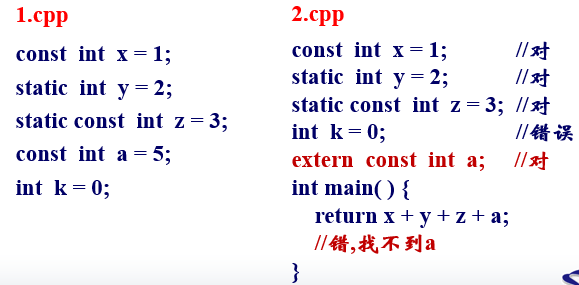
C++程序设计期末考试-补充PPT 2022.11.25

《深入理解 static, const, extern》

static、const 对象的作用域是当前模块 (当前.cpp文件)。不同的模块可以定义同名的static、const 变量。



变量定义

变量定义 (需要分配内存)，变量定义的2种形式：

* 没有使用extern限定，可以不初始化。例如, const int k = 0; int x, y; A a; //A定义了无参构造函数
* 使用extern限定, 这时必须初始化 (不初始化表明仅仅是声明, 这个变量实际上可能不存在)。

extern const int k = 0; extern int x = 1;

extern A a; //不是定义, 仅是声明

extern A a(…); //A定义了有参构造函数

extern A a = A(); //A定义了无参构造函数

变量定义有3种形式:

* 只定义不声明、先声明后定义、先定义后声明

extern 和 static 不能连用。

例：int x = 1;

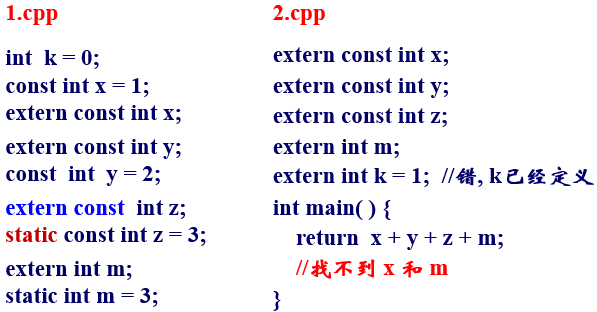
extern int x, y; static int y = 2; const int z = 1; extern const int z; int m, n;

extern int m;

* 在一个模块内部，一般只定义变量 (不声明)。既定义又声明没有意义。
* 只有对于 const [static] 变量，先声明后定义才有意义 (先定义后声明 ⬄ 只定义不声明)，表明将该变量的作用域修改为整个程序。

变量声明 extern

* 不给变量分配内存。用于声明可能有这个变量 (不管这个变量实际上是否存在)。
* 编译器遇到语句 extern变量 时，如果程序并没有访问这个变量，则不做任何处理。否则：先在当前模块查找该变量的定义；若没找到则在其他模块中查找该变量的定义，若找到且其作用域是整个程序则成功 (否则报错)。



《C++的变量、常量、程序空间》

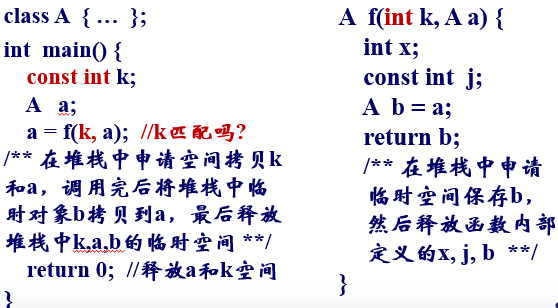
* 正在运行中的程序所占用的内存空间：程序代码指令空间、变量空间、常量空间、程序动态申请的空间 (new、malloc等)。这些空间被组织(管理)为段:
* 代码段 (用于存放程序的指令代码)、堆栈段 (非静态局部变量、及其他用途)、数据段 (用于全局变量和静态变量)、常量段 (用于保存常量数据, 例如常量字符串)
* 不管程序以何种方式退出，代码段空间和常量段空间都会被释放。

