块注释的形式

```
/*
Explanation Sentence
*/
```

的基础上，C++ 语言提供了一种新的单行注释形式：

```
//Explanation Sentence
```

即用“//”表示注释开始，从该位置直到当前行结束的所有字符都被作为注释。

举个简单的例子：

```
/*
下面的程序段计算从1到100的整数和，
结果记录在变量sum中
*/
sum = 0; //变量sum初值置为零
for ( i = 1; i <= 100; i++ )
{ //循环体开始
    sum += i; // 依次累加i
}
```

2. 更加灵活的变量说明

在传统的C语言中，局部变量的说明必须集中放在执行代码的前面，数据说明语句和执行语句的混合将引起编译错误。而在C++中，可以在程序代码块的任何地方进行局部变量的说明。比如下面的代码在C语言中是不正确的，在C++语言中却可以正常运行。

```
for(int i = 1; i <= 100; i++);
```

这样做的好处是使变量的定义和它的使用集中在一起，意义一目了然。

3. 更加严格的函数原型说明

C++摒弃了C语言对函数原型随意简化的方式，这种简化是许多C语言程序错误的根源。C++语言要求编程者为函数提供完整的原型，包括全部参数的类型和返回值说明。

例如，有字符型和双精度类型两个参数、返回整型值的函数f，原型应该写为：

```
int f(char, double);
```

而C语言中允许将这个原型写成“f()；”。

在函数原型说明中，参数名可有可无，并且可以和函数定义中的参数名不一致。

4. 增加了函数重载机制

重载是程序语言领域的重要概念。常规语言中最典型的例子是“+、-、×、/”等各种算术运算符的重载，这些符号可以同时用来表示多种类型数据之间的运算，这种对一个名字或一个符号赋予多重意义的情况就叫重载。

C++语言增加了C语言所没有的函数重载机制。对一个函数名可以给出多个函数定义，只要这些定义可以通过参数个数或类型的不同区别开来即可。

C++还允许对系统中预先定义的运算符进行重载，增加新的定义。这样做的优点是在今后对新定义类型的变量进行运算时，计算公式写起来方便自然。

5. 函数缺省参数

C++中允许函数有缺省参数。所谓缺省，是指函数调用时可以不给出实际的参数值。下面是一个有缺省参数的函数定义的实例：

```
int f(int a, int b=1)
{
    return a*b;
}
```

此后，函数调用f(3,1)和f(3)将返回同样的结果。

8. 输入/输出流机制

C++保留了C语言标准库中各种输入/输出函数，而且提供了一套新的输入/输出机制——流机制。

比如向标准输出输出一个字符串：

```
cout<<"C++ is beautiful!";
```

或者由标准输入读一个整数，赋给变量a

```
int a;
cin>>a;
```

流式输入/输出运算符能够根据变量类型自动确定数据交换过程中的转换方式，还可以定义“<<、>>”的重载，方便了编程者自定义类型的数据的输入/输出。

9. 作用域限定运算符 ::

作用域限定运算符 :: 用于对当前作用域之外的同名变量进行访问。例如在下面的例子中，我们可以利用 :: 实现在局部变量 a 的作用域范围内对全局变量 a 的访问。

```
#include <iostream.h>

int a;

void main()
{
    float a;
    a = 3.14;
    ::a = 6;
    cout<<"local variable a = "<<a<<endl;
    cout<<"global variable a = "<<::a<<endl;
}
```

程序执行结果如下：

```
local variable a = 3.14
global variable a = 6
```

附录 2：C 语言 register 关键字——最快的关键字

register: 这个关键字请求编译器尽可能的将变量存在 CPU 内部寄存器中，而不是通过内存寻址访问，以提高效率。**注意是尽可能，不是绝对**。你想想，一个 CPU 的寄存器也就那么几个或几十个，你要是定义了很多很多 register 变量，它累死也可能不能全部把这些变量放入寄存器吧，轮也可能轮不到你。

一、皇帝身边的小太监——寄存器

不知道什么是寄存器？那见过太监没有？没有？其实我也没有。没见过不要紧，见过就麻烦大了。^_^，大家都看过古装戏，那些皇帝们要阅读奏章的时候，大臣总是先将奏章交给皇帝旁边的小太监，小太监呢再交给皇帝同志处理。这个小太监只是个**中转站**，并无别的功能。

好，那我们再联想到我们的 CPU。CPU 不就是我们的皇帝同志么？大臣就相当于我们的内存，数据从他这拿出来。那小太监就是我们的寄存器了（这里先不考虑 CPU 的高速缓存区）。数据从内存里拿出来先放到寄存器，然后 CPU 再从寄存器里读取数据来处理，处理完后同样把数据通过寄存器存放到内存里，**CPU 不直接和内存打交道**。这里要说明的一点是：小太监是主动的从大臣手里接过奏章，然后主动的交给皇帝同志，但寄存器没这么自觉，它从不主动干什么事。一个皇帝可能有好些小太监，那么一个 CPU 也可以有很多寄存器，不同型号的 CPU 拥有寄存器的数量不一样。

为啥要这么麻烦啊？**速度！就是因为速度**。寄存器其实就是一块一块小的存储空间，只不过其存取速度要比内存快得多。进水楼台先得月嘛，它离 CPU 很近，CPU 一伸手就拿到数据了，比在那么大的内存里去寻找某个地址上的数据是不是快多了？那有人问既然

它速度那么快，那我们的内存硬盘都改成寄存器得了呗。我要说的是：你真有钱！

二、举例

`register` 修饰符暗示编译程序相应的变量将被频繁地使用，如果可能的话，应将其保存在 CPU 的寄存器中，以加快其存储速度。例如下面的内存块拷贝代码，

```
#ifdef NOSTRUCTASSIGN

memcpy (d, s, l)

{

    register char *d;

    register char *s;

    register int i;

    while (i--)

        *d++ = *s++;

}

#endif
```

三、使用 `register` 修饰符的注意点

但是使用 `register` 修饰符有几点限制。

首先，`register` 变量必须是能被 CPU 所接受的类型。这通常意味着 `register` 变量必须是一个单个的值，并且长度应该小于或者等于整型的长度。不过，有些机器的寄存器也能存放浮点数。

其次，因为 `register` 变量可能不存放在内存中，所以不能用“&”来获取 `register` 变量的地址。

由于寄存器的数量有限，而且某些寄存器只能接受特定类型的数据（如指针和浮点数），因此真正起作用的 `register` 修饰符的数目和类型都依赖于运行程序的机器，而任何多余的 `register` 修饰符都将被编译程序所忽略。

在某些情况下，把变量保存在寄存器中反而会降低程序的运行速度。因为被占用的寄存器不能再用于其它目的；或者变量被使用的次数不够多，不足以装入和存储变量所带来的额外开销。

早期的 C 编译程序不会把变量保存在寄存器中，除非你命令它这样做，这时 `register` 修饰符是 C 语言的一种很有价值的补充。然而，随着编译程序设计技术的进步，在决定那些变量应该被存到寄存器中时，现在的 C 编译环境能比程序员做出更好的决定。实际上，许多编译程序都会忽略 `register` 修饰符，因为尽管它完全合法，但它仅仅是暗示而不是命令。

9 作业及强化训练

- 1 复杂数据类型引用做函数参数
分析内存四区变化图
- 2 代码敲一遍
- 3 设计一个类, 求圆形的周长
- 4 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

2、类和对象

1 前言

C++学习技术路线及目标

研究 C++编译器管理类和方法的避免死角

C++编译器对类对象的生命周期管理，对象创建、使用、销毁

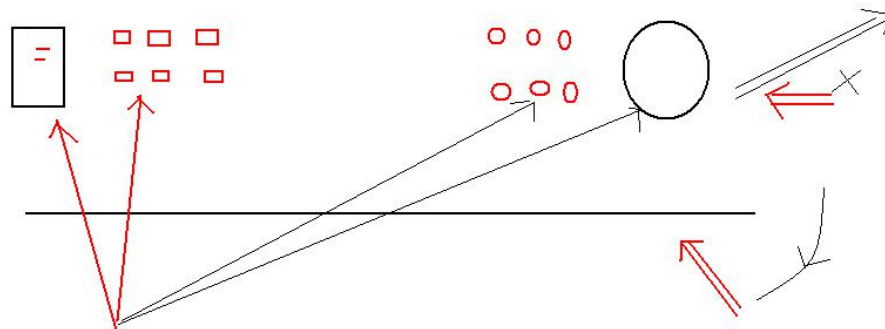
C++面向对象模型初探

C++面向对象多态原理探究

操作符重载

C++基础课程学习完毕以后，有没有一个标准，来判断自己有没有入门。

面向抽象类（接口）编程



1 视角问题

2 具体路线

1个对象的生命周期

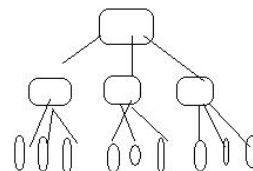
2个对象的生命周期 类和类之间的关系

class A 含有一个 class B

n个对象的生命周期 一族对象

C++面向编程思想

工具：面向抽象类编程
(面向接口/软件分层)



2 类和对象

2.1 基本概念

1) 类、对象、成员变量、成员函数

2) 面向对象三大概念

封装、继承、多态

3) 编程实践

类的定义和对象的定义，对象的使用

求圆形的面积

定义 Teacher 类，打印 Teacher 的信息（把类的声明和类的实现分开）

2.2 类的封装

1) 封装（Encapsulation）

A) 封装，是面向对象程序设计最基本的特性。把数据（属性）和函数（操作）合成一个整体，这在计算机世界中是用类与对象实现的。

B) 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。

备注：有 2 层含义（把属性和方法进行封装 对属性和方法进行访问控制）

C++ 中类的封装

成员变量，C++ 中用于表示类属性的变量

成员函数，C++ 中用于表示类行为的函数

2) 类成员的访问控制

在 C++ 中可以给成员变量和成员函数定义访问级别

Public 修饰成员变量和成员函数可以在类的内部和类的外部被访问

Private 修饰成员变量和成员函数只能在类的内部被访问

//类是把属性和方法封装 同时对信息进行访问控制

//类的内部，类的外部

//我们抽象了一个类，用类去定义对象

//类是一个数据类型，类是抽象的

//对象是一个具体的变量。。占用内存空间。

class Circle

{

public:

double r;

double s;

public:

double getR()

{

a++;

return r;

}

void setR(double val)

{

r = val;

}

public:

double getS() //增加功能时，是在修改类，修改类中的属性或者是方法

{

s = 3.14f*r*r;

return s;

}

//private:

```
int a;  
};
```

3) struct 和 class 关键字区别

在用 struct 定义类时，所有成员默认属性为 public

在用 class 定义类时，所有成员默认属性为 private

2.3 C++ 面向对象程序设计举例

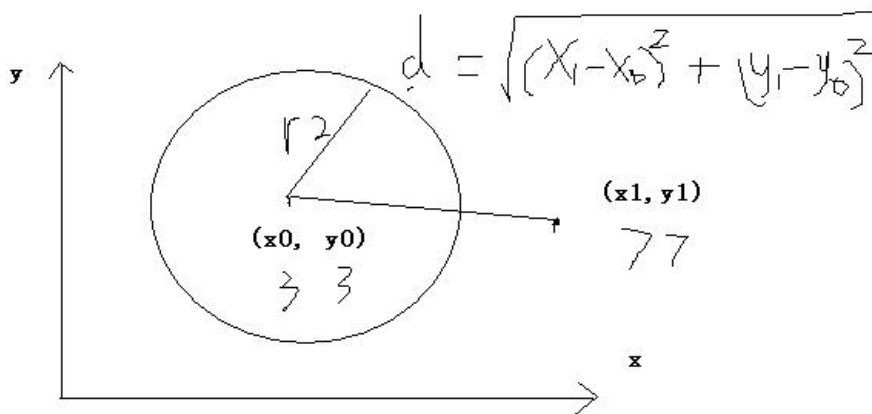
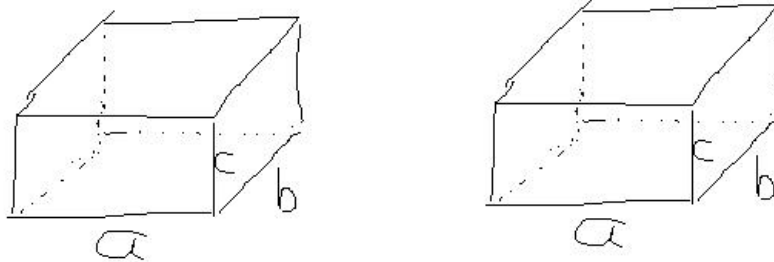
目标：面向过程向面向对象思想转变

初学者要仔细体会类和对象之间的关系，并通过适当练习巩固和提高！

案例 1 设计立方体类(cube)，求出立方体的面积和体积
求两个立方体，是否相等（全局函数和成员函数）

案例 2 设计一个圆形类 (AdvCircle)，和一个点类 (Point)，计算点在圆内部还是圆外
即：求点和圆的关系（圆内和圆外）

案例 3 对于第二个案例，类的声明和类的实现分开



2.4 作业

作业 1: 编写 C++ 程序完成以下功能：

- 1) 定义一个 `Point` 类，其属性包括点的坐标，提供计算两点之间距离的方法；
- 2) 定义一个圆形类，其属性包括圆心和半径；
- 3) 创建两个圆形对象，提示用户输入圆心坐标和半径，判断两个圆是否相交，并输出结果。

作业 2: 设计并测试一个名为 `Rectangle` 的矩形类，其属性为矩形的左下角与右上角两个点的坐标，根据坐标能计算出矩形的面积

作业 3: 定义一个 `Tree` 类，有成员 `ages` (树龄)，成员函数 `grow(int years)` 对 `ages` 加上 `years`，`age()` 显示 `tree` 对象的 `ages` 的值。

3 对象的构造和析构

前言

创建一个对象时，常常需要作某些初始化的工作，例如对数据成员赋初值。**注意，类的数据成员是不能在声明类时初始化的。**

为了解决这个问题，C++编译器提供了**构造函数(constructor)**来处理对象的初始化。**构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用它，而是在建立对象时自动执行。**

3.1 构造和析构函数

1 构造函数和析构函数的概念

有关构造函数

1 构造函数定义及调用

1) C++中的类可以定义与类名相同的特殊成员函数，这种与类名相同的成员函数叫做构造函数；

2) 构造函数在定义时可以有参数；

3) 没有任何返回类型的声明。

2 构造函数的调用

自动调用：一般情况下 C++编译器会自动调用构造函数

手动调用：在一些情况下则需要手工调用构造函数

有关析构函数

3) 析构函数定义及调用

1) C++中的类可以定义一个特殊的成员函数清理对象，这个特殊的成员函数叫做析构函数

语法：~ClassName()

2) 析构函数没有参数也没有任何返回类型的声明

3) 析构函数在对象销毁时自动被调用

4) 析构函数调用机制

C++编译器自动调用

代码演示: [dm01_构造函数的基础.cpp](#)

2 C++编译器构造析构方案 PK 对象显示初始化方案

设计构造函数和析构函数的原因

面向对象的思想是从生活中来,手机、车出厂时,是一样的。

生活中存在的对象都是被初始化后才上市的; **初始状态**是对象普遍存在的一个状态的普通方案:

为每个类都提供一个 public 的 initialize 函数;

对象创建后立即调用 initialize 函数进行初始化。

优缺点分析

1) initialize 只是一个普通的函数,必须显示的调用

2) 一旦由于失误的原因,对象没有初始化,那么结果将是不确定的

没有初始化的对象,其内部成员变量的值是不定的

3) 不能完全解决问题

//为什么对象需要初始化 有什么样的初始化方案

```
#include "iostream"
```

```
using namespace std;
```

```
/*
```

思考为什么需要初始化

面向对象思想来自生活,手机、车、电子产品,出厂时有初始化
怎么样进行初始化?

方案 1: 显示调用方法

缺点: 易忘、麻烦; 显示调用 init, 不能完全解决问题

```
*/
```

```
class Test21
```

```
{
```

```
public:
```

```
    int m;
```

```
    int getM() const { return m; }
```

```
    void setM(int val) { m = val; }
```

```
    int n;
```

```
    int getN() const { return n; }
```

```
    void setN(int val) { n = val; }
```

```
public:
```

```
    int init(int m,int n)
```

```
{
```

```
        this->m = m;
        this->n = n;
        return 0;
    }
protected:
private:
};

int main()
{
    int rv = 0;
    Test21 t1;  //无参构造函数的调用方法
    Test21 t2;

    //t1.init(100, 200);
    //t2.init(300, 400);
    cout<<t1.getM()<<" "<<t1.getN()<<endl;
    cout<<t2.getM()<<" "<<t2.getN()<<endl;

    //定义对象数组时，没有机会进行显示初始化
    Test21 arr[3];
    //Test arr_2[3] = {Test(1,3), Test(), Test()};

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

3.2 构造函数的分类及调用

C++编译器给程序员提供的对象初始化方案，高端大气上档次。

//有参数构造函数的三种调用方法

```
class Test
{
private:
    int a;
    int b;

public:

    //无参数构造函数
    Test()
    {
        ;
    }
};
```

```
    }

    //带参数的构造函数
    Test(int a, int b)
    {
        ;
    }
    //赋值构造函数
    Test(const Test &obj)
    {
        ;
    }

public:
    void init(int _a, int _b)
    {
        a = _a;
        b = _b;
    }
};
```

1 无参数构造函数

调用方法： Test t1, t2;

2 有参构造函数

有参构造函数的三种调用方法

```
//有参数构造函数的三种调用方法
class Test5
{
private:
    int a;
public:
    //带参数的构造函数
    Test5(int a)
    {
        printf("\na:%d", a);
    }
    Test5(int a, int b)
    {
        printf("\na:%d b:%d", a, b);
    }
};
```



```
    }
public:
};

int main55()
{
    Test5 t1(10); //c++编译器默认调用有参构造函数 括号法
    Test5 t2 = (20, 10); //c++编译器默认调用有参构造函数 等号法
    Test5 t3 = Test5(30); //程序员手工调用构造函数 产生了一个对象 直接调用构造构造函数法

    system("pause");
    return 0;
}
```

3 拷贝构造函数调用时机

赋值构造函数的四种调用场景（调用时机）

第 1 和第 2 个调用场景

```
#include "iostream"
using namespace std;

class AA
{
public:
    AA() //无参构造函数 默认构造函数
    {
        cout<<"我是构造函数，自动被调用了"<<endl;
    }
    AA(int _a) //有参构造函数 默认构造函数
    {
        a = _a;
    }
    AA(const AA &obj2)
    {
        cout<<"我也是构造函数，我是通过另外一个对象 obj2，来初始化我自己"<<endl;
        a = obj2.a + 10;
    }
    ~AA()
    {
        cout<<"我是析构函数，自动被调用了"<<endl;
    }
    void getA()
    {
```

```
        printf("a:%d \n", a);
    }
protected:
private:
    int a;
};
//单独搭建一个舞台
void ObjPlay01()
{
    AA a1; //变量定义

    //赋值构造函数的第一个应用场景
    //用对象 1 初始化 对象 2
    AA a2 = a1; //定义变量并初始化 //初始化法

    a2 = a1; //用 a1 来=号给 a2 编译器给我们提供的浅 copy
}
```

第二个应用场景

//单独搭建一个舞台

void ObjPlay02()

```
{
    AA a1(10); //变量定义

    //赋值构造函数的第一个应用场景
    //用对象 1 初始化 对象 2
    AA a2(a1); //定义变量并初始化 //括号法

    //a2 = a1; //用 a1 来=号给 a2 编译器给我们提供的浅 copy
    a2.getA();
}
//注意: 初始化操作 和 等号操作 是两个不同的概念
```

第 3 个调用场景

```
#include "iostream"
using namespace std;

class Location
{
public:
    Location( int xx = 0 , int yy = 0 )
    {
        X = xx ;   Y = yy ;   cout << "Constructor Object.\n" ;
    }
}
```

```

    Location( const Location & p )           //复制构造函数
    {
        X = p.X ;   Y = p.Y ;   cout << "Copy_constructor called." << endl ;
    }
    ~Location()
    {
        cout << X << "," << Y << " Object destroyed." << endl ;
    }
    int  GetX () { return X ; }           int GetY () { return Y ; }
private :   int   X , Y ;
};

//alt + f8 排版
void f ( Location  p )
{
    cout << "Funtion:" << p.GetX() << "," << p.GetY() << endl ;
}

void mainobjplay()
{
    Location A ( 1, 2 ) ; //形参是一个元素，函数调用，会执行实参变量初始化形参变量
    f ( A ) ;
}

void main()
{
    mainobjplay();
    system("pause");
}

```

第 4 个调用场景

第四个应用场景

```

#include "iostream"
using namespace std;
class Location
{
public:
    Location( int xx = 0 , int yy = 0 )
    {
        X = xx ;   Y = yy ;   cout << "Constructor Object.\n" ;
    }
    Location( const Location & p )           //复制构造函数
    {

```

```
        X = p.X;  Y = p.Y;   cout << "Copy_constructor called." << endl ;
    }
    ~Location()
    {
        cout << X << "," << Y << " Object destroyed." << endl ;
    }
    int  GetX () { return X ; }      int GetY () { return Y ; }
private:  int  X , Y ;
};