指针。指针 (不管多少重) 也是变量，编译器会给指针变量分配内存单元。

* 指针变量是用来保存地址的，对于win32 (x86) 系统，地址是32位 (4个字节)，所以编译器会给指针变量分配4个字节的内存单元。不管多少重指针、也不管是什么类型的指针，由于指针本质上是一个地址，所以编译器会给不同的指针变量分配相同大小 (4个字节)的内存单元。
* short x[10][20];
* short \*y[10][20]; //怎么解释？
* short (\*z)[10][20];
* x是1个指向10个元素的数组，其中每个元素又包含20个元素，每个元素是short类型。所以x是1个2维short型的数组，x变量占10\*20\*2个字节的内存。
* y是1个指向10个元素的数组，其中每个元素又包含20个元素，每个元素是1个short类型的指针。所以y是1个2维short \*类型的数组，y变量占10\*20\*4个字节的内存。
* z是1个指针，指向10个元素的数组，其中每个元素又包含20个元素，每个元素是short类型。由于z是1个指针 (指向1个2维数组)，所以z变量占4个字节的内存。

1. 局部空间 (堆栈段)

* 用于分配局部变量 (生命期是函数内部)：
* 函数内部定义的非静态变量 (包括非静态const变量)、函数调用时的传入的实参、函数返回的对象
* 在堆栈段定义的变量都会被C++释放：
* 函数内部的局部变量在函数返回时被释放；函数调用时的传入的实参和函数返回的对象，在主程序调用完函数后被释放。



(2) 全局空间 (数据段)

* 全局空间上定义的变量 (对象), 生命期是整个程序的生命期。
* 定义在全局空间的变量 (对象) ：
* 非静态全局变量 (包括 const 对象), 不包括全局的 const 简单类型 (如 const int k)；静态变量 (局部和非局部), 不包括 static const 简单类型 (如 static const int k)；用指令申请的空间 (如new、malloc等) ；全局变量 (非静态)具有唯一性，作用域是整个程序。2个.cpp文件不能定义同名的全局变量。如果1个.cpp文件需要访问定义在另外1个.cpp文件的全局变量时，必须用extern声明。
* 非局部静态变量 (定义在函数外面的静态变量)，作用域是当前.cpp文件。2个不同.cpp文件可以定义同名静态变量。非局部静态变量可以与其他.cpp中的全局变量同名，但访问不到同名的其他.cpp中的全局变量。
* 局部静态变量 (定义在函数内部的静态变量), 作用域是当前函数内部(生命期是整个程序运行期间)。可以定义与非局部静态变量同名的局部静态变量。

(3) 常量空间 (const段)

一些常量的值被存贮在该空间。定义在该空间的常量是不能修改的，若强行修改会引起程序崩溃。

定义在const段的3个典型常量 (变量)：

* 常量字符串。类型是 const char \* ，而不是 char \*。字符串需要以 0 结尾，所以 “abc” 的长度是3，需要4个字节的存贮空间；所有(全局、局部、类内)的静态const简单变量 (如 static const int k)；全局非静态const简单变量 (如 const int k)
* const char \*p = “abc”;
* char c1 = p[1];
* char c2 = “abc”[1]; //???
* char c3 = “abc”[-1]; //???
* p[0] = ‘1’; //???
* (char \*)p[0] = ‘1’; //???
* char \*q = “abc”; //???
* char \*q = (char \*)“abc”; //???
* q[0] = ‘1’; //???
* (char \*)“abc”[1] = ‘1’; //???
* char s[20]; strcpy(s, “abc”);
* s[3] = ‘1’; //???

const int x = 1;

struct A {

int k;

const int i;

static const int j;

A( ): i (-1) { }

} a;

const int A::j = 2;