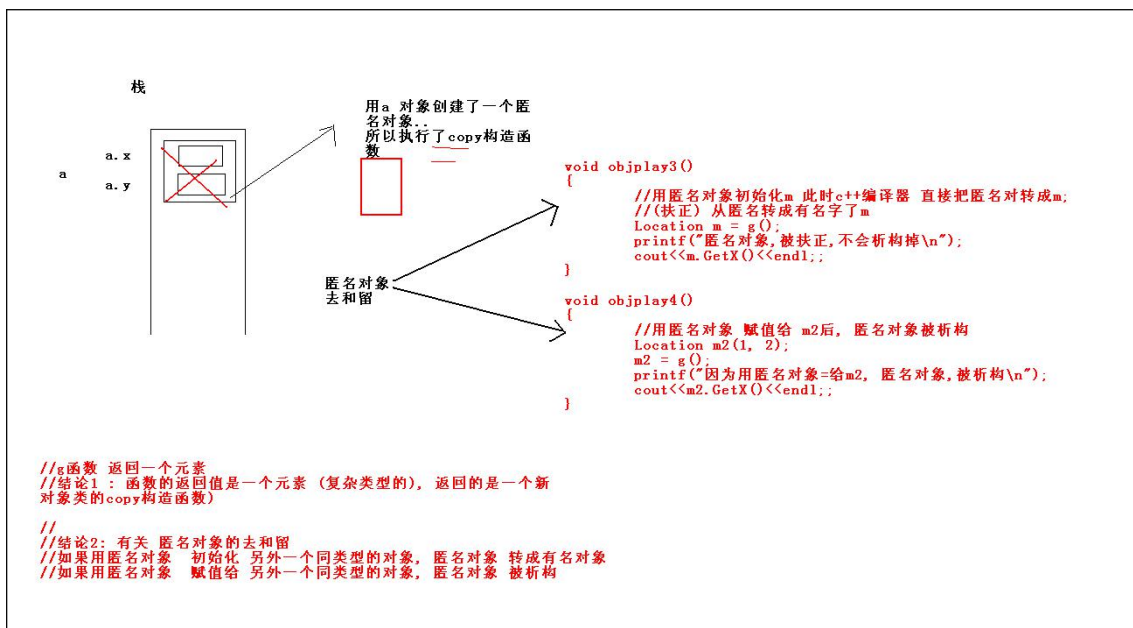
//alt + f8 排版
void f ( Location  p )
{
    cout << "Funtion:" << p.GetX() << "," << p.GetY() << endl ;
}

Location g()
{
    Location A(1, 2);
    return A;
}

//对象初始化操作 和 =等号操作 是两个不同的概念
//匿名对象的去和留，关键看，返回时如何接
void mainobjplay()
{
    //若返回的匿名对象，赋值给另外一个同类型的对象，那么匿名对象会被析构
    //Location B;
    //B = g();  //用匿名对象 赋值 给 B 对象，然后匿名对象析构

    //若返回的匿名对象，来初始化另外一个同类型的对象，那么匿名对象会直接转成新的
    对象
    Location B = g();
    cout<<"传智扫地僧测试"<<endl;
}

void main()
{
    mainobjplay();
    system("pause");
}
```



4 默认构造函数

二个特殊的构造函数

1) 默认无参构造函数

当类中没有定义构造函数时，编译器默认提供一个无参构造函数，并且其函数体为空

2) 默认拷贝构造函数

当类中没有定义拷贝构造函数时，编译器默认提供一个默认拷贝构造函数，简单的进行成员变量的值复制

3.3 构造函数调用规则研究

1) 当类中没有定义任何一个构造函数时，c++编译器会提供默认无参构造函数和默认拷贝构造函数

2) 当类中定义了拷贝构造函数时，c++编译器不会提供无参数构造函数

3) 当类中定义了任意的非拷贝构造函数(即: 当类中提供了有参构造函数或无参构造函数)，c++编译器不会提供默认无参构造函数

4) 默认拷贝构造函数成员变量简单赋值

总结：只要你写了构造函数，那么你必须用。

构造析构阶段性总结

1) 构造函数是 C++ 中用于初始化对象状态的特殊函数

2) 构造函数在对象创建时自动被调用

3) 构造函数和普通成员函数都遵循重载规则

4) 拷贝构造函数是对象正确初始化的重要保证

5) 必要的时候，必须手工编写拷贝构造函数

=====》1 个对象的初始化讲完了，增加一个案例。

3.4 深拷贝和浅拷贝

- 默认复制构造函数可以完成对象的数据成员值简单的复制
- 对象的数据资源是由指针指示的堆时，默认复制构造函数仅作指针值复制

1 浅拷贝问题抛出和分析

深拷贝浅拷贝现象出现的原因

2 浅拷贝程序 C++提供的解决方法

显示提供 copy 构造函数

显示操作重载=号操作，不使用编译器提供的浅 copy

```
class Name
{
public:
    Name(const char *pname)
    {
        size = strlen(pname);
        pName = (char *)malloc(size + 1);
        strcpy(pName, pname);
    }
    Name(Name &obj)
    {
        //用 obj 来初始化自己
        pName = (char *)malloc(obj.size + 1);
        strcpy(pName, obj.pName);
        size = obj.size;
    }
    ~Name()
    {
        cout<<"开始析构"<<endl;
        if (pName!=NULL)
        {
            free(pName);
            pName = NULL;
            size = 0;
        }
    }

    void operator=(Name &obj3)
```

```
{
    if (pName != NULL)
    {
        free(pName);
        pName = NULL;
        size = 0;
    }
    cout<<"测试有没有调用我。。。。"<<endl;

    //用 obj3 来=自己
    pName = (char *)malloc(obj3.size + 1);
    strcpy(pName, obj3.pName);
    size = obj3.size;
}

protected:
private:
    char *pName;
    int size;
};

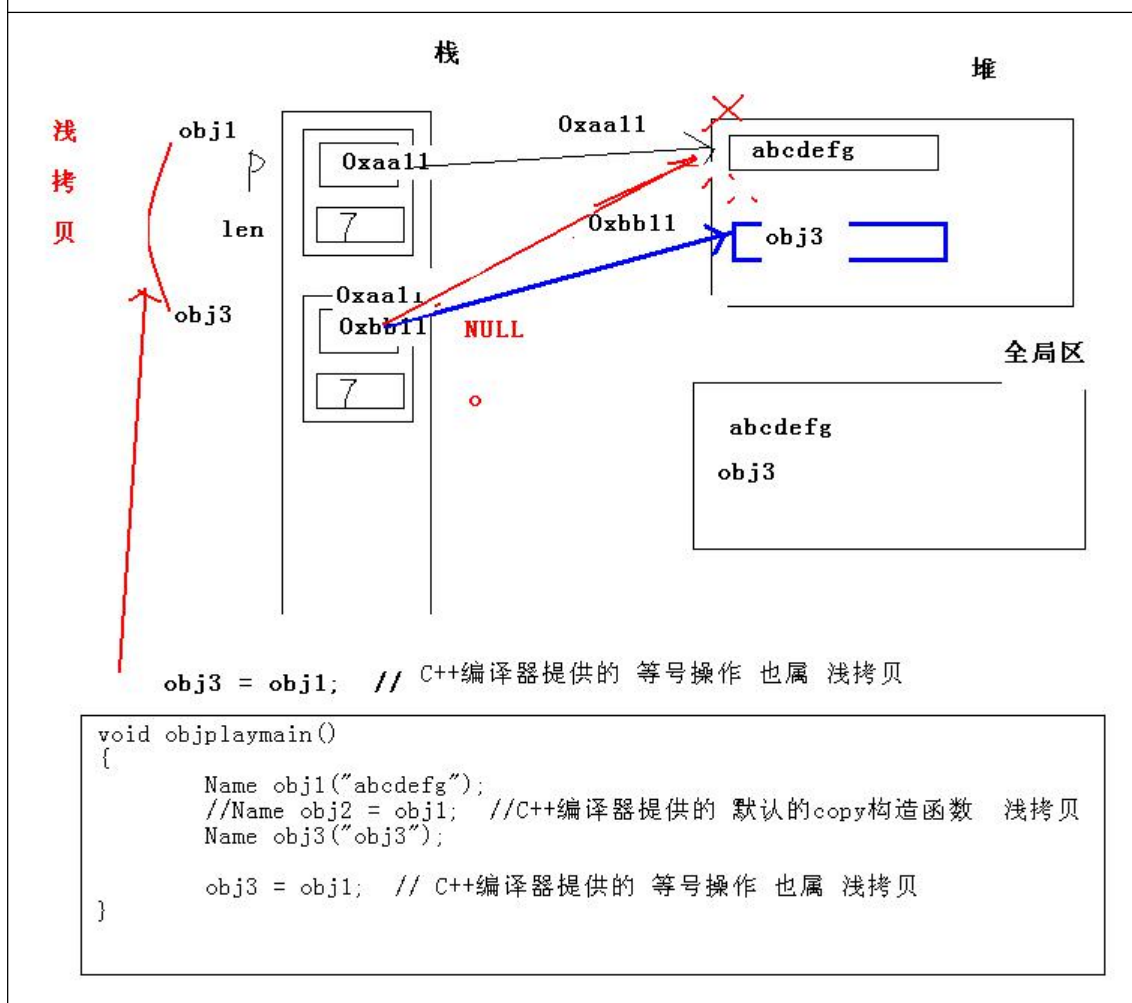
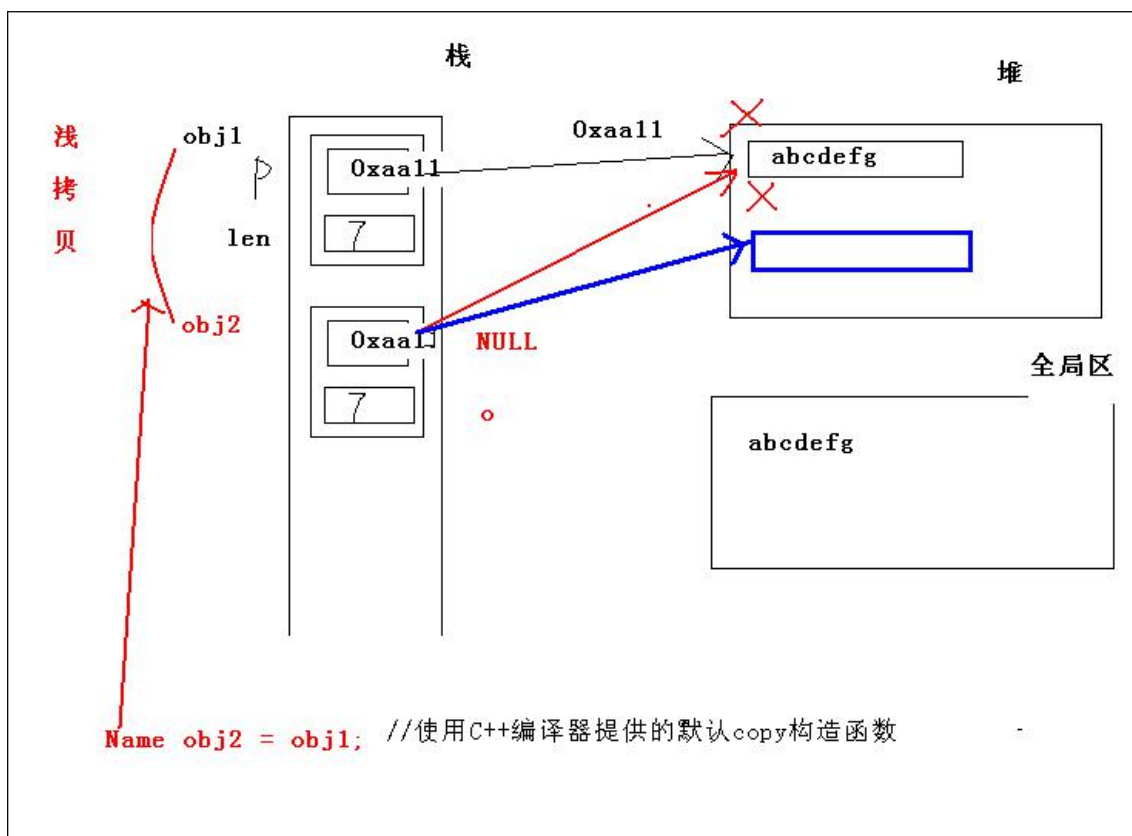
//对象的初始化 和 对象之间=号操作是两个不同的概念
void playObj()
{
    Name obj1("obj1.....");
    Name obj2 = obj1; //obj2 创建并初始化

    Name obj3("obj3...");

    //重载=号操作符
    obj2 = obj3; //!=号操作

    cout<<"业务操作。。。5000"<<endl;
}

void main61()
{
    playObj();
    system("pause");
}
```



3.5 多个对象构造和析构

1 对象初始化列表

1) 对象初始化列表出现原因

1. 必须这样做：

如果我们有一个类成员，它本身是一个类或者是一个结构，而且这个成员它只有一个带参数的构造函数，没有默认构造函数。这时要对这个类成员进行初始化，就必须调用这个类成员的带参数的构造函数，

如果没有初始化列表，那么他将无法完成第一步，就会报错。

2、类成员中若有 `const` 修饰，必须在对象初始化的时候，给 `const int m` 赋值

当类成员中含有一个 `const` 对象时，或者是一个引用时，他们也必须要通过成员初始化列表进行初始化，

因为这两种对象要在声明后马上初始化，而在构造函数中，做的是对他们的赋值，这样是不被允许的。

2) C++ 中提供初始化列表对成员变量进行初始化

语法规则

```
Constructor::Constructor() : m1(v1), m2(v1,v2), m3(v3)
{
    // some other assignment operation
}
```

3) 注意概念

初始化：被初始化的对象正在创建

赋值：被赋值的对象已经存在

4) 注意：

成员变量的初始化顺序与声明的顺序相关，与在初始化列表中的顺序无关

初始化列表先于构造函数的函数体执行

```
/*
1 C++ 中提供了初始化列表对成员变量进行初始化
2 使用初始化列表出现原因：
1. 必须这样做：
如果我们有一个类成员，它本身是一个类或者是一个结构，而且这个成员它只有一个带参数的构造函数，
而没有默认构造函数，这时要对这个类成员进行初始化，就必须调用这个类成员的带参数的构造函数，
如果没有初始化列表，那么他将无法完成第一步，就会报错。
*/
```

2、类成员中若有 `const` 修饰，必须在对象初始化的时候，给 `const int m` 赋值
当类成员中含有一个 `const` 对象时，或者是一个引用时，他们也必须要通过成员初始化列表进行初始化，
因为这两种对象要在声明后马上初始化，而在构造函数中，做的是对他们的赋值，这样是不被允许的。

```
*/
```

```
//总结 构造和析构的调用顺序
```

```
#include "iostream"
using namespace std;

class ABC
{
public:
    ABC(int a, int b, int c)
    {
        this->a = a;
        this->b = b;
        this->c = c;
        printf("a:%d,b:%d,c:%d \n", a, b, c);
        printf("ABC construct ..\n");
    }
    ~ABC()
    {
        printf("a:%d,b:%d,c:%d \n", a, b, c);
        printf("~ABC() ..\n");
    }
protected:
private:
    int a;
    int b;
    int c;
};

class MyD
{
public:
    MyD():abc1(1,2,3),abc2(4,5,6),m(100)
    //MyD()
    {
        cout<<"MyD()"<<endl;
```

```
    }
    ~MyD()
    {
        cout<<"~MyD()"<<endl;
    }

protected:
private:
    ABC abc1; //c++编译器不知道如何构造 abc1
    ABC abc2;
    const int m;
};

int run()
{
    MyD myD;
    return 0;
}

int main_dem03()
{

    run();
    system("pause");
    return 0;
}
```

3.6 构造函数和析构函数的调用顺序研究

构造函数与析构函数的调用顺序

- 1) 当类中有成员变量是其它类的对象时，首先调用成员变量的构造函数，调用顺序与声明顺序相同；之后调用自身类的构造函数
- 2) 析构函数的调用顺序与对应的构造函数调用顺序相反

3.7 构造函数和析构函数综合练习

通过训练，把所学知识点都穿起来

1 构造析综合训练

demo10_构造析构练习强化.cpp （讲解）

展示分析过程，注意赋值构造函数的调用

2 匿名对象强化训练

demo10_构造析构练习强化.cpp

- 1) 匿名对象生命周期
- 2) 匿名对象的去和留

3 匿名对象强化训练

- 3) 构造中调用构造

demo11_匿名对象练习强化.cpp

构造函数中调用构造函数，是一个整脚的行为。

3.8 对象的动态建立和释放

1 new 和 delete 基本语法

1) 在软件开发过程中，常常需要动态地分配和撤销内存空间，例如对动态链表中结点的插入与删除。在 C 语言中是利用库函数 `malloc` 和 `free` 来分配和撤销内存空间的。C++ 提供了较简便而功能较强的运算符 `new` 和 `delete` 来取代 `malloc` 和 `free` 函数。

注意： `new` 和 `delete` 是运算符，不是函数，因此执行效率高。

2) 虽然为了与 C 语言兼容，C++ 仍保留 `malloc` 和 `free` 函数，但建议用户不用 `malloc` 和 `free` 函数，而用 `new` 和 `delete` 运算符。`new` 运算符的例子：

`new int;` //开辟一个存放整数的存储空间，返回一个指向该存储空间的地址(即指针)

`new int(100);` //开辟一个存放整数的空间，并指定该整数的初值为 100，返回一个指向该存储空间的地址

`new char[10];` //开辟一个存放字符数组(包括 10 个元素)的空间，返回首元素的地址

`new int[5][4];` //开辟一个存放二维整型数组(大小为 5*4)的空间，返回首元素的地址

`float *p=new float (3.14159);` //开辟一个存放单精度数的空间，并指定该实数的初值为 3.14159，将返回的该空间的地址赋给指针变量 p

- 3) `new` 和 `delete` 运算符使用的一般格式为：

➤ **new** 运算符动态分配堆内存

使用形式：**指针变量 = new 类型 (常量) ;**

指针变量 = new 类型[表达式] ;

作用：从堆分配一块“类型”大小的存储空间，返回首地址

其中：“常量”是初始化值，可缺省

创建数组对象时，不能为对象指定初始值

➤ **delete** 运算符释放已分配的内存空间

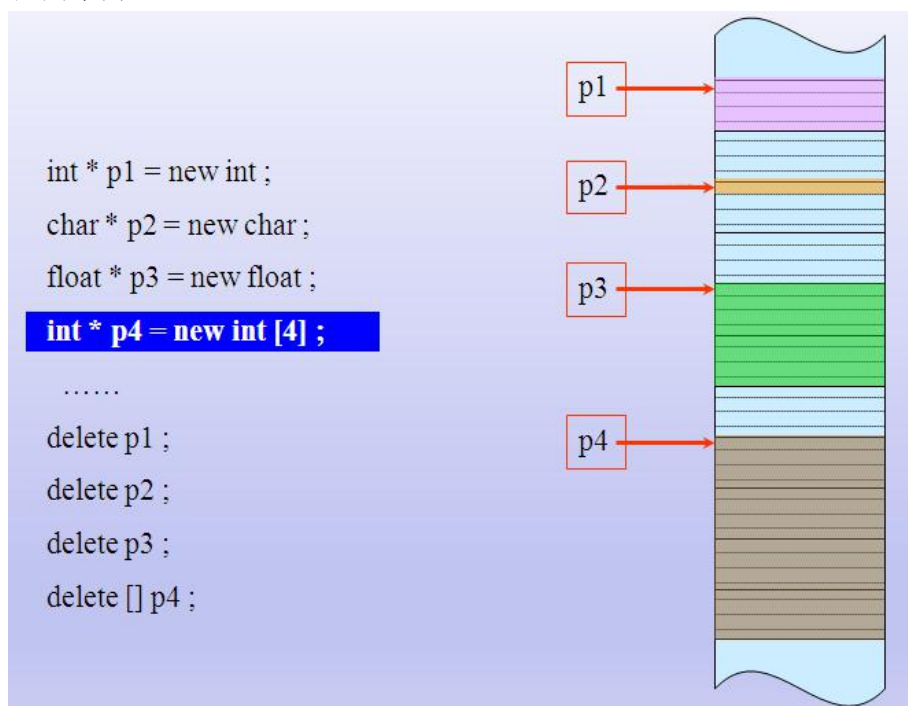
使用形式：**delete 指针变量 ;**

delete [] 指针变量 ;

其中：“指针变量”必须是一个 **new** 返回的指针

用 **new** 分配数组空间时不能指定初值。如果由于内存不足等原因而无法正常分配空间，则 **new** 会返回一个空指针 **NULL**，用户可以根据该指针的值判断分配空间是否成功。

4) 应用举例



2 类对象的动态建立和释放

使用类名定义的对象都是静态的，在程序运行过程中，对象所占的空间是不能随时释放的。但有时人们希望在需要用到对象时才建立对象，在不需要用该对象时就撤销它，释放它所占的内存空间以供别的数据使用。这样可提高内存空间的利用率。

C++中，可以用 **new** 运算符动态建立对象，用 **delete** 运算符撤销对象

比如：

```
Box *pt; //定义一个指向 Box 类对象的指针变量 pt
```

```
pt=new Box; //在 pt 中存放了新建对象的起始地址
```

在程序中就可以通过 pt 访问这个新建的对象。如

```
cout<<pt->height; //输出该对象的 height 成员
```

```
cout<<pt->volume( ); //调用该对象的 volume 函数，计算并输出体积
```

C++还允许在执行 new 时，对新建立的对象进行初始化。如

```
Box *pt=new Box(12,15,18);
```

这种写法是把上面两个语句(定义指针变量和用 new 建立新对象)合并为一个语句，并指定初值。这样更精炼。

新对象中的 height, width 和 length 分别获得初值 12,15,18。调用对象既可以通过对象名，也可以通过指针。

在执行 new 运算时，如果内存量不足，无法开辟所需的内存空间，目前大多数 C++编译系统都使 new 返回一个 0 指针值。只要检测返回值是否为 0，就可判断分配内存是否成功。

ANSI C++标准提出，在执行 new 出现故障时，就“抛出”一个“异常”，用户可根据异常进行有关处理。但 C++标准仍然允许在出现 new 故障时返回 0 指针值。当前，不同的编译系统对 new 故障的处理方法是不同的。

在不再需要使用由 new 建立的对象时，可以用 delete 运算符予以释放。如

```
delete pt; //释放 pt 指向的内存空间
```

这就撤销了 pt 指向的对象。此后程序不能再使用该对象。

如果用一个指针变量 pt 先后指向不同的动态对象，应注意指针变量的当前指向，以免删错了对象。在执行 delete 运算符时，在释放内存空间之前，自动调用析构函数，完成有关善后清理工作。

3 编程实践

```
//1 malloc free 函数      c 关键字  
// new delete 操作符号    c++的关键字
```

```
//2 new 在堆上分配内存 delete  
//分配基础类型 、分配数组类型、分配对象
```

```
//3 new 和 malloc 深入分析  
混用测试、异同比较  
结论： malloc 不会调用类的构造函数  
Free 不会调用类的析构函数
```

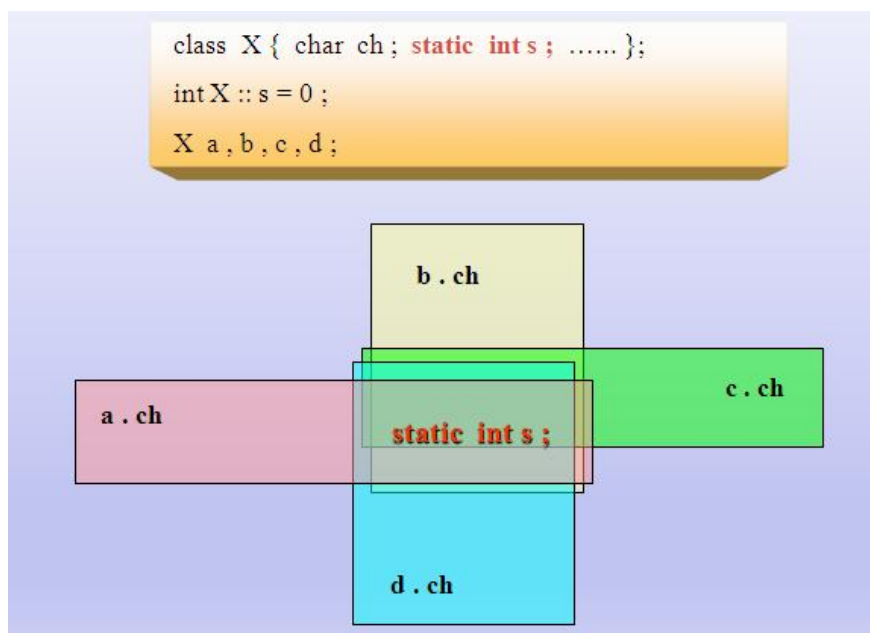
4 静态成员变量成员函数

思考：每个变量，拥有属性。有没有一些属性，归所有对象拥有？

4.1 静态成员变量

1) 定义静态成员变量

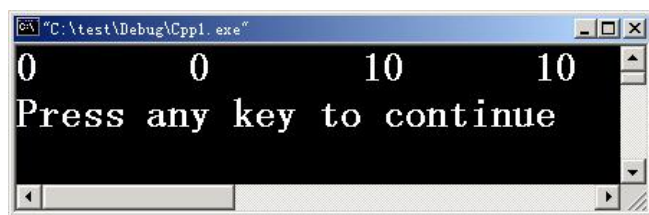
- 关键字 **static** 可以用于说明一个类的成员，静态成员提供了一个同类对象的共享机制
- 把一个类的成员说明为 **static** 时，这个类无论有多少个对象被创建，这些对象共享这个 **static** 成员
- 静态成员局部于类，它不是对象成员



例如：

```
#include<iostream>
using namespace std;
class counter
{
    static int num; //声明与定义静态数据成员
public:
    void setnum ( int i ) { num = i; } //成员函数访问静态数据成员
    void shownum() { cout << num << '\t'; }
};
int counter::num = 0; //声明与定义静态数据成员
void main ()
{
    counter a, b;
    a.shownum(); //调用成员函数访问私有静态数据成员
    b.shownum();
    a.setnum(10);
    a.shownum();
    b.shownum();
}
```

从结果可以看出，访问的是同一个静态数据成员



2) 使用静态成员变量

// 例 5-14 使用公有静态数据成员

```
#include<iostream.h>
class counter
{ public :
    counter (int a) { mem = a; }
    int mem;          //公有数据成员
    static int Smem ; //公有静态数据成员
};
int counter :: Smem = 1; //初始值为1
void main()
{   counter c(5);
    int i ;
    for( i = 0 ; i < 5 ; i ++ )
        { counter::Smem += i ;
          cout << counter::Smem << '\t' ; //访问静态成员变量方法 2
        }
    cout<<endl;
    cout<<"c.Smem = "<<c.Smem<<endl; //访问静态成员变量方法 1
    cout<<"c.mem = "<<c.mem<<endl;
}
```

4.2 静态成员函数

1) 概念

- 静态成员函数数冠以关键字 **static**
- 静态成员函数提供不依赖于类数据结构的操作，它没有 **this** 指针
- 在类外调用静态成员函数用 “**类名 ::**” 作限定词，或通过对象调用

2) 案例

例如

```
class X
{
    int DatMem;
public:
    static void StaFun(int i, X *ptr);
};

void X::StaFun(int i, X *ptr)
{
    ptr->DatMem = i;    // 正确
}
```

```
void g()
{
    X obj;
    X::StaFun(1, &obj);    // 正确
    obj.StaFun(1, &obj);    // 正确
}
```

都表示
静态成员函数的地址

3) 疑难问题：静态成员函数中，不能使用普通变量。

//静态成员变量属于整个类的，分不清楚，是那个具体对象的属性。

```
class BB
{
public:
    int getC()
    {
        return c;
    }
    void setC(int myc)
    {
        c = myc;
    }
    //静态成员函数是属于整个类，
    //类的静态数据成员函数，不能调用普通成员变量a，
    //C++编译器无法确认是b1.a b2.a ....
    static void getMem()
    {
        //cout<<a<<endl;
        cout<<c<<endl;
    }
protected:
private:
    int a;
    int b;
    static int c;
};

int BB::c = 0;

//static修饰的变量，是属于类，所有的对象都能共享用。
void main()
{
    BB b1;
    BB b2;
    cout<<b2.getC()<<endl;;
    b1.setC(100);
    cout<<b2.getC()<<endl;;
    system("pause");
}
```

对象 n

类BB

是那个对象的a

4.3 综合训练

5 C++面向对象模型初探

前言

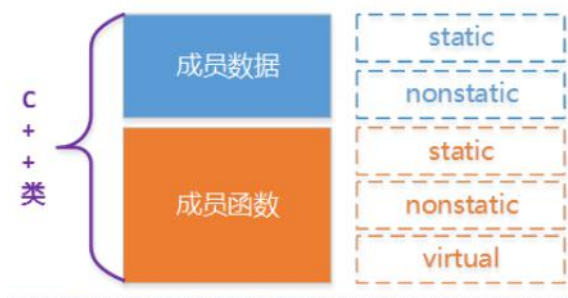
C++对象模型可以概括为以下 2 部分：

1. 语言中直接支持面向对象程序设计的部分，主要涉及如构造函数、析构函数、虚函数、继承（单继承、多继承、虚继承）、多态等等。

2. 对于各种支持的底层实现机制。

在 c 语言中，“数据”和“处理数据的操作（函数）”是分开来声明的，也就是说，语言本身并没有支持“数据和函数”之间的关联性。在 c++中，通过抽象数据类型（abstract data type, ADT），在类中定义数据和函数，来实现数据和函数直接的绑定。

概括来说，在 C++类中有两种成员数据：static、nonstatic；三种成员函数：static、nonstatic、virtual。



5.1 基础知识

C++中的 class 从面向对象理论出发，将变量(属性)和函数(方法)集中定义在一起，用于描述现实世界中的类。从计算机的角度，程序依然由数据段和代码段构成。

C++编译器如何完成面向对象理论到计算机程序的转化？

换句话：C++编译器是如何管理类、对象、类和对象之间的关系

具体的说：具体对象调用类中的方法，那，c++编译器是如何区分，是那个具体的类，调用这个方法那？

思考一下程序结果

```
#include "iostream"
```

```
using namespace std;
```

```
class C1
```

```
{
```

```
public:
```

```
    int i;  //4
```

```
    int j; //4
```

```
    int k;  //4
```

```
protected:
```

```
private:
```

```
}; //12
```

```
class C2
```

```
{
```

```
public:
```

```
int i;
int j;
int k;
static int m; //4
public:
    int getK() const { return k; } //4
    void setK(int val) { k = val; } //4

protected:
private:
}; //24 16 12(铁钉的不对)

struct S1
{
    int i;
    int j;
    int k;
}; //

struct S2
{
    int i;
    int j;
    int k;
    static int m;
}; //

int main()
{
    printf("c1:%d \n", sizeof(C1));
    printf("c2:%d \n", sizeof(C2));
    printf("s1:%d \n", sizeof(S1));
    printf("s2:%d \n", sizeof(S2));

    system("pause");
}
```

5.2 编译器对属性和方法的处理机制

通过上面的案例，我们可以得出：

- 1) C++类对象中的成员变量和成员函数是分开存储的
成员变量：

普通成员变量：存储于对象中，与 struct 变量有相同的内存布局和字节对齐方式

静态成员变量：存储于全局数据区中

成员函数：存储于代码段中。

问题出来了：**很多对象共用一块代码？代码是如何区分具体对象的那？**

换句话说：int getK() const { return k; }，代码是**如何区分**，具体 obj1、obj2、obj3 对象的 **k 值**？

2) C++编译器对普通成员函数的内部处理

```
class Test
{
private:
    int mI;

public:
    Test(int i)
    {
        mI = i;
    }

    int getI()
    {
        return mI;
    }

    static void Print()
    {
        printf("This is class Test.\n");
    }
};

Test a(10);
a.getI();
Test::Print();

struct Test
{
    int mI;
};

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

int Test_getI(Test* pThis)
{
    return pThis->mI;
}

void Test_Print()
{
    printf("This is class Test.\n");
}

Test a;
Test_initialize(&a, 10);
Test_getI(&a);
Test_Print();
```

请仔细思考，并说出你的总结！

5.3 总结

1、C++类对象中的成员变量和成员函数是分开存储的。C 语言中的内存四区模型仍然有效！

2、**C++中类的普通成员函数都隐式包含一个指向当前对象的 this 指针。**

3、静态成员函数、成员变量属于类

静态成员函数与普通成员函数的区别

静态成员函数不包含指向具体对象的指针

普通成员函数包含一个指向具体对象的指针

5.4 this 指针

```

#include<iostream.h> // 例5-5
class Simple
{ int x, y;
public:
    void setXY ( int a, int b) { x = a ; y = b ; }
    void printXY() { cout << x << " " << y << endl ; } ;
    void setXY (Simple * const this, int a, int b) { this->x = a ; this->y = b ; }
};

void main()
{ Simple obj1, obj2, obj3 ;
  obj1 . setXY ( 10, 15, obj1 ) ;
  obj1 . printXY() ;
  obj2 . setXY ( 20, 25 ) ;
  obj2 . printXY() ;
  obj3 . setXY ( 30, 35 ) ;
  obj3 . printXY() ;
}
  
```

成员函数隐含定义 **this** 指针
接受调用对象的地址

obj1.x obj1.y
 obj2.x obj2.y
 obj3.x obj3.y

实验 1: 若类成员函数的形参 和 类的属性, 名字相同, 通过 **this** 指针来解决。

实验 2: 类的成员函数可通过 **const** 修饰, 请问 **const** 修饰的是谁

5.5 全局函数 PK 成员函数

- 1、把全局函数转化成成员函数, 通过 **this** 指针隐藏左操作数

Test add(Test &t1, Test &t2)===》Test add(Test &t2)

- 2、把成员函数转换成全局函数, 多了一个参数

void printAB()===》void printAB(Test *pthis)

- 3、函数返回元素和返回引用

Test& add(Test &t2) /*this //函数返回引用

{

 this->a = this->a + t2.getA();

 this->b = this->b + t2.getB();

 return *this; /*操作让 this 指针回到元素状态

}

Test add2(Test &t2) /*this //函数返回元素

{

 //t3 是局部变量

```
Test t3(this->a+t2.getA(), this->b + t2.getB());  
return t3;  
}
```

6 友元

6.1 友元函数

```
class A  
{ private:  
    int i;  
    friend void FriendFun(A *, int);  
public:  
    void MemberFun(int);  
};  
...  
void FriendFun( A * ptr , int x )  
    { ptr -> i = x ; } ;  
void A:: MemberFun( int x )  
    { i = x ; } ;
```

说明语句位置
与访问描述无关

```
class A  
{ private:  
    int i;  
    friend void FriendFun(A *, int);  
public:  
    void MemberFun(int);  
};  
...  
void FriendFun( A * ptr , int x )  
    { ptr -> i = x ; } ;  
void A:: MemberFun( int x )  
    { i = x ; } ;
```

友元函数通过对象参数
访问私有数据成员

例如

```
class A1
{
public:
    A1()
    {
        a1 = 100;
        a2 = 200;
    }
    int getA1()
    {
        return this->a1;
    }
    //声明一个友元函数
    friend void setA1(A1 *p, int a1); //这个函数是这个类的好朋友

protected:
private:
    int a1;
    int a2;
};

void setA1(A1 *p, int a1)
{
    p->a1 = a1;
}

void main()
{
    A1 mya1;
    cout<<mya1.getA1()<<endl;
    setA1(&mya1, 300); //通过友元函数 修改 A 类的私有属性
    cout<<mya1.getA1()<<endl;

    system("pause");
}
```

```
#include<iostream>
```

```
#include<math.h>
```

```
class Point
```

```
{ public:
```

```
    Point(double xi, double yi) { X = xi ; Y = yi ;}
```

```
    double GetX() { return X ; }
```

```
    double GetY() { return Y ; }
```

```
    friend double Distance ( Point & a, Point & b ) ;
```

```
private: double X, Y ;
```

```
};
```

```
double Distance(Point & a, Point & b )
```

```
{ double dx = a.X - b.X ;
```

```
  double dy = a.Y - b.Y ;
```

```
  return sqrt ( dx * dx + dy * dy ) ;
```

```
}
```

```
void main()
```

```
{ Point p1( 3.0, 5.0 ) , p2( 4.0, 6.0 ) ;
```

```
  double d = Distance ( p1, p2 ) ;
```

```
  cout << "This distance is " << d << endl ;
```

```
}
```

用友元函数计算两点之间的距离

定义函数 Distance

6.2 友元类

- 若 B 类是 A 类的友元类，则 B 类的所有成员函数都是 A 类的友元函数
- 友元类通常设计为一种对数据操作或类之间传递消息的辅助类

演示友员类

```
#include<iostream>
class A
{ friend class B; //B是A的友元
public:
    void Display() { cout<< x << endl; };
private:
    int x;
};
class B
{ public:
    void Set ( int i ) { Aobject . x = i; }
    void Display () { Aobject . Display (); }
private:
    A Aobject; //B中有个A, A是B的子属性
};
void main()
{ B Bobject;
  Bobject . Set ( 100 );
  Bobject . Display ();
}
```

通过类成员
访问A类的私有数据成员

7 强化训练

1 static 关键字强化训练题

- 某商店经销一种货物。货物购进和卖出时以箱为单位，各箱的重量不一样，因此，商店需要记录目前库存的总重量。现在用 C++ 模拟商店货物购进和卖出的情况。

```
#include "iostream"
using namespace std;

class Goods
{
public:
    Goods ( int w ) { weight = w; total_weight += w; }
    ~ Goods() { total_weight -= weight; }
    int Weight() { return weight; };
    static int TotalWeight() { return total_weight; }
    Goods *next;
private:
    int weight;
    static int total_weight;
}
```

```
};

int Goods::total_weight = 0 ;

//r 尾部指针
void purchase( Goods * &f, Goods * &r, int w )
{
    Goods *p = new Goods(w) ;
    p -> next = NULL ;
    if ( f == NULL ) f = r = p ;
    else { r -> next = p ; r = r -> next ; } //尾部指针下移或新结点变成尾部结点
}

void sale( Goods * &f , Goods * &r )
{
    if ( f == NULL ) { cout << "No any goods!\n" ; return ; }
    Goods *q = f ; f = f -> next ; delete q ;
    cout << "saled.\n" ;
}

void main()
{
    Goods * front = NULL , * rear = NULL ;
    int w ; int choice ;
    do
    {
        cout << "Please choice:\n" ;
        cout << "Key in 1 is purchase,\nKey in 2 is sale,\nKey in 0 is over.\n" ;
        cin >> choice ;
        switch ( choice ) // 操作选择
        {
            case 1 : // 键入 1，购进 1 箱货物
                { cout << "Input weight: " ;
                  cin >> w ;
                  purchase( front, rear, w ) ; // 从表尾插入 1 个结点
                  break ;
                }
            case 2 : // 键入 2，售出 1 箱货物
                { sale( front, rear ) ; break ; } // 从表头删除 1 个结点
            case 0 : break ; // 键入 0，结束
        }
        cout << "Now total weight is:" << Goods::TotalWeight() << endl ;
    } while ( choice ) ;
}
```

2 数组类封装

目标：解决实际问题，训练构造函数、copy 构造函数等，为操作符重载做准备

数组类的测试

```
#include "iostream"
#include "Array.h"
using namespace std;

int main()
{
    Array a1(10);

    for(int i=0; i<a1.length(); i++)
    {
        a1.setData(i, i);
    }

    for(int i=0; i<a1.length(); i++)
    {
        printf("array %d: %d\n", i, a1.getData(i));
    }

    Array a2 = a1;

    for(int i=0; i<a2.length(); i++)
    {
        printf("array %d: %d\n", i, a2.getData(i));
    }

    system("pause");
    return 0;
}
```

数组类的头文件

```
#ifndef _MYARRAY_H_
#define _MYARRAY_H_

class Array
{
private:
    int mLength;
    int* mSpace;
```

```
public:
    Array(int length);
    Array(const Array& obj);
    int length();
    void setData(int index, int value);
    int getData(int index);
    ~Array();
};

#endif
```

3 小结

- 类通常用关键字 **class** 定义。类是数据成员和成员函数的封装。类的实例称为对象。
- 结构类型用关键字 **struct** 定义，是由不同类型数据组成的数据类型。
- 类成员由 **private**, **protected**, **public** 决定访问特性。**public** 成员集称为接口。
- 构造函数在创建和初始化对象时自动调用。析构函数则在对象作用域结束时自动调用。
- 重载构造函数和复制构造函数提供了创建对象的不同初始化方式。
- 静态成员是局部于类的成员，提供一种同类对象的共享机制。
- 友员用关键字 **friend** 声明。友员是对类操作的一种辅助手段。一个类的友员可以访问该类各种性质的成员。
- 链表是一种重要的动态数据结构，可以在程序运行时创建或撤消数据元素。

8 运算符重载

8.1 概念

什么是运算符重载

运算符重载使得用户自定义的数据以一种更简洁的方式工作

能表示为

$c1 = c1 + c2;$?

定义
运算符重载函数

例如

```
int x, y ;  
y = x + y ;  
complex c1 , c2 ;  
c1 = Cadd (c1 , c2) ;  
matrix m1 , m2 ;  
m1 = Madd ( m1 , m2 ) ;
```

// 矩阵类对象
// 调用函数计算两个矩阵的和

所谓重载，就是重新赋予新的含义。函数重载就是对一个已有的函数赋予新的含义，使之实现新功能，因此，一个函数名就可以用来代表不同功能的函数，也就是“一名多用”。

运算符也可以重载。实际上，我们已经在不知不觉之中使用了运算符重载。例如，大家都已习惯于用加法运算符“+”对整数、单精度数和双精度数进行加法运算，如 $5+8$ ， $5.8+3.67$ 等，其实计算机对整数、单精度数和双精度数的加法操作过程是很不相同的，但由于 C++ 已经对运算符“+”进行了重载，所以就能适用于 `int`, `float`, `double` 类型的运算。

又如“<<”是 C++ 的位运算中的位移运算符（左移），但在输出操作中又是与流对象 `cout` 配合使用的流插入运算符，“>>”也是位移运算符（右移），但在输入操作中又是与流对象 `cin` 配合使用的流提取运算符。这就是运算符重载(operator overloading)。C++ 系统对“<<”和“>>”进行了重载，用户在不同的场合下使用它们时，作用是不同的。对“<<”和“>>”的重载处理是放在头文件 `stream` 中的。因此，如果要在程序用“<<”和“>>”作流插入运算符和流提取运算符，必须在本文件模块中包含头文件 `stream`（当然还应当包括“`using namespace std`”）。现在要讨论的问题是：用户能否根据自己的需要对 C++ 已提供的运算符进行重载，赋予它们新的含义，使之一名多用？

运算符重载入门技术推演

1 为什么会用运算符重载机制

用复数类举例