int main( ) {

\*(int \*)&x = 0; //语法正确, 程序奔溃

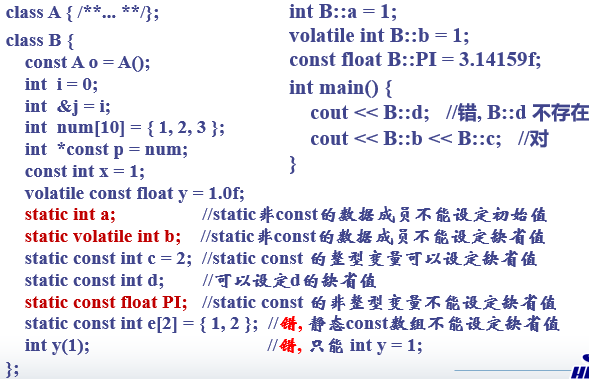
\*(int \*)&A::j = 0; //语法正确, 程序奔溃

\*(int \*)&a.i = 0; //正确, a.i = 0

}

《类内数据成员的缺省值》

* 在定义一个类时，一般可以设定类中数据成员的缺省值。但对于 static 数据成员，一般不能直接设定缺省值，一个例外的情况是“static const 整型变量”。 也就是说，可以在类内定义“static const 整型变量=整数”。
* 整型变量包括：bool、char、short、int、long、long long 及它们的 unsigned 型。
* 对于类内的“static const 整型变量”，一旦在类内设定了缺省值，这个静态变量也就创建了; 如果没有在类内设定缺省值，也没有在类外初始化，那么这个静态变量是不存在的。
* 对于类内的非static变量，即使设定了缺省值，也能重新赋值 (也可以在构造函数的初始化参数列表中重新初始化)。
* 类内数据成员，只能使用等于号(=)设定缺省值，不能直接使用圆括号的方式，即类内只能使用 int x=1，而不能使用 int x(1)。



《函数的调用与返回》

class A {

char \*p;

public:

A( ) { p = new char [10]; }

~A( ) { if(p) delete p; p = 0; }

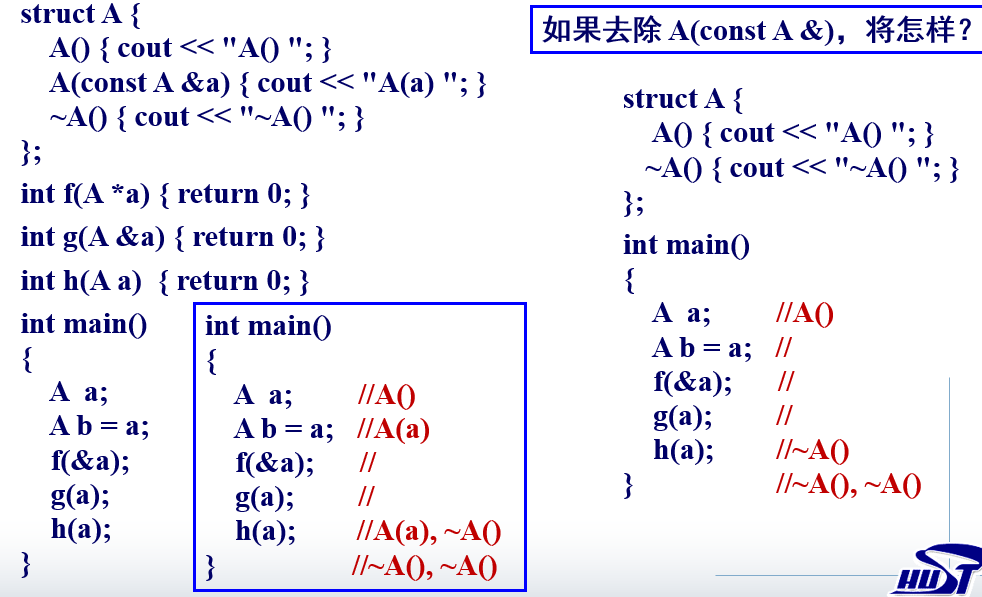
} a;