```
//Complex c3 = c1 + c2;
```

//原因 Complex 是用户自定义类型，编译器根本不知道如何进行加减

//编译器给提供了一种机制，让用户自己去完成，自定义类型的加减操作。。。。。

//这个机制就是运算符重载机制

2 运算符重载的本质是一个函数

```
class Complex
{
public:
    int a;
    int b;
    friend Complex operator+(Complex &c1, Complex &c2);
public:
    Complex(int a=0, int b=0)
    {
        this->a = a;
        this->b = b;
    }

public:
    void printCom()
    {
        cout<<a<<" + "<<b<<"i "<<endl;
    }

private:
};

/*
Complex myAdd(Complex &c1, Complex &c2)
{
    Complex tmp(c1.a+ c2.a, c1.b + c2.b);
    return tmp;
}
*/

Complex operator+(Complex &c1, Complex &c2)
{
    Complex tmp(c1.a+ c2.a, c1.b + c2.b);
```

```
        return tmp;
    }

void main()
{
    Complex c1(1, 2), c2(3, 4);

    //Complex c3 = c1 + c2; //用户自定义类型 编译器无法让变量相加
    //Complex myAdd(Complex &c1, Complex &c2);

    //1 普通函数
    //Complex c3 = myAdd(c1, c2);
    //c3.printCom();

    //2 operator+ 函数名称
    //Complex c3 = operator+(c1, c2);
    //c3.printCom();

    //3 +替换 函数名
    Complex c3 = c1 + c2; //思考 C++编译器如何支持操作符重载机制的 (根据类型)
    c3.printCom();
    {
        int a = 0, b = 0, c; //基础类型 C++编译器知道如何加减
        c = a + b;
    }

    //4 把 Complex 类变成私有属性
    //友元函数的应用场景
    //friend Complex operator+(Complex &c1, Complex &c2);

    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

8.2 运算符重载的限制

可以重载的运算符

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

不能重载的算符

.	::	.*	?:	sizeof
---	----	----	----	--------

重载运算符函数可以对运算符作出新的解释，但原有基本语义不变：

- 不改变运算符的优先级
- 不改变运算符的结合性
- 不改变运算符所需要的操作数
- 不能创建新的运算符

8.3 运算符重载编程基础

➤ 运算符函数是一种特殊的成员函数或友员函数

➤ 成员函数的语法形式为：

```
    类型 类名 :: operator op ( 参数表 )
注意 {
    // 相对于该类定义的操作
}
```

（注：原图中“返回值”、“关键字”、“函数名”、“操作数”被圈出，且“返回值”前有“注意”字样）

➤ 一个运算符被重载后，原有意义没有失去，只是定义了相对一特定类的一个新运算符

例如：

```
//全局函数 完成 +操作符 重载
Complex operator+(Complex &c1, Complex &c2)
//类成员函数 完成 -操作符 重载
Complex operator-(Complex &c2)
```

运算符重载的两种方法

用成员或友员函数重载运算符

➤ 运算符函数可以重载为成员函数或友员函数

➤ 关键区别在于成员函数具有 this 指针，友员函数没有 this 指针

➤ 不管是成员函数还是友员函数重载，运算符的使用方法相同。

但传递参数的方式不同，实现代码不同，应用场合也不同

用成员或友员函数重载运算符

2. 二元运算符

ObjectL *op* ObjectR

➤重载为成员函数，解释为：

ObjectL . *operator op* (ObjectR)

左操作数由ObjectL通过this指针传递，右操作数由参数ObjectR传递

➤重载为友员函数，解释为：

operator op (ObjectL, ObjectR)

左右操作数都由参数传递

例如 1:

```
//通过类成员函数完成-操作符重载
//函数声明 Complex operator-(Complex &c2)
//函数调用分析
//用类成员函数实现-运算符重载
Complex c4 = c1 - c2;
c4.printCom();
//c1.operator-(c2);
```

例如 2:

```
//通过全局函数方法完成+操作符重载
//函数声明 Complex operator+(Complex &c1, Complex &c2)
//函数调用分析
int main()
{
    Complex c1(1, 2), c2(3, 4);
    //Complex c31 = operator+(c1, c2);
    Complex c3 = c1 + c2;
    c3.printCom();
}
```

例如3: 学员自己练习 实现 * /

用成员或友员函数重载运算符

1. 一元运算符

Object op 或 op Object

➤ 重载为成员函数，解释为：

Object . operator op ()

操作数由对象Object通过this指针隐含传递

➤ 重载为友员函数，解释为：

operator op (Object)

操作数由参数表的参数Object提供

例如 3

//前置++操作符 用全局函数实现

Complex& operator++(Complex &c1)

```
{
    c1.a ++;
    c1.b ++;
    return c1;
}
```

//调用方法

++c1 ; **//=>** 需要写出操作符重载函数原形
c1.printCom();

//运算符重载函数名定义

//首先承认操作符重载是一个函数 定义函数名->operator++

//分析函数参数 根据左右操作数的个数, ->operator++(Complex &c1)

//分析函数返回值 -> Complex& operator++(Complex &c1) 返回它自身

例如 4

//4.1 前置--操作符 成员函数实现

Complex& operator--()

```
{
    this->a--;
    this->b--;
    return *this;
}
```

```
//4.2 调用方法
--c1;
c1.printCom();
//4.3 前置—运算符重载函数名定义
//c1.operator--()
```

例如 5

```
//5.1 //后置++ 操作符 用全局函数实现
Complex operator++(Complex &c1, int)
{
    Complex tmp = c1;
    c1.a++;
    c1.b++;
    return tmp;
}
//5.2 调用方法
c1 ++; //先使用 后++
//5.3 后置++运算符重载函数名定义
Complex operator++(Complex &c1, int) //函数占位参数 和 前置++ 相区别
```

例如 6

```
//6.1 后置— 操作符 用类成员函数实现
Complex operator--(int)
{
    Complex tmp = *this;
    this->a--;
    this->b--;
    return tmp;
}
//6.2 调用方法
c1 --; //先使用 后--
//6.3 后置--运算符重载函数名定义
Complex operator--(int) //函数占位参数 和 前置-- 相区别
```

前置和后置运算符总结

C++中通过一个占位参数来区分前置运算和后置运算

重载 ++ 与 --

设 `A Aobject ;`

运算符 ++ 和 -- 有两种方式：

前置方式：

<code>++Aobject</code>	<code>--Aobject</code>
一元 成员函数 重载	<code>A::A operator++ ();</code>
解释为：	<code>Aobject.operator++();</code>
友元函数 重载	<code>friend A operator++ (A &);</code>
解释为：	<code>operator++(Aobject);</code>

后置方式：

<code>Aobject++</code>	<code>Aobject--</code>
二元 成员函数 重载	<code>A::A operator++ (int);</code>
解释为：	<code>Aobject.operator++(0);</code>
友元函数 重载：	<code>friend A operator++ (A &, int);</code>
解释为：	<code>operator++(Aobject, 0);</code>

伪参数

定义运算符重载函数名的步骤

全局函数、类成员函数方法实现运算符重载步骤

- 1) 要承认操作符重载是一个函数，写出函数名称 `operator+ ()`
- 2) 根据操作数，写出函数参数
- 3) 根据业务，完善函数返回值(看函数是返回引用 还是指针 元素)，及实现函数业务

友元函数实现操作符重载的应用场景

1) 友元函数和成员函数选择方法

- 当无法修改左操作数的类时，使用全局函数进行重载
- `=, [], ()` 和 `->` 操作符只能通过成员函数进行重载

2) 用友元函数 重载 << >> 操作符

- `istream` 和 `ostream` 是 C++ 的预定义流类
- `cin` 是 `istream` 的对象，`cout` 是 `ostream` 的对象
- 运算符 `<<` 由 `ostream` 重载为插入操作，用于输出基本类型数据
- 运算符 `>>` 由 `istream` 重载为提取操作，用于输入基本类型数据
- 用友元函数重载 `<<` 和 `>>`，输出和输入用户自定义的数据类型

a) 用全局函数方法实现 << 操作符

```
ostream& operator<<(ostream &out, Complex &c1)
{
```

```

        //out<<"12345，生活真是苦"<<endl;
        out<<c1.a<<" " + "<<c1.b<<"i "<<endl;
        return out;
    }
//调用方法
    cout<<c1;
//链式编程支持
    cout<<c1<<"abcc";
    //cout.operator<<(c1).operator<<("abcd");
//函数返回值充当左值 需要返回一个引用
b) 类成员函数方法无法实现 << 操作符重载
    //因拿到 cout 这个类的源码
    //cout.operator<<(c1);

```

3) 友元函数重载操作符使用注意点

a) 友元函数重载运算符常用于运算符的左右操作数类型不同的情况

友元函数重载运算符常用于运算符的左右操作数类型不同的情况

例如

```

class Complex
{
    int Real;    int Imag;
public:
    Complex( int a ) { Real = a; Imag = 0; }
    Complex( int a ,int b ) { Real = a; Imag = b; }
    Complex operator + ( Complex );
    .....
};

int f()
{
    Complex z( 2, 3 ), k( 3, 4 );
    z = z + 27;
    z = 27 + z;
    .....
}

```

27.operator+(z)

NO
27 不是Complex对象
不能调用函数

b) 其他

- 在第一个参数需要隐式转换的情形下，使用友元函数重载运算符是正确的选择
- 友元函数没有 this 指针，所需操作数都必须在参数表显式声明，很容易实现类型的隐式转换
- C++中不能用友元函数重载的运算符有
= () [] ->

4) 友元函数案例 vector 类

```

#include <iostream>
using namespace std;

```

```
//为 vector 类重载流插入运算符和提取运算符
class vector
{
public :
    vector( int size =1 ) ;
    ~vector() ;
    int & operator[] ( int i ) ;
    friend ostream & operator << ( ostream & output , vector & ) ;
    friend istream & operator >> ( istream & input, vector & ) ;
private :
    int * v ;
    int len ;
};

vector::vector( int size )
{
    if (size <= 0 || size > 100 )
    {
        cout << "The size of " << size << " is null !\n" ; abort() ;
    }
    v = new int[ size ] ; len = size ;
}

vector :: ~vector()
{
    delete[] v ;
    len = 0 ;
}

int &vector::operator[] ( int i )
{
    if( i >=0 && i < len ) return v[ i ] ;
    cout << "The subscript " << i << " is outside !\n" ; abort() ;
}

ostream & operator << ( ostream & output, vector & ary )
{
    for(int i = 0 ; i < ary.len ; i ++ )
        output << ary[ i ] << " " ;
    output << endl ;
    return output ;
}

istream & operator >> ( istream & input, vector & ary )
{

```

```
        for( int i = 0 ; i < ary.len ; i ++ )
            input >> ary[ i ] ;
        return  input ;
    }

void main()
{
    int k ;
    cout << "Input the length of vector A :\n" ;
    cin >> k ;
    vector A( k ) ;
    cout << "Input the elements of vector A :\n" ;
    cin >> A ;
    cout << "Output the elements of vector A :\n" ;
    cout << A ;
    system("pause");
}
```

8.4 运算符重载提高

1 运算符重载机制

C++编译器是如何支持操作符重载机制的?

2 重载赋值运算符=

- 赋值运算符重载用于对象数据的复制
- `operator=` 必须重载为成员函数
- 重载函数原型为:
类型 & 类名 :: operator= (const 类名 &);

案例：完善 Name 类，支持=号操作。


```

#include<iostream.h>
#include<string.h>
class Name
{ public :
    Name ( char *pN );
    Name( const Name & );           //复制构造函数
    Name& operator=( const Name& ); //重载赋值运算符
    ~Name();
protected:
    char *pName;
    int size;
};

void main()
{ Name Obj1( "ZhangSan" );
  Name Obj2 = Obj1;               // 调用复制构造函数
  Name Obj3( "NoName" );
  Obj3 = Obj2 = Obj1;             // 调用重载赋值运算符函数
}

```

定义Name类的重载赋值函数

修改对象时
调用重载赋值运算符函数

结论:

- 1 //先释放旧的内存
- 2 返回一个引用
- 3 =操作符 从右向左

```

//obj3 = obj1;   // C++编译器提供的 等号操作 也属 浅拷贝
// obj4 = obj3 = obj1
//obj3.operator=(obj1)

```

```

Name& operator=(Name &obj1)
{
    //1 先释放obj3旧的内存
    if (this->m_p != NULL)
    {
        delete[] m_p;
        m_len = 0;
    }
    //2 根据obj1分配内存大小
    this->m_len = obj1.m_len;
    this->m_p = new char [m_len+1];

    //3把obj1赋值给obj3
    strcpy(m_p, obj1.m_p);
    return *this;
}

```

3 重载数组下标运算符[]

重载[]和()运算符

- 运算符 [] 和 () 是二元运算符
- [] 和 () 只能用成员函数重载，不能用友元函数重载

重载下标运算符 []

[] 运算符用于访问数据对象的元素

重载格式 类型 类 :: operator[] (类型) ;

设 x 是类 X 的一个对象，则表达式

x[y]

可被解释为

x.operator[](y)

1. 重载下标运算符 []

// 例6-7

```
#include<iostream.h>
```

```
class vector
```

```
{ public:
```

```
    vector ( int n ) { v = new int [ n ] ; size = n ; }
```

```
    ~vector ( ) { delete [ ] v ; size = 0 ; }
```

```
    int & operator [] ( int i ) { return v [ i ] ; }
```

```
private:
```

```
    int * v ;    int size ;
```

```
};
```

```
void main ( )
```

```
{ vector a ( 5 ) ;
```

```
  a [ 2 ] = 12 ;
```

```
  cout << a [ 2 ] << endl ;
```

```
}
```

返回元素的引用

this -> v[i]

1. 重载下标运算符 []

// 例6-7

```
#include<iostream.h>
class vector
{ public:
    vector ( int n ) { v = new int [ n ]; size = n ; }
    ~vector ( ) { delete [] v ; size = 0 ; }
    int & operator [] ( int i ) { return v [ i ]; }
private:
    int * v ;    int size ;
};
void main ( )
{ vector a ( 5 );
  a [ 2 ] = 12 ;
  cout << a [ 2 ] << endl ;
}
```

返回引用的函数调用
作左值

4 重载函数调用符 ()

() 运算符用于函数调用

重载格式 类型 类 :: operator() (表达式表) ;

例1

设 x 是类 X 的一个对象，则表达式

$x (arg1, arg2, \dots)$

可被解释为

$x . operator () (arg1, arg2, \dots)$

案例：

// 例2：用重载()运算符实现数学函数的抽象

```
#include <iostream>
class F
{ public:
    double operator ( ) ( double x, double y );
};
double F :: operator ( ) ( double x, double y )
{ return x * x + y * y ; }
void main ( )
{
    F f ;
    f.getA();
    cout << f ( 5.2, 2.5 ) << endl ; // f . operator() ( 5.2, 2.5 )
}
```

比较普通成员函数

//例3 用重载()运算符实现 pk 成员函数

```
#include <iostream.h>

class F
{ public :
    double memFun ( double x, double y );
};

double F::memFun ( double x, double y )
{ return x * x + y * y ; }

void main ( )
{
    F f ;
    cout << f.memFun ( 5.2 , 2.5 ) << endl ;
}
```

5 为什么不要重载&&和||操作符

理论知识：

- 1) &&和||是 C++ 中非常特殊的操作符
- 2) &&和||内置实现了短路规则
- 3) 操作符重载是靠函数重载来完成的
- 4) 操作数作为函数参数传递
- 5) C++ 的函数参数都会被求值，无法实现短路规则

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Test
{
    int i;
public:
    Test(int i)
    {
        this->i = i;
    }

    Test operator+ (const Test& obj)
    {
        Test ret(0);

        cout << "执行+号重载函数" << endl;
```

```
        ret.i = i + obj.i;
        return ret;
    }

    bool operator&& (const Test& obj)
    {
        cout<<"执行&&重载函数"<<endl;
        return i && obj.i;
    }
};

// && 从左向右
void main()
{
    int a1 = 0;
    int a2 = 1;

    cout<<"注意：&&操作符的结合顺序是从左向右"<<endl;

    if( a1 && (a1 + a2) )
    {
        cout<<"有一个是假，则不在执行下一个表达式的计算"<<endl;
    }

    Test t1 = 0;
    Test t2 = 1;

    If ( t1 && (t1 + t2) )
    {
        //t1  && t1.operator(t2)
        // t1.operator( t1.operator(t2) )
        cout<<"两个函数都被执行了，而且是先执行了+"<<endl;
    }

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

8.5 运算符重载在项目开发中的应用

1 实现一个数组类

添加<< >>

2 实现一个字符串类

构造函数要求

```
MyString a;  
MyString a("dddd");  
MyString b = a;
```

常用的操作符

```
<< >> != == > < =
```

//C 语言中 没有字符串这种类型，是通过数组来模拟字符串
//C++中 我们来设计一个字符串 以零结尾的字符串

```
class MyString  
{  
    friend ostream& operator<<(ostream &out, const MyString &s);  
public: //构造和析构  
    MyString(int len = 0);  
    MyString(const char *p);  
    MyString(const MyString& obj);  
    ~MyString();  
  
public: //操作符重载  
    MyString& operator=(const char *p);  
    MyString& operator=(const MyString& obj);  
    char& operator[](int index) const;  
  
public:  
    bool operator==(const char* p) const;  
    bool operator!=(const char* p) const;  
    bool operator==(const MyString& s) const;  
    bool operator!=(const MyString& s) const;  
  
public: //string to c  
    char *c_str();  
    const char* c_str() const;  
    int length()
```

```
    {  
        return m_len;  
    }  
  
public:  
    int operator<(const char *p);  
    int operator>(const char *p);  
  
    int operator<(const MyString &s);  
    int operator>(const MyString &s);  
  
private:  
    int      m_len;  
    char *m_p;  
};
```

3 智能指针类编写

1 问题抛出

指针使用过程中，经常会出现内存泄漏和内存多次被释放常

2 解决方案：例如：boost 库的智能指针

项目开发中，要求开发者使用预先编写的智能指针类对象代替 C 语言中的原生指针

3 智能指针思想

工程中的智能指针是一个类模板

通过构造函数接管申请的内存

通过析构函数确保堆内存被及时释放

通过重载指针运算符* 和 -> 来模拟指针的行为

通过重载比较运算符 == 和 != 来模拟指针的比较

```
class Test  
{  
public:  
    Test()  
    {  
        this->a = 10;  
    }  
    void printT()  
    {  
        cout<<a<<endl;  
    }  
};
```

```
private:
    int a;
};

class MyTestPointer
{
public:
public:
    MyTestPointer()
    {
        p = NULL;
    }
    MyTestPointer(Test* p)
    {
        this->p = p;
    }
    ~MyTestPointer()
    {
        delete p;
    }
    Test* operator->()
    {
        return p;
    }
    Test& operator*()
    {
        return *p;
    }

protected:
    Test *p;
};

void main01_classp()
{
    Test *p = new Test;
    p->printT();
    delete p;

    MyTestPointer myp = new Test; //构造函数
    myp->printT(); //重载操作符 ->
};
```



```
class MyIntPtrter
{
public:
public:
    MyIntPtrter()
    {
        p = NULL;
    }
    MyIntPtrter(int* p)
    {
        this->p = p;
    }
    ~MyIntPtrter()
    {
        delete p;
    }
    int* operator->()
    {
        return p;
    }
    int& operator*()
    {
        return *p;
    }

protected:
    int *p;
};

void main02_intp()
{
    int *p = new int(100);
    cout<<*p<<endl;
    delete p;

    MyIntPtrter myp = new int(200);
    cout<<*myp<<endl; //重载*操作符
};
```

8.7 附录：运算符和结合性

附录 C 运算符和结合性

优先级	运 算 符	含 义	要 求 运 算 对象的个数	结 合 方 向
1	()	圆括号		自左至右
	[]	下标运算符		
	->	指向结构体成员运算符		
	·	结构体成员运算符		
2	!	逻辑非运算符	1 (单目运算符)	自右至左
	~	按位取反运算符		
	++	自增运算符		
	--	自减运算符		
	-	负号运算符		
	(类型)	类型转换运算符		
	*	指针运算符		
	&	取地址运算符		
	sizeof	长度运算符		
3	*	乘法运算符	2 (双目运算符)	自左至右
	/	除法运算符		
	%	求余运算符		
4	+	加法运算符	2 (双目运算符)	自左至右
	-	减法运算符		
5	<<	左移运算符	2 (双目运算符)	自左至右
	>>	右移运算符		
6	< <= > >=	关系运算符	2 (双目运算符)	自左至右

续表

优先级	运算符	含 义	要 求 运 算 对象的个数	结合方向
7	==	等于运算符	2 (双目运算符)	自左至右
	!=	不等于运算符		
8	&	按位与运算符	2 (双目运算符)	自左至右
9	^	按位异或运算符	2 (双目运算符)	自左至右
10		按位或运算符	2 (双目运算符)	自左至右
11	&&	逻辑与运算符	2 (双目运算符)	自左至右
12		逻辑或运算符	2 (双目运算符)	自左至右
13	? :	条件运算符	3 (三目运算符)	自右至左
14	= += -= *= /= %= >>= <<= &= ^= =	赋值运算符	2 (双目运算符)	自右至左
15	,	逗号运算符 (顺序求值运算符)		自左至右

总结

操作符重载是 C++ 的强大特性之一

操作符重载的本质是通过函数扩展操作符的语义

operator 关键字是操作符重载的关键

friend 关键字可以对函数或类开发访问权限

操作符重载遵循函数重载的规则

操作符重载可以直接使用类的成员函数实现

=, [], () 和 -> 操作符只能通过成员函数进行重载

++ 操作符通过一个 **int** 参数进行前置与后置的重载

C++ 中不要重载 && 和 || 操作符

3、继承和派生

3.1 继承概念

面向对象程序设计有 4 个主要特点：抽象、封装、继承和多态性。我们已经讲解了类和对象，了解了面向对象程序设计的两个重要特征——数据抽象与封装，已经能够设计出基于对象的程序，这是面向对象程序设计的基础。

要较好地进行面向对象程序设计，还必须了解面向对象程序设计另外两个重要特征——继承性和多态性。本章主要介绍有关继承的知识，多态性将在后续章节中讲解。

继承性是面向对象程序设计最重要的特征，可以说，如果没有掌握继承性，就等于没有掌握类和对象的精华，就是没有掌握面向对象程序设计的真谛。

3.1.1 类之间的关系

has-A, uses-A 和 is-A

has-A 包含关系，用以描述一个类由多个“部件类”构成。实现 has-A 关系用类成员表示，即一个类中的数据成员是另一种已经定义的类。

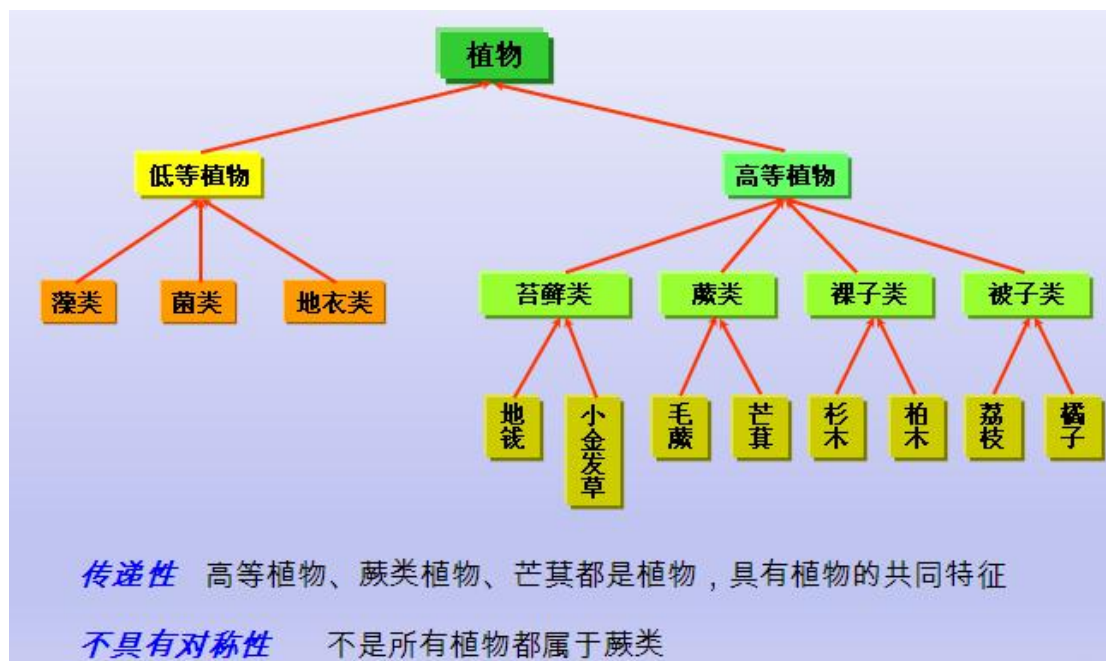
uses-A 一个类部分地使用另一个类。通过类之间成员函数的相互联系，定义友员或对象参数传递实现。

is-A 机制称为“继承”。关系具有传递性,不具有对称性。

3.1.2 继承关系举例

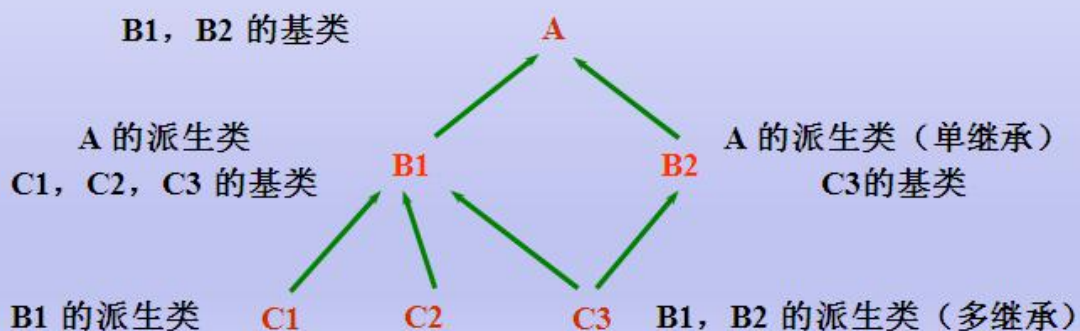
万事万物中皆有继承，是重要的现象

两个案例：1) 植物继承图；2) 程序员继承图



3.1.3 继承相关概念

- **继承** 是类之间定义的一种重要关系
- 一个 B 类继承 A 类，或称从类 A 派生类 B
类 A 称为基类（父类），类 B 称为派生类（子类）



3.1.4 派生类的定义

类继承关系的语法形式

```
class 派生类名 : 基类名表
{
    数据成员和成员函数声明
};
```

基类名表 构成

访问控制 基类名₁, **访问控制** 基类名₂, ..., **访问控制** 基类名_n

访问控制表示派生类对基类的继承方式，使用关键字：

public	公有继承
private	私有继承
protected	保护继承

注意：C++中的继承方式（public、private、protected）会影响子类的对外访问属性。

3.1.5 继承重要说明

- 1、子类拥有父类的所有成员变量和成员函数
- 4、子类可以拥有父类没有的方法和属性
- 2、子类就是一种特殊的父类
- 3、子类对象可以当作父类对象使用

3.2 派生类的访问控制

派生类继承了基类的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），但是这些成员的访问属性，在派生过程中是可以调整的。

3.2.1 单个类的访问控制

- 1、类成员访问级别（public、private、protected）
- 2、思考：类成员的访问级别只有 public 和 private 是否足够？

3.2.2 不同的继承方式会改变继承成员的访问属性

1) C++中的继承方式会影响子类的对外访问属性

public 继承：父类成员在子类中保持原有访问级别

private 继承：父类成员在子类中变为 private 成员

protected 继承：父类中 public 成员会变成 protected

父类中 protected 成员仍然为 protected

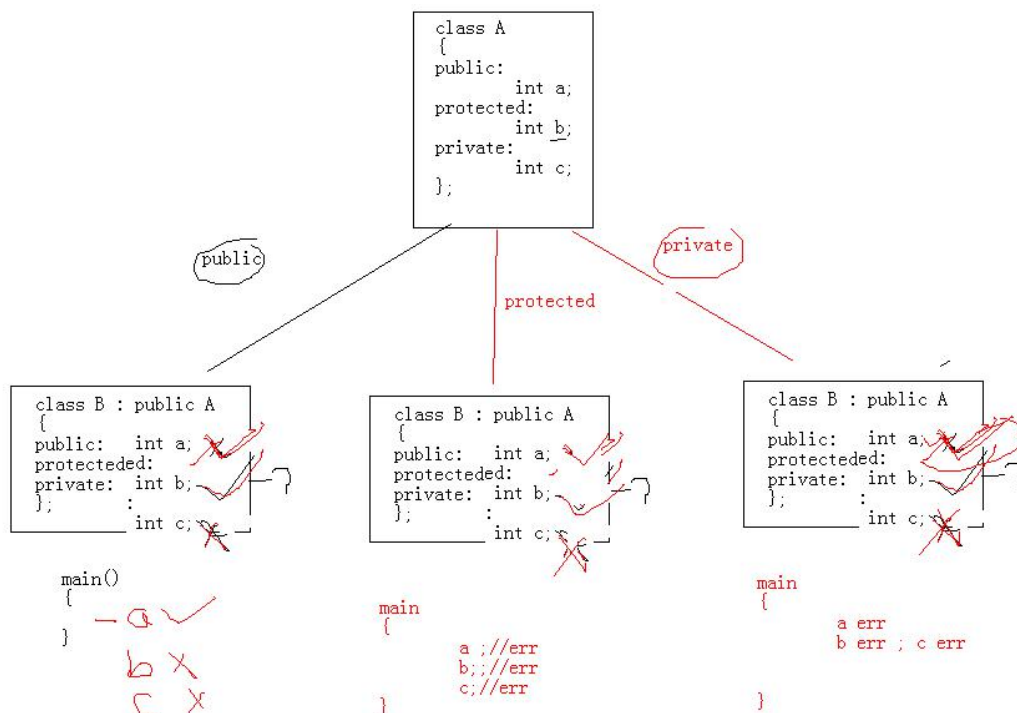
父类中 private 成员仍然为 private

2) private 成员在子类中依然存在，但是却无法访问到。不论种方式继承基类，派生类都不能直接使用基类的私有成员。

3) C++中子类对外访问属性表

继承方式	父类成员访问级别			
		public	protected	private
	public	public	protected	private
	protected	protected	protected	private
	private	private	private	Private

4) 继承中的访问控制



3.2.3 “三看”原则

C++中的继承方式（public、private、protected）会影响子类的对外访问属性

判断某一句话，能否被访问

- 1) 看调用语句，这句话写在子类的内部、外部
- 2) 看子类如何从父类继承（public、private、protected）
- 3) 看父类中的访问级别（public、private、protected）

3.2.3 派生类类成员访问级别设置的原则

思考：如何恰当的使用 public，protected 和 private 为成员声明访问级别？

- 1、需要被外界访问的成员直接设置为 public
- 2、只能在当前类中访问的成员设置为 private
- 3、只能在当前类和子类中访问的成员设置为 protected，protected 成员的访问权限介于 public 和 private 之间。

3.2.4 综合训练

练习：

public 继承不会改变父类对外访问属性；

private 继承会改变父类对外访问属性为 private；

protected 继承会部分改变父类对外访问属性。

结论：一般情况下 class B : public A

//类的继承方式对子类对外访问属性影响

```
#include <cstdlib>
```

```
#include <iostream>
```

```
using namespace std;
```

```
class A
```

```
{
```

```
private:
```

```
    int a;
```

```
protected:
```

```
    int b;
```

```
public:
```

```
    int c;
```

```
    A()
```

```
{
```

```
    a = 0;
```

```
    b = 0;
```

```
    c = 0;
```

```
}
```

```
    void set(int a, int b, int c)
```

```
{
```

```
        this->a = a;
```

```
        this->b = b;
```

```
        this->c = c;
```

```
}
```

```
};
```

```
class B : public A
```

```
{
```

```
public:
```

```
    void print()
```

```
{
```

```
        //cout<<"a = "<<a; //err
```

```
        cout<<"b = "<<b;
```

```
        cout<<"c = "<<endl;
```

```
}
```

```
};
```

```
class C : protected A
```

```
{
```



```
public:
    void print()
    {
        //cout<<"a = "<<a; //err
        cout<<"b = "<<b;
        cout<<"c = "<<endl;
    }
};

class D : private A
{
public:
    void print()
    {
        //cout<<"a = "<<a; //err
        cout<<"b = "<<b<<endl;
        cout<<"c = "<<c<<endl;
    }
};

int main_01(int argc, char *argv[])
{
    A aa;
    B bb;
    C cc;
    D dd;

    aa.c = 100; //ok
    bb.c = 100; //ok
    //cc.c = 100; //err 类的外部是什么含义
    //dd.c = 100; //err

    aa.set(1, 2, 3);
    bb.set(10, 20, 30);
    //cc.set(40, 50, 60); //ee
    //dd.set(70, 80, 90); //ee

    bb.print();
    cc.print();
    dd.print();

    system("pause");
    return 0;
}
```

3.3 继承中的构造和析构

3.3.1 类型兼容性原则

类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员。这样，公有派生类实际就具备了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。类型兼容规则中所指的替代包括以下情况：

- 子类对象可以当作父类对象使用
- 子类对象可以直接赋值给父类对象
- 子类对象可以直接初始化父类对象
- 父类指针可以直接指向子类对象
- 父类引用可以直接引用子类对象

在替代之后，派生类对象就可以作为基类的对象使用，但是只能使用从基类继承的成员。类型兼容规则是多态性的重要基础之一。

总结：子类就是特殊的父类 (`base *p = &child;`)

```
#include <cstdlib>
#include <iostream>

using namespace std;

/*
子类对象可以当作父类对象使用
    子类对象可以直接赋值给父类对象
    子类对象可以直接初始化父类对象
    父类指针可以直接指向子类对象
    父类引用可以直接引用子类对象
*/
//子类就是特殊的父类
class Parent03
{
protected:
    const char* name;
public:
    Parent03()
    {
        name = "Parent03";
    }

    void print()
    {
        cout<<"Name: "<<name<<endl;
```

```
    }  
};  
  
class Child03 : public Parent03  
{  
protected:  
    int i;  
public:  
    Child03(int i)  
    {  
        this->name = "Child2";  
        this->i = i;  
    }  
};  
  
int main()  
{  
    Child03 child03(1000);  
    //分别定义父类对象 父类指针 父类引用 child  
    Parent03 parent = child03;  
    Parent03* pp = &child03;  
    Parent03& rp = child03;  
  
    parent.print();  
    pp->print();  
    rp.print();  
    system("pause");  
    return 0;  
}
```

3.3.2 继承中的对象模型

类在 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得到的

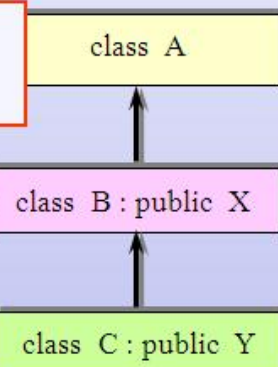
公有继承的测试

```

#include<iostream>
class A
{ public :
    void get_XY() { cout << "Enter two numbers of x, y : " ; cin >> x >> y ; }
    void put_XY() { cout << "x = " << x << ", y = " << y << '\n' ; }
    protected: int x, y ;
};
class B : public A
{ public :
    int get_S() { return s ; } ;
    void make_S() { s = x * y ; } ; // 使用基类数据成员
    protected: int s ;
};
class C : public B
{ public :
    void get_H() { cout << "Enter a number of h : " ; cin >> h ; }
    int get_V() { return v ; }
    void make_V() { make_S() ; v = get_S() * h ; } // 使用基类成员函数
    protected: int h, v ;
};

```

保护数据成员
在类层次中可见

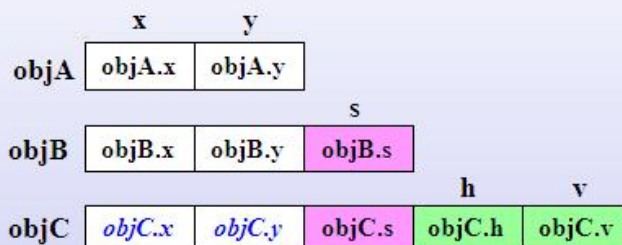


```

void main()
{ A objA ;
  B objB ;
  C objC ;
  cout << "It is object_A :\n" ;
  objA.get_XY() ;
  objA.put_XY() ;
  cout << "It is object_B :\n" ;
  objB.get_XY() ;
  objB.make_S() ;
  cout << "S = " << objB.get_S() << endl ;
  cout << "It is object_C :\n" ;
  objC.get_XY() ;
  objC.get_H() ;
  objC.make_V() ;
  cout << "V = " << objC.get_V() << endl ;
}

```

公有继承的测试



调用基类A成员函数
对 objC 的数据成员操作

问题：如何初始化父类成员？父类与子类的构造函数有什么关系

在子类对象构造时，需要调用父类构造函数对其继承得来的成员进行初始化

在子类对象析构时，需要调用父类析构函数对其继承得来的成员进行清理

```
#include <cstdlib>
#include <iostream>
using namespace std;

class Parent04
{
public:
    Parent04(const char* s)
    {
        cout<<"Parent04()"<<" "<<s<<endl;
    }

    ~Parent04()
    {
        cout<<"~Parent04()"<<endl;
    }
};

class Child04 : public Parent04
{
public:
    Child04() : Parent04("Parameter from Child!")
    {
        cout<<"Child04()"<<endl;
    }

    ~Child04()
    {
        cout<<"~Child04()"<<endl;
    }
};

void run04()
{
    Child04 child;
}

int main_04(int argc, char *argv[])
{
    run04();
}
```

```
system("pause");
return 0;
}
```

3.3.3 继承中的构造析构调用原则

- 1、子类对象在创建时会首先调用父类的构造函数
- 2、父类构造函数执行结束后，执行子类的构造函数
- 3、当父类的构造函数有参数时，需要在子类的初始化列表中显示调用
- 4、析构函数调用的先后顺序与构造函数相反

3.3.4 继承与组合混搭情况下，构造和析构调用原则

原则： 先构造父类，再构造成员变量、最后构造自己
先析构自己，在析构成员变量、最后析构父类
//先构造的对象，后释放

练习：demo05_extend_construct_destory.cpp

```
//子类对象如何初始化父类成员
//继承中的构造和析构
//继承和组合混搭情况下，构造函数、析构函数调用顺序研究
```

```
#include <iostream>

using namespace std;

class Object
{
public:
    Object(const char* s)
    {
        cout<<"Object()"<<" "<<s<<endl;
    }
    ~Object()
    {
        cout<<"~Object()"<<endl;
    }
};

class Parent : public Object
{
public:
```

```
    Parent(const char* s) : Object(s)
    {
        cout<<"Parent()"<<" "<<s<<endl;
    }
    ~Parent()
    {
        cout<<"~Parent()"<<endl;
    }
};

class Child : public Parent
{
protected:
    Object o1;
    Object o2;
public:
    Child() : o2("o2"), o1("o1"), Parent("Parameter from Child!")
    {
        cout<<"Child()"<<endl;
    }
    ~Child()
    {
        cout<<"~Child()"<<endl;
    }
};

void run05()
{
    Child child;
}

int main05(int argc, char *argv[])
{
    cout<<"demo05_extend_construct_destory.cpp"<<endl;
    run05();

    system("pause");
    return 0;
}
```

3.3.5 继承中的同名成员变量处理方法

- 1、当子类成员变量与父类成员变量同名时
- 2、子类依然从父类继承同名成员

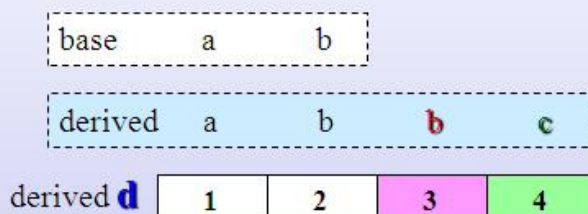
3、在子类中通过作用域分辨符::进行同名成员区分（在派生类中使用基类的同名成员，显式地使用类名限定符）

4、同名成员存储在内存中的不同位置

1. 重名数据成员

例：

```
class base
{ public :
    int a, b ;
};
class derived : public base
{ public :
    int b, c ;
};
void f()
{ derived d ;
  d . a = 1 ;
  d . base :: b = 2 ;
  d . b = 3 ;
  d . c = 4 ;
};
```



➤ 基类成员的作用域延伸到所有派生类

➤ 派生类的重名成员屏蔽基类的同名成员

2. 重名成员函数

```
#include<iostream.h>
class A
{ public:
    int a1, a2 ;
    A( int i1=0, int i2=0 ) { a1 = i1; a2 = i2; }
```

```
void print()
{ cout << "a1=" << a1 << '\t' << "a2=" << a2 << endl ; }
```

```
};
class B : public A
{ public:
    int b1, b2 ;
    B( int j1=1, int j2=1 ) { b1 = j1; b2 = j2; }
```

```
void print() //定义同名函数
{ cout << "b1=" << b1 << '\t' << "b2=" << b2 << endl ; }
```

```
void printAB()
{ A::print() ; //派生类对象调用基类版本同名成员函数
  print() ; //派生类对象调用自身的成员函数
}
```

```
};
void main()
{ B b ; b.A::print() ; b.printAB() ; }
```

派生类屏蔽基类同名成员函数
调用自身的成员函数

总结：同名成员变量和成员函数通过作用域分辨符进行区分

3.3.6 派生类中的 static 关键字

继承和 static 关键字在一起会产生什么现象哪？

理论知识

- 基类定义的静态成员，将被所有派生类共享
- 根据静态成员自身的访问特性和派生类的继承方式，在类层次体系中具有不同的访问性质（遵守派生类的访问控制）
- 派生类中访问静态成员，用以下形式显式说明：