int f(A a) { return 0; }

int main( ) { f(a); } ——程序崩溃！

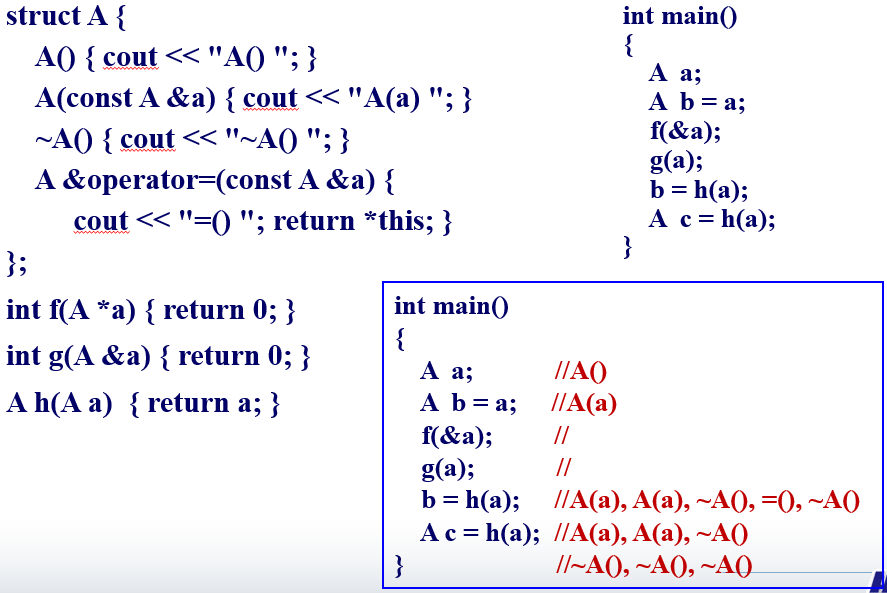
调用子程序时, 编译器在堆栈中构建所需要的参数。

* 如果参数是指针 (引用 )类型，则将实参的地址拷贝到堆栈。
* 如果参数是简单类型的变量，则将实参的值拷贝到堆栈。
* 如果参数是对象 (例如 A a)，则调用 A(const A &) 在堆栈中构建对象，该临时对象在失去作用范围时将被析构。



子程序返回机制：

* 如果返回类型是void，则不做任何事情。
* 如果返回类型是指针 (引用 )、简单类型(int、float等)，则地址、简单类型的值保存到EAX寄存器中。主程序从EAX获取返回值。
* 如果返回1个对象 (例如 A a)，则调用 A(const A &) 在堆栈中构建1个临时对象。主程序调用 operator=(const A &)将该临时对象赋值给变量，然后析构该临时对象。



《虚函数与多态》

(1) 虚函数定义

用virtual定义的成员函数 (虚函数必须是类的实例成员函数, 即有this指针的函数)。

class A {

int k;

public:

int f( ) { return k; }

virtual int g( ) { return 1; }

virtual static int h( ); //error, 不是实例成员函数

A(int k) { this->k = k; }

};

(2) 虚函数作用

在类的继承链中，实现动态多态。

* 用基类对象指针 (引用)指向类型派生类对象，通过这个基类指针(引用)去调用虚函数，就可实现动态多态 (基类和派生类中定义了函数原型相同的虚函数，运行时确定调用哪一个函数)。
* 虚函数只有在继承关系时才起作用。

(3) 虚函数的继承性

* + 一旦基类定义了虚函数，即使没有virtual声明，所有派生类中原型相同的非静态成员函数自动成为虚函数。
  + 构造函数构造对象的类型是确定的，不需根据类型表现出多态性，故不能定义为虚函数。
  + 析构函数可通过基类指针(引用)调用，基类指针指向的对象类型可能是不确定的，因此析构函数可定义为虚函数。

(4) 虚函数的多态性

假定如下的继承关系：C0 ← C1 ← … Ck ← … ← Cn 即，C0是祖先类，Cn是子孙类。有如下语句：

* Ck c; C0 \*p = (C0 \*)&c; p->f( ); int k = p->i;
* 或者：C0 &q = c; q.f( ); int k = q.i;

对于语句p->f()，编译器将会做如下工作：

(a)在C0类寻找函数 f()，如果没有找到或找到但f()不能访问，则报错；

(b)如果C0中有f()且可以访问，这时判断f()是否是虚函数，若不是虚函数则直接调用C0::f()；若f()是虚函数，则转下一步；

(c)沿着Ck到C0的方向，查找虚函数f()，只要发现某个类Cm (0<=m<=k)中重定义了f() (即使Cm::f()是private属性)，则调用Cm::f()。