类名 :: 成员

或通过对象访问 对象名 . 成员

在派生类中访问静态成员

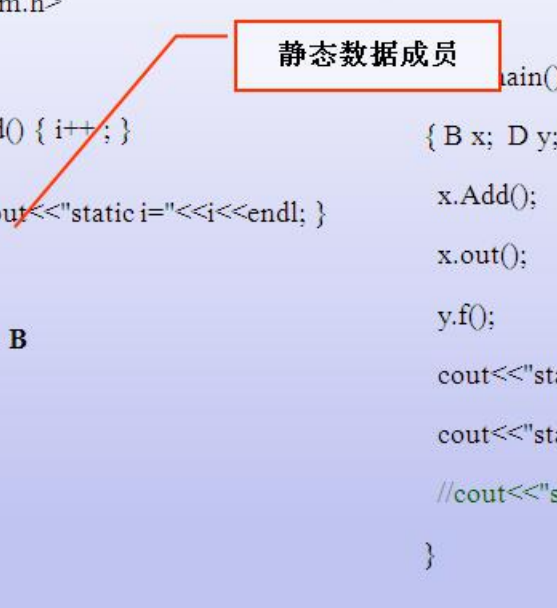
```
#include<iostream.h>

class B
{ public:
    static void Add() { i++; }
    static int i;
    void out() { cout<<"static i="<<i<<endl; }
};

int B::i=0;

class D : private B
{ public:
    void f();
    { i=5;
      Add();
      B::i++;
      B::Add();
    }
};

int main()
{ B x; D y;
  x.Add();
  x.out();
  y.f();
  cout<<"static i="<<B::i<<endl;
  cout<<"static i="<<x.i<<endl;
  //cout<<"static i="<<y.i<<endl;
}
```



//例7-5 在派生类中访问静态成员

```

#include<iostream.h>
class B
{ public:
    static void Add() { i++ ;}
    static int i;
    void out() { cout<<"static i="<<i<<endl; }
};
int B::i=0;
class D : private B
{ public:
    void f();
    { i=5;
      Add();
      B::i++;
      B::Add();
    }
};

void main()
{ B x; D y;
  x.Add();
  x.out();
  cout<<"static i="<<B::i<<endl;
  cout<<"static i="<<x.i<<endl;
  //cout<<"static i="<<y.i<<endl;
}

```

访问B类的静态成员

//例7-5 在派生类中访问静态成员

```

#include<iostream.h>
class B
{ public:
    static void Add() { i++ ;}
    static int i;
    void out() { cout<<"static i="<<i<<endl; }
};
int B::i=0;
class D : private B
{ public:
    void f();
    { i=5;
      Add();
      B::i++;
      B::Add();
    }
};

void main()
{ B x; D y;
  x.Add();
  x.out();
  y.f();
  cout<<"static i="<<B::i<<endl;
  cout<<"static i="<<x.i<<endl;
  //cout<<"static i="<<y.i<<endl;
}

```

访问B类的静态数据成员

总结:

- 1> static 函数也遵守 3 个访问原则
- 2> static 易犯错误 (不但要初始化, 更重要的显示的告诉编译器分配内存)
- 3> 构造函数默认为 private

3.4 多继承

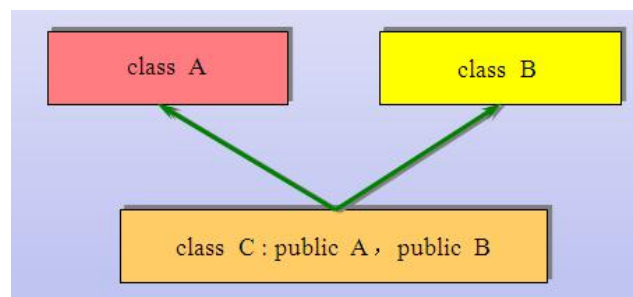
3.4.1 多继承的应用

多继承概念

- 一个类有多个直接基类的继承关系称为多继承
- 多继承声明语法

```
class 派生类名 : 访问控制 基类名 1, 访问控制 基类名 2, ..., 访问控制 基类名 n
{
    数据成员和成员函数声明
};
```

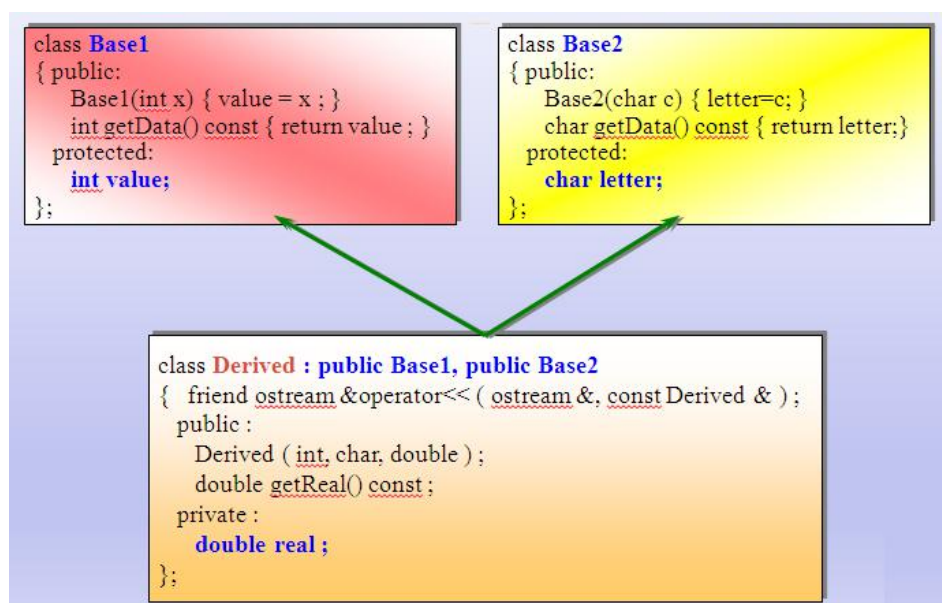
- 类 C 可以根据访问控制同时继承类 A 和类 B 的成员，并添加自己的成员



多继承的派生类构造和访问

- 多个基类的派生类构造函数可以用初始式调用基类构造函数初始化数据成员
- 执行顺序与单继承构造函数情况类似。多个直接基类构造函数执行顺序取决于定义派生类时指定的各个继承基类的顺序。
- 一个派生类对象拥有多个直接或间接基类的成员。不同名成员访问不会出现二义性。如果不同的基类有同名成员，派生类对象访问时应该加以识别。

多继承简单应用



多继承的简单应用

```

class Base1
{ public:
    Base1(int x) { value = x; }
    int getData() const { return value; }
protected:
    int value;
};

class Base2
{ public:
    Base2(char c) { letter=c; }
    char getData() const { return letter; }
protected:
    char letter;
};

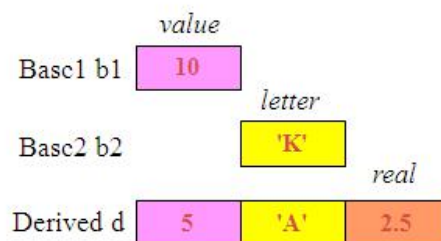
class Derived : public Base1, public Base2
{ friend ostream& operator<< (ostream&, const
public:
    Derived (int, char, double);
    double getReal() const;
private:
    double real;
};

```

```

void main()
{ Base1 b1 ( 10 );
  Base2 b2 ( 'k' );
  Derived d ( 5, 'A', 2.5 );
  :
  return;
}

```

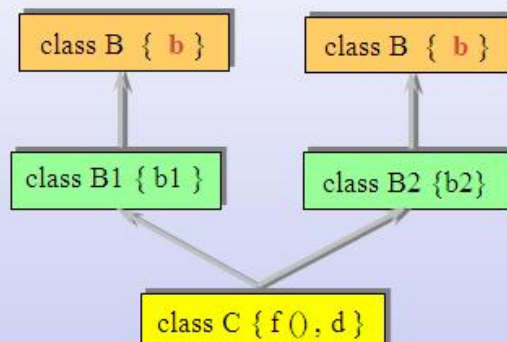


3.4.2 虚继承

如果一个派生类从多个基类派生，而这些基类又有一个共同的基类，则在对该基类中声明的名字进行访问时，可能产生二义性

例如：

```
class B { public: int b; };
class B1 : public B { private: int b1; };
class B2 : public B { private: int b2; };
class C : public B1, public B2
{ public: int f(); private: int d; };
```



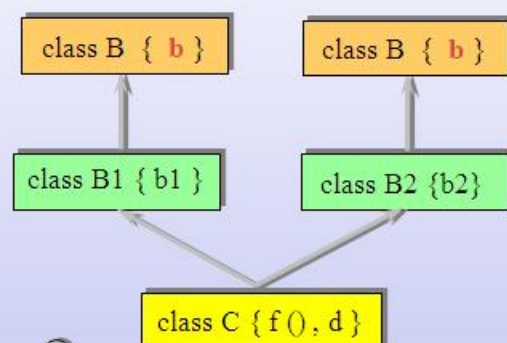
有：

```
C c;
c.B; // error
c.B::b; // error, 从哪里继承的?
c.B1::b; // ok, 从B1继承的
c.B2::b; // ok, 从B2继承的
```

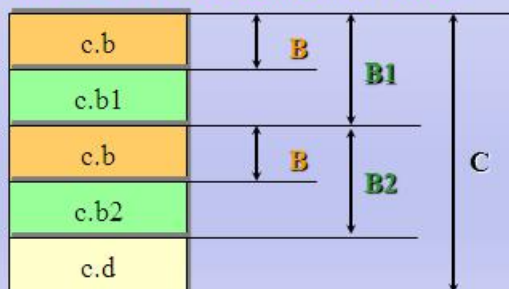
分析：

例如：

```
class B { public: int b; };
class B1 : public B { private: int b1; };
class B2 : public B { private: int b2; };
class C : public B1, public B2
{ public: int f(); private: int d; };
```



多重派生类C的对象的存储结构示意



建立C类的对象时，B的构造函数将被调用两次：一次由B1调用，另一次由B2调用，以初始化C类的对象中所包含的两个B类的子对象

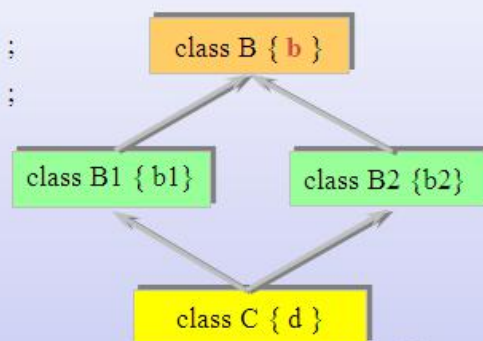
总结：

- 如果一个派生类从多个基类派生，而这些基类又有一个共同的基类，则在对该基类中声明的名字进行访问时，可能产生二义性
- 如果在多条继承路径上有一个公共的基类，那么在继承路径的某处汇合点，这个公共基类就会在派生类的对象中产生多个基类子对象

- 要使这个公共基类在派生类中只产生一个子对象，必须对这个基类声明为虚继承，使这个基类成为虚基类。
- 虚继承声明使用关键字 `virtual`

例如：

```
class B { public: int b; };
class B1: virtual public B { private: int b1; };
class B2: virtual public B { private: int b2; };
class C: public B1, public B2
    { private: float d; };
```



有：

```
C cc;
```

```
cc.b // ok
```

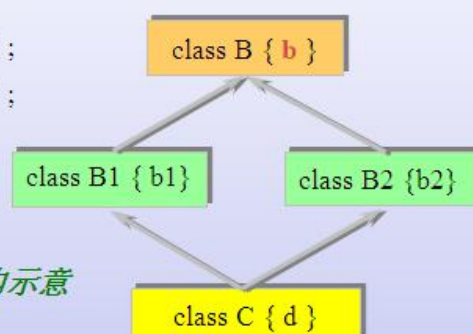
由于类 C 的对象中只有一个 B 类子对象，名字 b 被约束到该子对象上，所以，当以不同路径使用名字 b 访问 B 类的子对象时，所访问的都是那个唯一的基类子对象。即

`cc.B1::b` 和 `cc.B2::b`

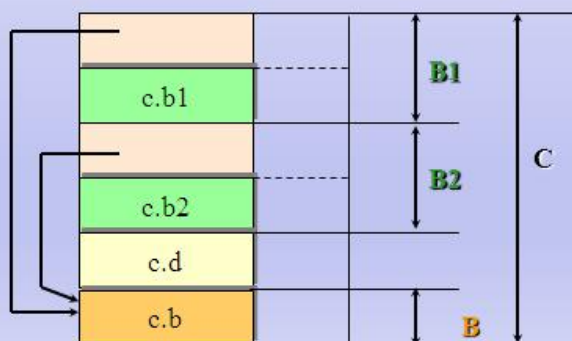
引用是同一个基类 B 的子对象

例如：

```
class B { public: int b; };
class B1: virtual public B { private: int b1; };
class B2: virtual public B { private: int b2; };
class C: public B1, public B2
    { private: float d; };
```



带有虚基类的派生类 C 的对象的存储结构示意



实验：注意增加 `virtual` 关键字后，构造函数调用的次数。

3.5 继承总结

- 继承是面向对象程序设计实现软件重用的重要方法。程序员可以在已有基类的基础上定义新的派生类。
- 单继承的派生类只有一个基类。多继承的派生类有多个基类。
- 派生类对基类成员的访问由继承方式和成员性质决定。
- 创建派生类对象时，先调用基类构造函数初始化派生类中的基类成员。调用析构函数的次序和调用构造函数的次序相反。
- C++ 提供虚继承机制，防止类继承关系中成员访问的二义性。
- 多继承提供了软件重用的强大功能，也增加了程序的复杂性。

4、多态

问题引出（赋值兼容性原则遇上函数重写）

面向对象新需求

C++ 提供的多态解决方案

多态案例

多态工程意义

面向对象三大概念、三种境界（封装、继承、多态）

多态成立条件

总结条件、看代码的时候要看出多态

4.1 多态

4.1.1 问题引出

如果子类定义了与父类中原型相同的函数会发生什么？

函数重写

在子类中定义与父类中原型相同的函数

函数重写只发生在父类与子类之间

```
class Parent
{
public:
    void print()
    {
        cout<<"Parent:print() do..."<<endl;
    }
}
```

```
};

class Child : public Parent
{
public:
    void print()
    {
        cout<<"Child:print() do..."<<endl;
    }
};

int main01()
{
    run00();

    /*
    Child child;
    Parent *p = NULL;
    p = &child;
    child.print();
    child.Parent::print();
    */

    system("pause");
    return 0;
}
```

父类中被重写的函数依然会继承给子类
默认情况下子类中重写的函数将隐藏父类中的函数
通过作用域分辨符::可以访问到父类中被隐藏的函数

```
/*
C/C++是静态编译型语言
在编译时，编译器自动根据指针的类型判断指向的是一个什么样的对象
*/
/*
1、在编译此函数的时，编译器不可能知道指针 p 究竟指向了什么。
2、编译器没有理由报错。
3、于是，编译器认为最安全的做法是编译到父类的 print 函数，因为父类和子类肯定都有相同的 print 函数。
*/

//面向对象新需求
//如果我传一个父类对象，执行父类的 print 函数
```



```
//如果我传一个子类对象，执行子类的 printf 函数
```

```
//现象产生的原因
```

```
//赋值兼容性原则遇上函数重写 出现的一个现象
```

```
//1 没有理由报错
```

```
//2 对被调用函数来讲，在编译器编译期间，我就确定了，这个函数的参数是 p，是 Parent 类型的。。。
```

```
//3 静态链编
```

```
//工程中如何判断是不是多态存在？
```

```
/*
```

```
在同一个类里面能实现函数重载
```

```
    继承的情况下，发生重写
```

```
    重载不一定；
```

```
重写的定义
```

```
    静态联编 重载是
```

```
    动态联编
```

```
*/
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Parent
```

```
{
```

```
public:
```

```
    void print()
```

```
    {
```

```
        cout<<"Parent:print() do..."<<endl;
```

```
    }
```

```
};
```

```
class Child : public Parent
```

```
{
```

```
public:
```

```
    void print()
```

```
    {
```

```
        cout<<"Child:print() do..."<<endl;
```

```
    }
```

```
};
```

```
/*
```

```
1、在编译此函数的时，编译器不可能知道指针 p 究竟指向了什么。
```

2、编译器没有理由报错。

3、于是，编译器认为最安全的做法是编译到父类的 `print` 函数，因为父类和子类肯定都有相同的 `print` 函数。

```
*/
```

```
void howToPrint(Parent* p)
```

```
{
```

```
    p->print();
```

```
}
```

```
void run00()
```

```
{
```

```
    Child child;
```

```
    Parent* pp = &child;
```

```
    Parent& rp = child;
```

```
    //child.print();
```

```
    //通过指针
```

```
    //pp->print();
```

```
    //通过引用
```

```
    //rp.print();
```

```
    howToPrint(&child);
```

```
}
```

```
int main01()
```

```
{
```

```
    run00();
```

```
    /*
```

```
    Child child;
```

```
    Parent *p = NULL;
```

```
    p = &child;
```

```
    child.print();
```

```
    child.Parent::print();
```

```
    */
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

4.1.2 面向对象新需求

编译器的做法不是我们期望的

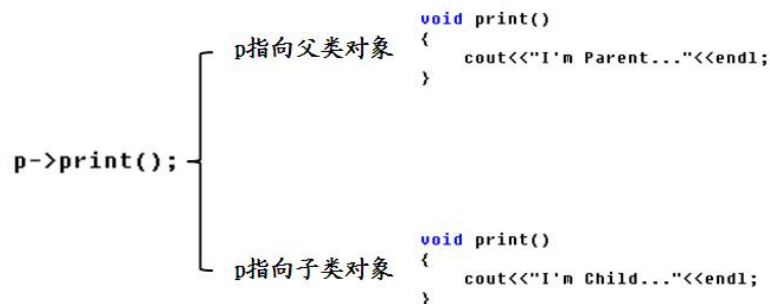
根据实际的对象类型来判断重写函数的调用

如果父类指针指向的是父类对象则调用父类中定义的函数

如果父类指针指向的是子类对象则调用子类中定义的重写函数

面向对象中的多态

根据实际的对象类型决定函数调用语句的具体调用目标



多态：同样的调用语句有多种不同的表现形态

4.1.3 解决方案

- C++中通过 `virtual` 关键字对多态进行支持
- 使用 `virtual` 声明的函数被重写后即可展现多态特性

4.1.4 多态实例

```
#include "iostream"
using namespace std;

class HeroFighter
{
public:
public:
    virtual int ackPower()
    {
        return 10;
    }
};

class AdvHeroFighter : public HeroFighter
```

```
{
public:
    virtual int ackPower()
    {
        return HeroFighter::ackPower()*2;
    }
};

class enemyFighter
{
public:
    int destoryPower()
    {
        return 15;
    }
};

//如果把这个结构放在动态库里面
//写了一个框架，可以调用
//我的第 3 代战机代码出现的时间晚于框架出现的时间。。。。
//框架 有使用后来人 写的代码的能力。。。
//面向对象 3 大概念
/*
封装
    突破了 C 语言函数的概念。。

继承
    代码复用 。。。。 我复用原来写好的代码。。。

多态
    多态可以使用未来。。。。。 80 年代写了一个框架。。。。。 90 人写的代码
    多态是我们软件行业追寻的一个目标。。。
*/
//
*/
//
void objPK(HeroFighter *hf, enemyFighter *enemyF)
{
    if (hf->ackPower() > enemyF->destoryPower())
    {
        printf("英雄打败敌人。。。胜利\n");
    }
    else
    {
        printf("英雄。。。牺牲\n");
    }
}
```

```
    }  
}  
  
void main()  
{  
    HeroFighter hf;  
    enemyFighter ef;  
  
    objPK(&hf, &ef);  
  
    AdvHeroFighter advhf;  
  
    objPK(&advhf, &ef);  
    system("pause");  
}
```

4.1.5 多态工程意义

//面向对象 3 大概念

/*

封装

突破了 C 语言函数的概念。。

继承

代码复用。。。。我复用原来写好的代码。。。

多态

多态可以使用未来。。。。80 年代写了一个框架。。。。90 人写的代码

多态是我们软件行业追寻的一个目标。。

//写了一个框架，可以调用后来人，写的代码的能力

////

*/

4.1.6 多态成立的条件

//间接赋值成立的 3 个条件

//1 定义两个变量。。

//2 建立关联。。。。

//3 *p

//多态成立的三个条件

//1 要有继承

```
//2 要有函数重写。。。C 虚函数
//3 要有父类指针（父类引用）指向子类对象
//多态是设计模式的基础，多态是框架的基础
```

4.1.7 多态的理论基础

01 静态联编和动态联编

- 1、联编是指一个程序模块、代码之间互相关联的过程。
- 2、静态联编（static binding），是程序的匹配、连接在编译阶段实现，也称为早期匹配。
重载函数使用静态联编。
- 3、动态联编是指程序联编推迟到运行时进行，所以又称为晚期联编（迟绑定）。
switch 语句和 if 语句是动态联编的例子。
- 4、理论联系实际

- 1、C++与 C 相同，是静态编译型语言
- 2、在编译时，编译器自动根据指针的类型判断指向的是一个什么样的对象；所以编译器认为父类指针指向的是父类对象。
- 3、由于程序没有运行，所以不可能知道父类指针指向的具体是父类对象还是子类对象
从程序安全的角度，编译器假设父类指针只指向父类对象，因此编译的结果为调用父类的成员函数。这种特性就是静态联编。

4.2 多态相关面试题

面试题 1：请谈谈你对多态的理解

多态的实现效果

多态：同样的调用语句有多种不同的表现形态；

多态实现的三个条件

有继承、有 virtual 重写、有父类指针（引用）指向子类对象。

多态的 C++ 实现

virtual 关键字，告诉编译器这个函数要支持多态；不是根据指针类型判断如何调用；而是要根据指针所指向的实际对象类型来判断如何调用

多态的理论基础

动态联编 PK 静态联编。根据实际的对象类型来判断重写函数的调用。

多态的重要意义

设计模式的基础 是框架的基石。

实现多态的理论基础

函数指针做函数参数

C 函数指针是 C++ 至高无上的荣耀。C 函数指针一般有两种用法（正、反）。

多态原理探究

与面试官展开讨论

面试题 2：谈谈 C++ 编译器是如何实现多态

C++ 编译器多态实现原理

面试题 3：谈谈你对重写，重载理解

函数重载

必须在同一个类中进行

子类无法重载父类的函数，父类同名函数将被名称覆盖

重载是在编译期间根据参数类型和个数决定函数调用

函数重写

必须发生于父类与子类之间

并且父类与子类中的函数必须有完全相同的原型

使用 `virtual` 声明之后能够产生多态(如果不使用 `virtual`，那叫重定义)

多态是在运行期间根据具体对象的类型决定函数调用

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Parent01
{
public:
    Parent01()
    {
        cout<<"Parent01:printf()..do"<<endl;
    }
public:
    virtual void func()
    {
        cout<<"Parent01:void func()"<<endl;
    }

    virtual void func(int i)
    {
        cout<<"Parent:void func(int i)"<<endl;
    }

    virtual void func(int i, int j)
    {
        cout<<"Parent:void func(int i, int j)"<<endl;
    }
}
```

```
};

class Child01 : public Parent01
{
public:

    //此处 2 个参数，和子类 func 函数是什么关系
    void func(int i, int j)
    {
        cout<<"Child:void func(int i, int j)"<<" "<<i + j<<endl;
    }

    //此处 3 个参数的，和子类 func 函数是什么关系
    void func(int i, int j, int k)
    {
        cout<<"Child:void func(int i, int j, int k)"<<" "<<i + j + k<<endl;
    }
};

void run01(Parent01* p)
{
    p->func(1, 2);
}

int main()
{
    Parent01 p;

    p.func();
    p.func(1);
    p.func(1, 2);

    Child01 c;
    //c.func(); //问题 1
    c.Parent01::func();
    c.func(1, 2);

    run01(&p);
    run01(&c);

    system("pause");
    return 0;
}
```



```
//问题 1: child 对象继承父类对象的 func, 请问这句话能运行吗? why
//c.func(); //因为名称覆盖, C++编译器不会去父类中寻找 0 个参数的 func 函数, 只会子类
中找 func 函数。

//1 子类里面的 func 无法重载父类里面的 func
//2 当父类和子类有相同的函数名、变量名出现, 发生名称覆盖 (子类的函数名, 覆盖了父
类的函数名。)
//3//c.Parent::func();
//问题 2 子类的两个 func 和父类里的三个 func 函数是什么关系?
```

面试题 4: 是否可类的每个成员函数都声明为虚函数, 为什么。

c++编译器多态实现原理

面试题 5: 构造函数中调用虚函数能实现多态吗? 为什么?

c++编译器多态实现原理

面试题 6: 虚函数表指针 (VPTR) 被编译器初始化的过程, 你是如何理解的?

c++编译器多态实现原理

面试题 7: 父类的构造函数中调用虚函数, 能发生多态吗?

c++编译器多态实现原理

面试题 8: 为什么要定义虚析构函数?

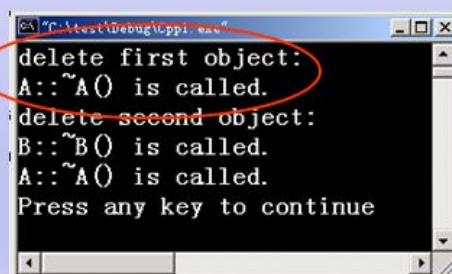
在什么情况下应当声明虚函数

- 构造函数不能是虚函数。建立一个派生类对象时, 必须从类层次的根开始, 沿着继承路径逐个调用基类的构造函数
- 析构函数可以是虚的。虚析构函数用于指引 delete 运算符正确析构动态对象

例8-4 普通析构函数在删除动态派生类对象的调用情况

```
#include<iostream.h>
class A
{ public:
    ~A(){ cout << "A::~~A() is called.\n"; }
};
class B : public A
{ public:
    ~B(){ cout << "B::~~B() is called.\n"; }
};
void main()
{ A *Ap = new B;
  B *Bp2 = new B;
  cout << "delete first object:\n";
  delete Ap;
  cout << "delete second object:\n";
  delete Bp2;
}
```

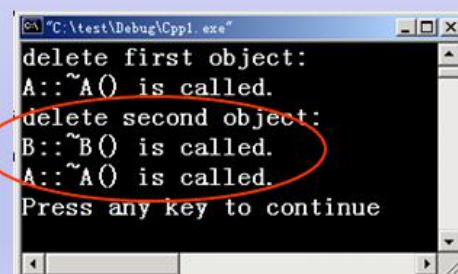
析构由基类指针建立的派生类对象
没有调用派生类析构函数



例8-4 普通析构函数在删除动态派生类对象的调用情况

```
#include<iostream.h>
class A
{ public:
    ~A(){ cout << "A::~~A() is called.\n"; }
};
class B : public A
{ public:
    ~B(){ cout << "B::~~B() is called.\n"; }
};
void main()
{ A *Ap = new B;
  B *Bp2 = new B;
  cout << "delete first object:\n";
  delete Ap;
  cout << "delete second object:\n";
  delete Bp2;
}
```

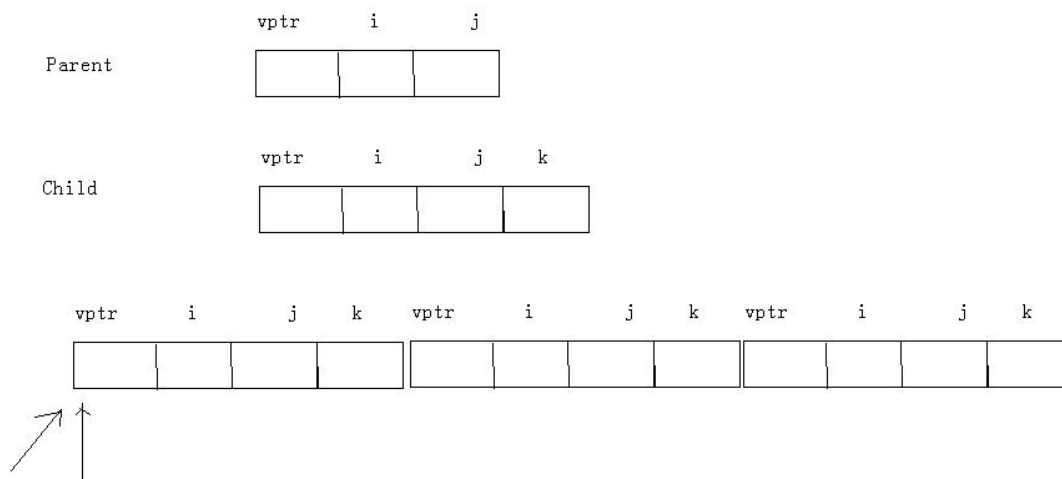
析构由派生类指针建立的派生类对象
正确调用派生类析构函数



其他

父类指针和子类指针的步长

- 1) 铁律 1: 指针也只一种数据类型, C++ 类对象的指针 $p++/--$, 仍然可用。
- 2) 指针运算是按照指针所指的类型进行的。
 $p++ \ll p = p + 1 // p = (\text{unsigned int})\text{basep} + \text{sizeof}(*p)$ 步长。
- 3) 结论: 父类 $p++$ 与子类 $p++$ 步长不同; 不要混搭, 不要用父类指针++方式操作数组。



4.3 多态原理探究

理论知识:

- 当类中声明虚函数时, 编译器会在类中生成一个虚函数表
- 虚函数表是一个存储类成员函数指针的数据结构
- 虚函数表是由编译器自动生成与维护的
- `virtual` 成员函数会被编译器放入虚函数表中
- 当存在虚函数时, 每个对象中都有一个指向虚函数表的指针 (C++ 编译器给父类对象、子类对象提前布局 `vptr` 指针; 当进行 `howToPrint(Parent *base)` 函数是, C++ 编译器不需要区分子类对象或者父类对象, 只需要再 `base` 指针中, 找 `vptr` 指针即可。)
- `VPTR` 一般作为类对象的第一个成员

4.3.1 多态的实现原理

C++ 中多态的实现原理

当类中声明虚函数时, 编译器会在类中生成一个虚函数表

虚函数表是一个存储类成员函数指针的数据结构

虚函数表是由编译器自动生成与维护的

`virtual` 成员函数会被编译器放入虚函数表中

存在虚函数时, 每个对象中都有一个指向虚函数表的指针(`vptr` 指针)

```

class Parent
{
public:
    virtual void func()
    {
        cout<<"Parent::func()"<<endl;
    }
    virtual void func(int i)
    {
        cout<<"Parent::func(int i)"<<endl;
    }
};

```



VTABLE

```

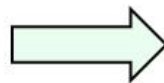
void Parent::func()
void Parent::func(int i)

```

```

class Child : public Parent
{
public:
    virtual void func()
    {
        cout<<"Child::func()"<<endl;
    }
    virtual void func(int i)
    {
        cout<<"Child::func(int i)"<<endl;
    }
};

```

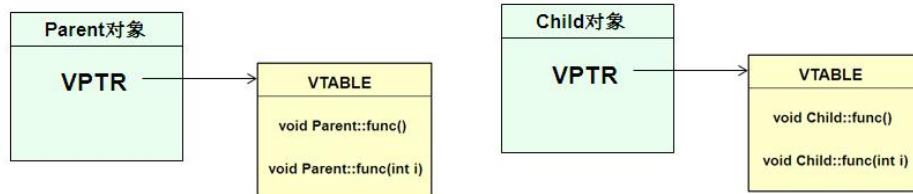


VTABLE

```

void Child::func()
void Child::func(int i)

```



```

void run(Parent* p)
{
    p->func();
}

```

编译器确定func是否为虚函数

1) func不是虚函数，编译器可直接确定被调用的成员函数，（静态链编，根据Parent类型来确定）。

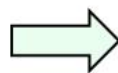
2) func是虚函数，编译器根据对象p的Vptr指针，所指的虚函数表中查找func（）函数，并调用。

注意：查找和调用在运行时完成，（实现所谓的动态链编）。

```

void run(Parent* p)
{
    p->func();
}

```



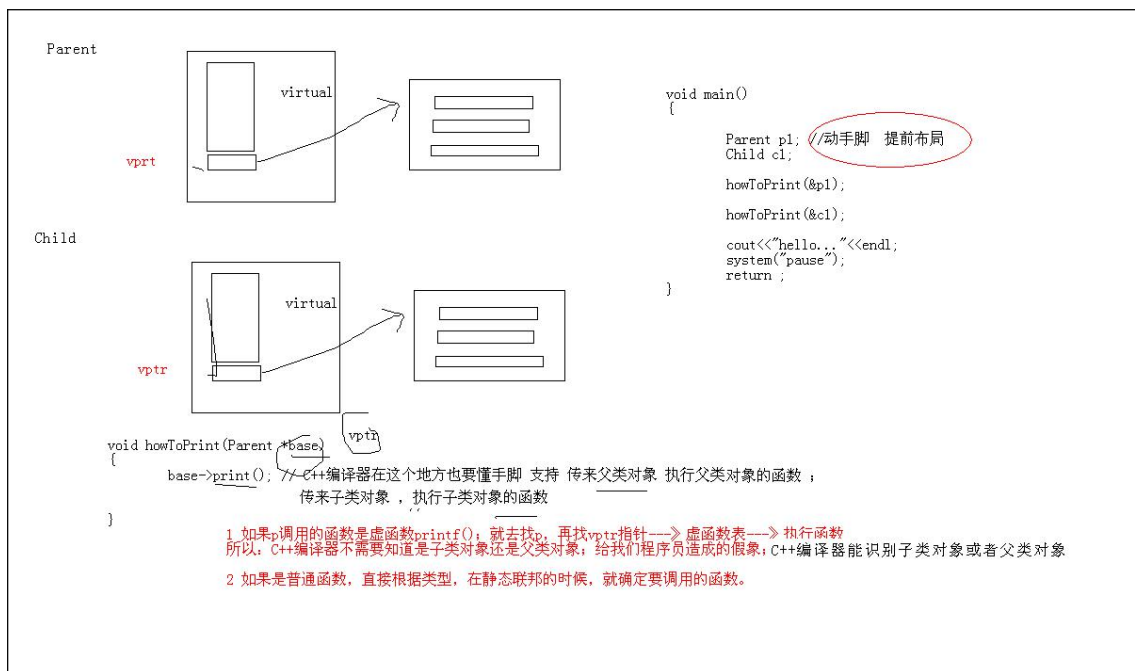
说明 1:

通过虚函数表指针 VPTR 调用重写函数是在程序运行时进行的，因此需要通过寻址操作才能确定真正应该调用的函数。而普通成员函数是在编译时就确定了调用的函数。在效率上，虚函数的效率要低很多。

说明 2:

出于效率考虑，没有必要将所有成员函数都声明为虚函数

说明 3：C++编译器，执行 HowToPrint 函数，不需要区分是子类对象还是父类对象



4.3.2 如何证明 vptr 指针的存在

```

#include <iostream>
using namespace std;

class A
{
public:
    void printf()
    {
        cout<<"aaa"<<endl;
    }
protected:
private:
    int a;
};

class B
{
public:
    virtual void printf()
    {
        cout<<"aaa"<<endl;
    }
protected:
private:

```

```
    int a;
};

void main()
{
    //加上 virtual 关键字 c++编译器会增加一个指向虚函数表的指针。。。
    printf("sizeof(a):%d, sizeof(b):%d \n", sizeof(A), sizeof(B));
    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

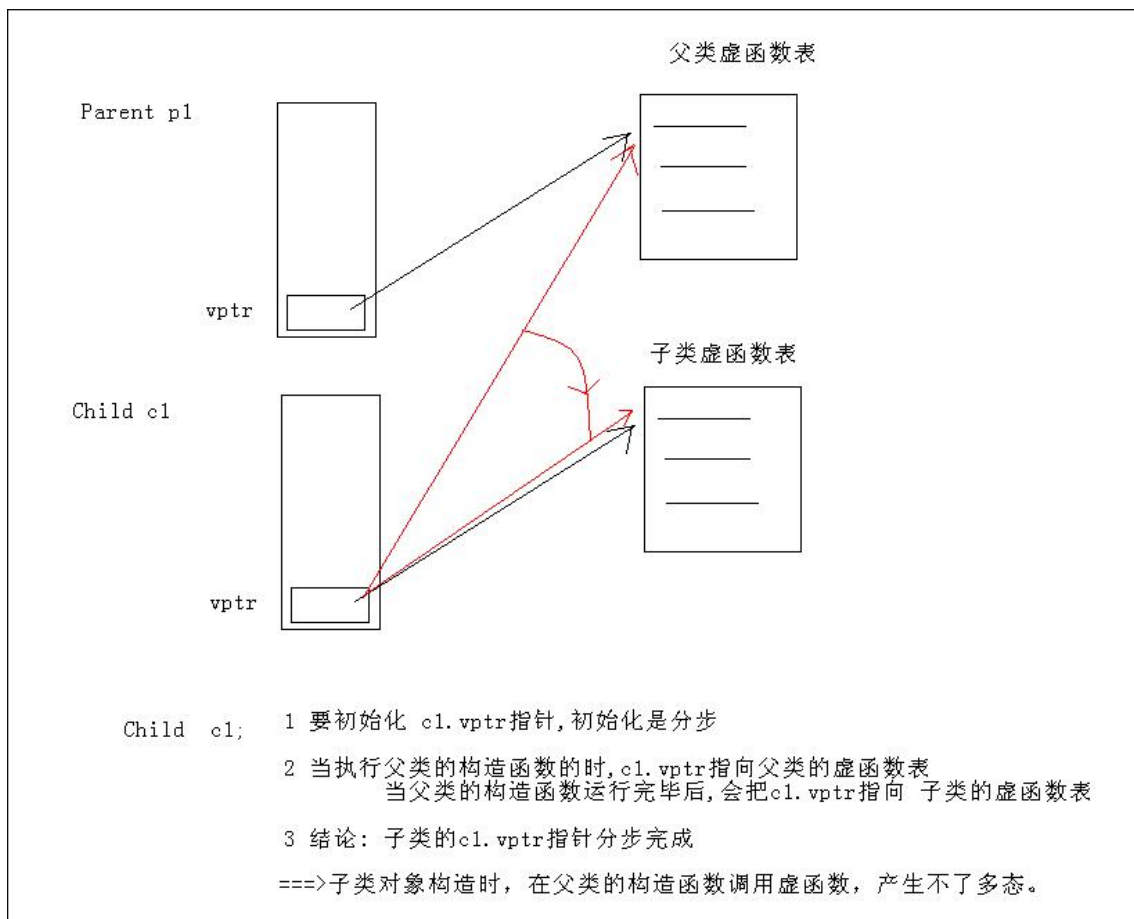
4.3.3 构造函数中能调用虚函数，实现多态吗

1) 对象中的 VPTR 指针什么时候被初始化?

对象在创建的时候,由编译器对 VPTR 指针进行初始化
只有当对象的构造完全结束后 VPTR 的指向才最终确定
父类对象的 VPTR 指向父类虚函数表
子类对象的 VPTR 指向子类虚函数表

2) 分析过程

画图分析



5、纯虚函数和抽象类

5.1 基本概念

纯虚函数和抽象类

- 纯虚函数是一个在基类中说明的虚函数，在基类中没有定义，要求任何派生类都定义自己的版本
- 纯虚函数为各派生类提供一个公共界面（接口的封装和设计、软件的模块功能划分）
- 纯虚函数说明形式：

virtual 类型 函数名(参数表) = 0 ;

- 一个具有纯虚函数的基类称为抽象类。

例如：

```
class point { /*.....*/ };  
class shape ;           // 抽象类  
{ point center ;  
    .....  
public :  
    point where () { return center ; }  
    void move ( point p ) { enter = p ; draw () ; }  
    virtual void rotate ( int ) = 0 ;           // 纯虚函数  
    virtual void draw () = 0 ;                 // 纯虚函数  
};  
    .....  
  
shape x ;           // error, 抽象类不能建立对象  
shape *p ;          // ok, 可以声明抽象类的指针  
shape f () ;         // error, 抽象类不能作为返回类型  
void g ( shape ) ;   // error, 抽象类不能作为参数类型  
shape & h ( shape & ) ; // ok, 可以声明抽象类的引用
```


例如:

```
class point { /*.....*/ };
```

```
class shape;
```

```
{ point center;
```

```
.....
```

```
public:
```

```
point where() { return ce
```

```
void move (point p) { en
```

```
virtual void rotate (int)
```

```
virtual void draw () = 0
```

```
};
```

```
.....
```

```
class ab_circle: public shape
```

```
{ int radius;
```

```
public: void rotate (int)
```

```
};
```

// 抽象类

要使 ab_circle 成为非抽象类，
必须作以下说明：

```
class ab_circle: public shape
```

```
{ int radius;
```

```
public:
```

```
void rotate (int);
```

```
void draw ();
```

```
};
```

并提供 ab_circle::draw ()

和 ab_circle::rotate (int)

的定义

5.2 抽象类案例

```
class figure
```

```
{ protected: double x,y;
```

```
public: void set_dim(double i, double j=0) { x = i; y = j; }
```

```
virtual void show_area() = 0;
```

// 纯虚函数

```
};
```

```
class triangle : public figure
```

```
{ public:
```

```
void show_area()
```

```
{ cout<<"Triangle with high "<<x<<" and base "<<y<<" has an area of "
```

```
}; <<x*0.5*y<<"\n"; }
```

```
class square : public figure
```

```
{ public:
```

```
void show_area()
```

```
{ cout<<"Square with dimension "<<x<<"*"<<y<<" has an area of "
```

```
}; <<x*y<<"\n"; }
```

```
class circle : public figure
```

```
{ public:
```

```
void show_area()
```

```
{ cout<<"Circle with radius "<<x;
```

```
cout<<" has an area of "<<3.14*x*x<<"\n";
```

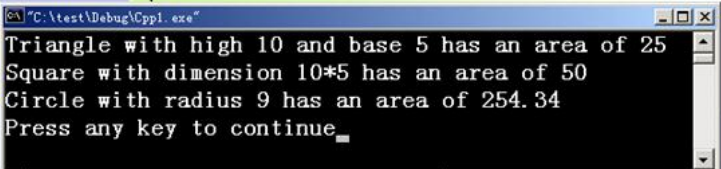
```
}
```

```
};
```

简单图形类

```
//figure.h
class figure
{ protected: double x,y;
  public: void set_dim(double i, double j);
          virtual void show_area();
};
class triangle : public figure
{ public:
    void show_area()
    { cout<<"Triangle with high " << x * 0.5 * y << "\n"; }
};
class square : public figure
{ public:
    void show_area()
    { cout<<"Square with dimension " << x * y << "\n"; }
};
class circle : public figure
{ public:
    void show_area()
    { cout<<"Circle with radius " << x << " has an area of " << y << "\n"; }
};

#include<iostream.h>
#include"figure.h"
void main()
{ figure *p;           // 声明抽象类指针
  triangle t;
  square s;
  circle c;
  p=&t;
  p->set_dim(10.0,5.0);  // triangle::set_dim()
  p->show_area();
  p=&s;
  p->set_dim(10.0,5.0);  // square::set_dim()
  p->show_area();
  p=&c;
  p->set_dim(9.0);       // circle::set_dim()
  p->show_area();
}
```



```
C:\test\Debug\Cppl.exe
Triangle with high 10 and base 5 has an area of 25
Square with dimension 10*5 has an area of 50
Circle with radius 9 has an area of 254.34
Press any key to continue
```

5.3 抽象类在多继承中的应用

C++中没有 Java 中的接口概念，抽象类可以模拟 Java 中的接口类。（接口和协议）

5.3.1 有关多继承的说明

工程上的多继承

被实际开发经验抛弃的多继承

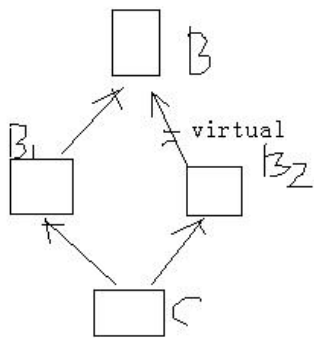
工程开发中真正意义上的多继承是几乎不被使用的

多重继承带来的代码复杂性远多于其带来的便利

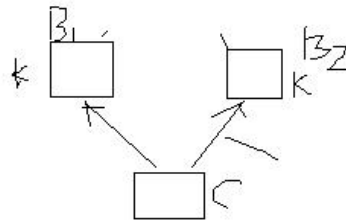
多重继承对代码维护性上的影响是灾难性的

在设计方法上，任何多继承都可以用单继承代替

多继承中的二义性和多继承不能解决的问题



多继承 第一种能解决



第二种暂时无法解决

5.3.2 多继承的应用场景

C++中是否有 Java 中的接口概念？
<p>绝大多数面向对象语言都不支持多继承 绝大多数面向对象语言都支持接口的概念 C++中没有接口的概念 C++中可以使用纯虚函数实现接口 接口类中只有函数原型定义，没有任何数据的定义。</p> <pre> class Interface { public: virtual void func1() = 0; virtual void func2(int i) = 0; virtual void func3(int i) = 0; }; </pre>
<p>实际工程经验证明 多重继承接口不会带来二义性和复杂性等问题 多重继承可以通过精心设计用单继承和接口来代替 接口类只是一个功能说明，而不是功能实现。 子类需要根据功能说明定义功能实现。</p>
<pre> #include "iostream" using namespace std; /* C++中没有接口的概念 C++中可以使用纯虚函数实现接口 接口类中只有函数原型定义，没有任何数据的定义。 */ class Interface1 </pre>

```
{
public:
    virtual void print() = 0;
    virtual int add(int a, int b) = 0;
};

class Interface2
{
public:
    virtual void print() = 0;
    virtual int add(int a, int b) = 0;
    virtual int minus(int a, int b) = 0;
};

class parent
{
public:
    int a;
};

class Child : public parent, public Interface1, public Interface2
{
public:
    void print()
    {
        cout<<"Child::print"<<endl;
    }

    int add(int a, int b)
    {
        return a + b;
    }

    int minus(int a, int b)
    {
        return a - b;
    }
};

int main()
{
    Child c;

    c.print();
}
```

```
cout<<c.add(3, 5)<<endl;
cout<<c.minus(4, 6)<<endl;

Interface1* i1 = &c;
Interface2* i2 = &c;

cout<<i1->add(7, 8)<<endl;
cout<<i2->add(7, 8)<<endl;
system("pause");
}
```

5.4 抽象类知识点强化

/*

编写一个 C++ 程序，计算程序员 (programmer) 工资

1 要求能计算出初级程序员 (junior_programmer) 中级程序员 (mid_programmer) 高级程序员 (adv_programmer) 的工资

2 要求利用抽象类统一界面，方便程序的扩展，比如：新增，计算 架构师 (architect) 的工资

*/

5.5 面向抽象类编程思想强化

理论知识

- 虚函数和多态性使成员函数根据调用对象的类型产生不同的动作
- 多态性特别适合于实现分层结构的软件系统，便于对问题抽象时 定义共性，实现时定义区别
- 面向抽象类编程（面向接口编程）是项目开发中重要技能之一。

5.4.1 案例：socket 库 c++ 模型设计和实现

企业信息系统框架集成第三方产品

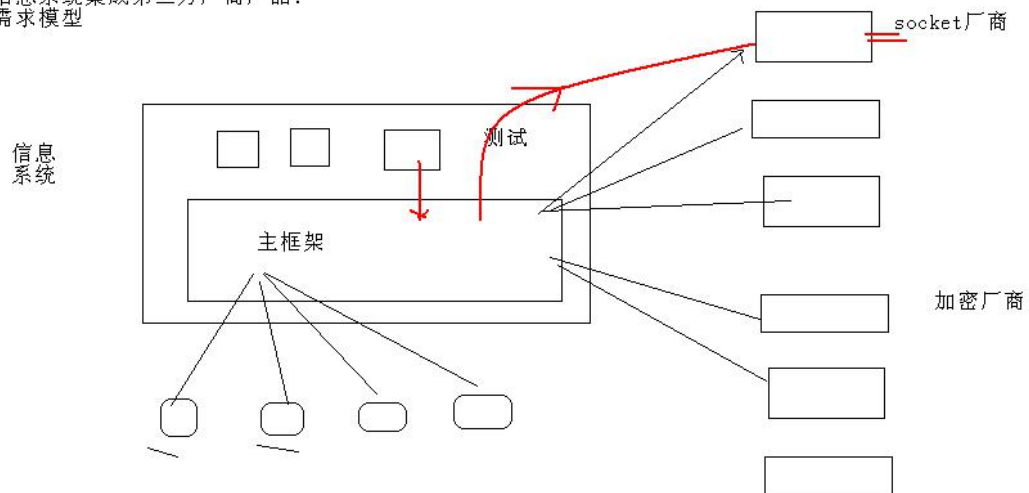
案例背景：一般的企业信息系统都有成熟的框架。软件框架一般不发生变化，能自由的集成第三方厂商的产品。

案例需求：请你在企业信息系统框架中集成第三方厂商的 Socket 通信产品和第三方厂商加密产品。

第三方厂商的 Socket 通信产品：完成两点之间的通信；

第三方厂商加密产品：完成数据发送时加密；数据解密时解密。

信息系统集成第三方厂商产品：
需求模型



- 案例要求：**
- 1) 能支持多个厂商的 Socket 通信产品入围
 - 2) 能支持多个第三方厂商加密产品的入围
 - 3) 企业信息系统框架不轻易发生框架

需求实现

- 思考 1：企业信息系统框架、第三方产品如何分层
- 思考 2：企业信息系统框架，如何自由集成第三方产品
(软件设计：模块要求松、接口要求紧)
- 思考 3：软件分成以后，开发企业信息系统框架的程序员，应该做什么？
第三方产品入围应该做什么？

编码实现

- 分析有多少个类 CSocketProtocol CSckFactoryImp1 CSckFactoryImp2
CEncDesProtocol HwEncdes ciscoEncdes
- 1、定义 CSocketProtocol 抽象类
 - 2、编写框架函数
 - 3、编写框架测试函数
 - 4、厂商1(CSckFactoryImp1)实现CSocketProtocol、厂商2(CSckFactoryImp1)实现CSocketProtoco
 - 5、抽象加密接口 (CEncDesProtocol)、加密厂商1(CHwImp)、加密厂商2(CCiscoImp)，集成实现业务模型
 - 6、框架 (c语言函数方式，框架函数；c++类方式，框架类)

几个重要的面向对象思想

- 继承-组合 (强弱)
- 注入
- 控制反转 IOC
- MVC
- 面向对象思想扩展aop思想
- aop思想是对继承编程思想的有力的补充

5.4.2 案例：计算员工工资

5.4.3 案例：计算几何体的表面积和体积

5.6 C 面向接口编程和 C 多态

友情提示：今天课程内容，更加贴近实战，并且语法和软件思想都较难，请学员紧跟思路。课后加强复习！

结论：只要你动手，又很容易！

5.6.1 函数类型语法基础

函数三要素： 名称、参数、返回值 C 语言中的函数有自己特定的类型
C 语言中通过 <code>typedef</code> 为函数类型重命名 <code>typedef type name(parameter list)</code> <code>typedef int f(int, int);</code> <code>typedef void p(int);</code>
函数指针
函数指针用于指向一个函数 函数名是函数体的入口地址 1) 可通过函数类型定义函数指针: <code>FuncType* pointer;</code> 2) 也可以直接定义: <code>type (*pointer)(parameter list);</code> <code>pointer</code> 为函数指针变量名 <code>type</code> 为指向函数的返回值类型 <code>parameter list</code> 为指向函数的参数类型列表
函数指针语法梳理 //函数类型 //函数指针类型 //函数指针变量 数组指针语法梳理 //数组类型语法 //数组指针类型 //数组指针变量

```
typedef int(FUNC)(int);

int test(int i)
{
    return i * i;
}

void f()
{
    printf("Call f()...\n");
}

int main()
{
    FUNC* pt = test;

    void(*pf)() = &f;

    pf();
    (*pf)();

    printf("Function pointer call: %d\n", pt(3));
}
```

5.6.2 函数指针做函数参数

1、指针做函数参数 pk 函数指针做函数参数

回忆指针做函数参数

一级指针做函数参数、二级。。。。、三级

2、函数指针做函数参数

当函数指针 做为函数的参数，传递给一个被调用函数，
被调用函数就可以通过这个指针调用外部的函数，这就形成了回调

3、练习

```
int add(int a, int b);
int libfun(int (*pDis)(int a, int b));

int main(void)
{
    int (*pfun)(int a, int b);
    pfun = add;
    libfun(pfun);
}
```



```
}

int add(int a, int b)
{
    return a + b;
}

int libfun(int (*pDis)(int a, int b))
{
    int a, b;
    a = 1;
    b = 2;
    add(1,3) //直接调用 add 函数
    printf("%d", pDis(a, b)); //通过函数指针做函数参数,间接调用 add 函数
    //思考 这样写 pDis(a, b)有什么好处?
}

//剖析思路
//1 函数的调用 和 函数的实现 有效的分离
//2 C++的多态,可扩展
```

现在这几个函数是在同一个文件当中

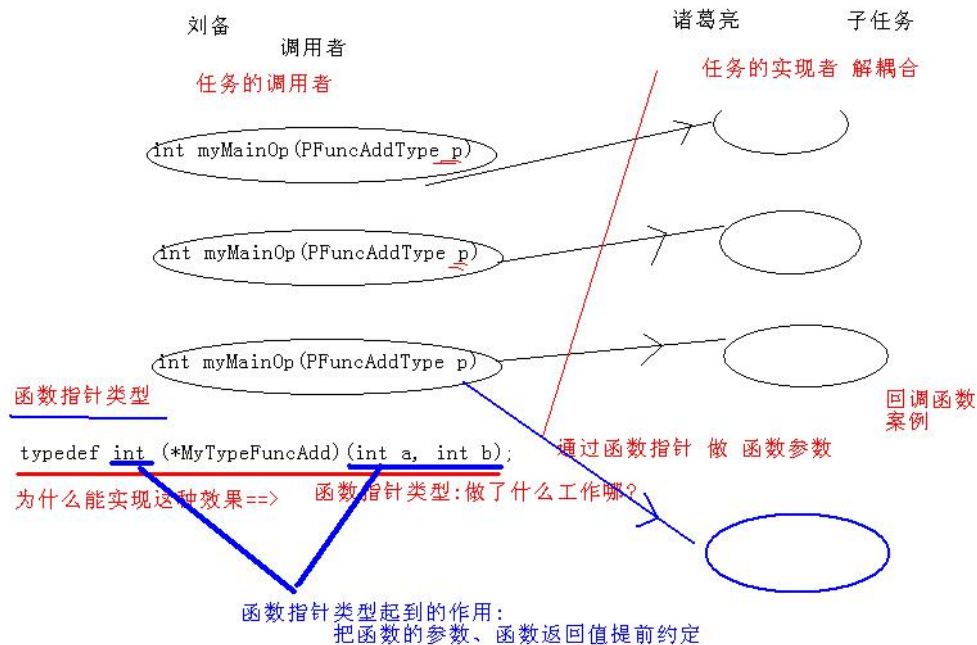
假如

```
int libfun(int (*pDis)(int a, int b))
```

是一个库中的函数，就只有使用回调了，通过函数指针参数将外部函数地址传入来实现调用

函数 `add` 的代码作了修改，也不必改动库的代码，就可以正常实现调用
便于程序的维护和升级

回调函数思想：



结论：回调函数的本质：提前做了一个协议的约定（把函数的参数、函数返回值提前约定）
请思考：C 编译器通过那个具体的语法，实现解耦合的？

C++编译器通过多态的机制(提前布局 `vptr` 指针和虚函数表,找虚函数入口地址来实现)

5.6.3 函数指针正向调用

1、函数指针做函数参数，调用方式

被调用函数和主调函数在同一文件中（用来教学，没有任何意义）

2、函数指针做函数参数

被调用函数和主调函数不在同一个文件中、模块中。

难点：理解被调用函数是什么机制被调用起来的。框架

框架提前设置了被调用函数的入口（框架提供了第三方模块入口地址的集成功能）

框架具备调用第三方模块入口函数

3、练习

```
typedef int (*EncDataFunc)(unsigned char *inData,int inDataLen,unsigned char *outData,int *outDataLen,void *Ref, int RefLen);
```

```
int MyEncDataFunc(unsigned char *inData,int inDataLen,unsigned char *outData,int *outDataLen,void *Ref, int RefLen)
```

```
{
```

```
    int rv = 0;
```

```
    char *p = "222222222222";
```

```
    strcpy(outData, p);
```

```
    *outDataLen = strlen(p);
```

```
        return rv;
    }

int Send_Data(EncDataFunc encDataFunc, unsigned char *inData, int inDataLen, unsigned char
*outData, int *outDatalen)
{
    int rv = 0;
    if (encDataFunc != NULL)
    {
        rv = encDataFunc(inData, inDataLen, outData, outDatalen,  NULL, 0);
        if (rv != 0)
        {
            printf("func encDataFunc() err.\n");
            return rv;
        }
    }
    return rv;
}

int main()
{
    int rv = 0;

    EncDataFunc encDataFunc = NULL;
    encDataFunc = MyEncDataFunc;

    // 第一个调用
    {
        unsigned char inData[2048];
        int inDataLen;
        unsigned char outData[2048];
        int outDatalen;
        strcpy(inData, "1111");
        inDataLen = strlen(inData);
        rv = encDataFunc(inData,inDataLen, outData, &outDatalen, NULL, 0);
        if (rv != 0)
        {
            printf("edf err ..... \n");
        }
        else
        {
            printf("edf ok \n");
            printf("%s \n", outData);
        }
    }
}
```

```
}

{
    unsigned char inData[2048];
    int inDataLen;
    unsigned char outData[2048];
    int outDataLen;
    strcpy(inData, "3333");
    inDataLen = strlen(inData);
    rv = Send_Data(MyEncDataFunc, inData, inDataLen, outData, &outDataLen);
    if (rv != 0)
    {
        printf("func Send_Data err:%d", rv);
        return rv;
    }
    printf("%s \n", outData);
}

getchar();
}
```

5.6.4 函数指针反向调用

回调函数效果展示。

5.6.5.C 动态库升级成框架案例

C 语言版本 Socket 动态库升级成框架集成第三方产品

简称：**C 动态库升级成框架案例**

名字解释

动态库：抽象类一个套接口，单独封装成模块，供别人调用；无法扩展。

框架：能自由的扩展

案例背景：一般的企业信息系统都有成熟的框架，可以有 C 语言写，也可以由 C++ 语言。软件框架一般不发生变化，能自由的集成第三方厂商的产品。

案例需求：在 socket 通信库中，完成数据加密功能，有 n 个厂商的加密产品供你选择，如何实现动态库和第三个厂商产品的解耦合。

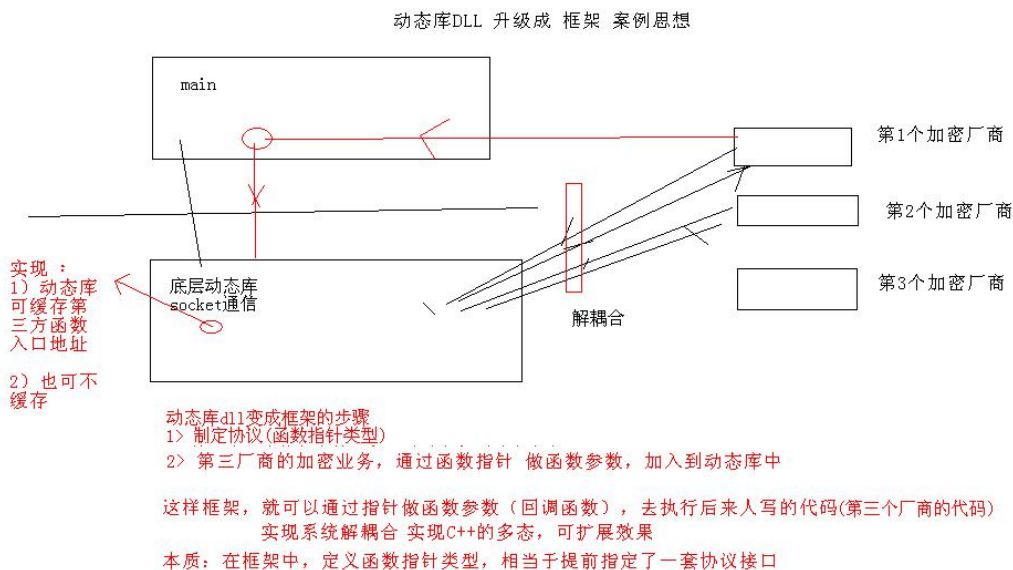
提醒：C++ 通过抽象类，也就是**面向抽象类编程**实现的（相当于 C++ 编译器通过多态机制，已经很好用了。提前布局 **vpitr** 指针、虚函数表；调用是迟绑定完成。），

C 语言中如何实现哪？

案例要求： 1) 能支持多个第三方厂商加密产品的入围

2) 企业信息系统框架不轻易发生框架

需求实现思路分析



传智扫地僧

思考 1: 企业信息系统框架、第三方产品如何分层

思考 2: 企业信息系统框架, 如何自由集成第三方产品

(软件设计: 模块要求松、接口要求紧)

思考 3: 软件分层确定后, 动态库应该做什么? 产品入围厂商应该做什么?

以后, 开发企业信息系统框架的程序员, 应该做什么?

第三方产品入围应该做什么?

编码实现

1、动态库中定义协议, 并完成任务的调用

```
typedef int (*EncData)(unsigned char *inData, int
inDataLen, unsigned char *outData, int *outDataLen, void *Ref, int
RefLen);
typedef int (*DecData)(unsigned char *inData, int inDataLen, unsigned
char *outData, int *outDataLen, void *Ref, int RefLen);
```

2、加密厂商完成协议函数的编写

3、对接调试。

4、动态库中可以缓存第三方函数的入口地址, 也可以不缓存, 两种实现方式。

案例总结

回调函数: 利用函数指针做函数参数, 实现的一种调用机制, 具体任务的实现者, 可以不知道什么时候被调用。

回调机制原理:

当具体事件发生时, 调用者通过函数指针调用具体函数

回调机制的将调用者和被调函数分开, 两者互不依赖

任务的实现 和 任务的调用 可以耦合 (提前进行接口的封装和设计)

5.6.6 附录：诸葛亮的锦囊妙计

刘备利用周瑜、曹仁厮杀之际，乘虚袭取了南郡、荆州、襄阳，以后又征服了长沙等四郡。周瑜想想十分气恨，正无处报复以夺还荆州。不久，刘备忽然丧偶，周瑜计上心来，对孙权说：“您的妹妹，美丽、刚强，我们以联姻抗曹名义向刘备招亲，把他骗来南徐幽禁，逼他们拿荆州来换。”孙权大喜，即派人到荆州说亲。

刘备认为这是骗局，想要拒绝，诸葛亮笑道：“送个好妻子上门何不答应？您只管去东吴，我叫赵云陪您去，自有安排，包您得了夫人又不失荆州。”

接着，诸葛亮暗暗关照赵云道：“我这里三个锦囊，内藏三条妙计。到南徐时打开第一个，到年底时打开第二个，危急无路时打开第三个。”

第一个锦囊

一到东吴就拜会乔国老

第二个锦囊

刘备被孙权设计留下就对他谎称曹操大军压境

第三个锦囊

被东吴军队追赶就求孙夫人解围