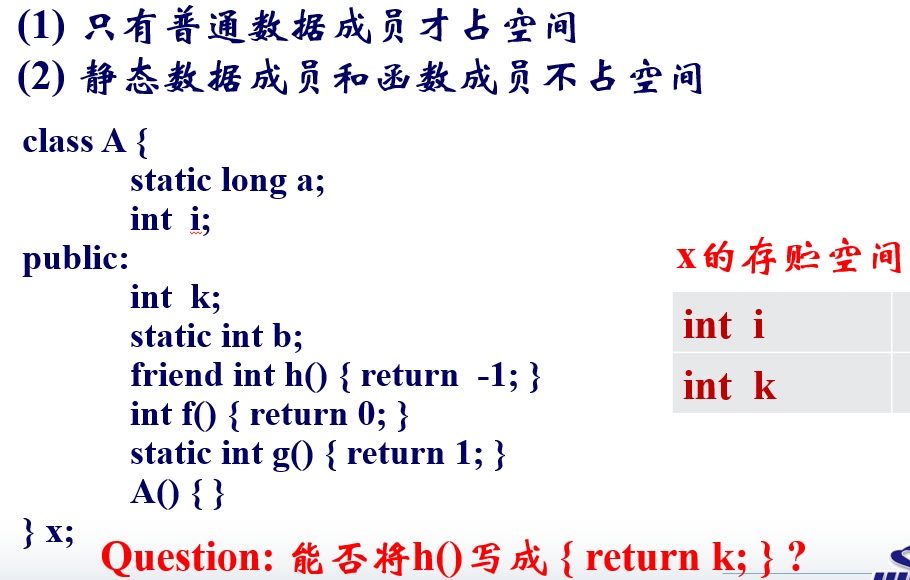
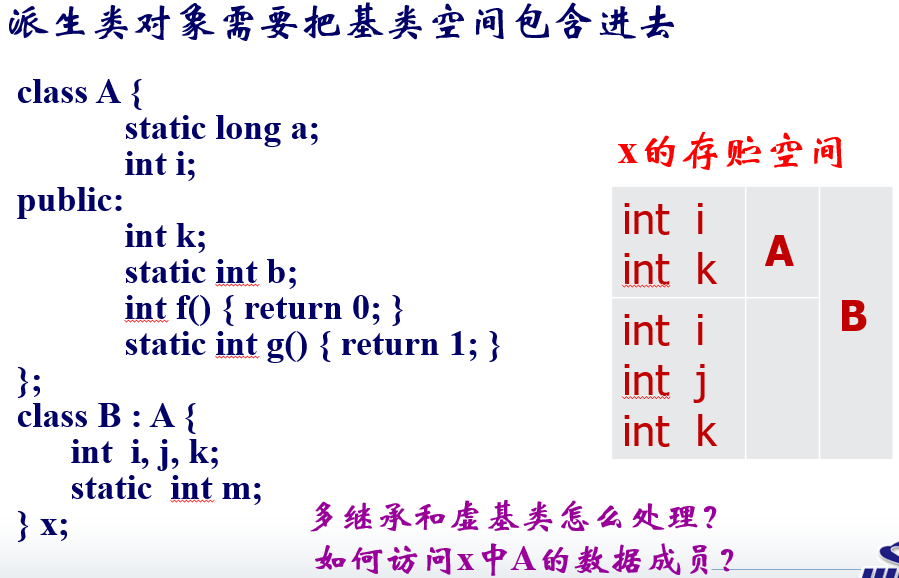
数据成员没有虚特性 (没有多态性)

* 对于语句 p->i，编译器在C0类寻找变量i，如果找到了且可以访问则直接使用；如果没有找到或找到了但 i 不能访问，则报错。

《类的存贮空间》

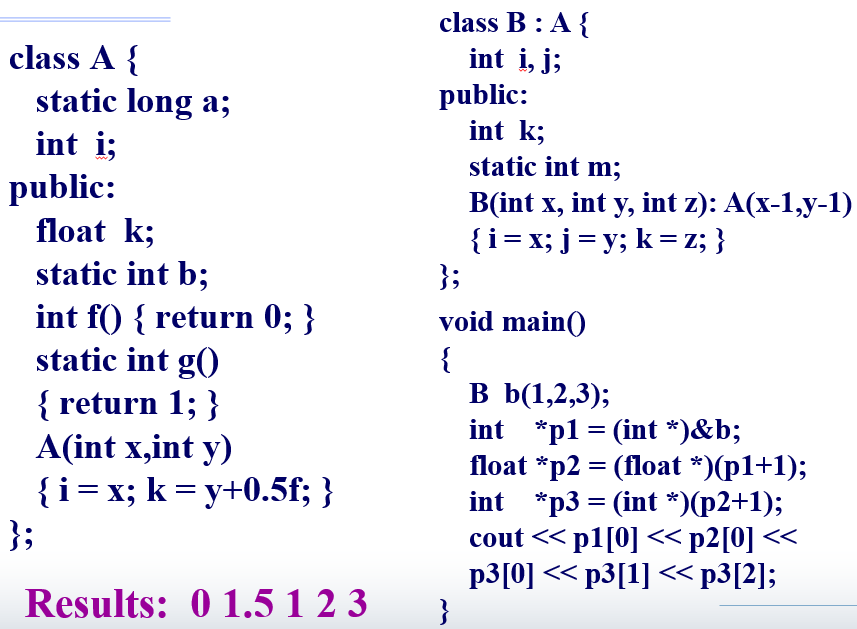
1. 简单类的存贮空间

2. 无虚函数的派生类存贮空间

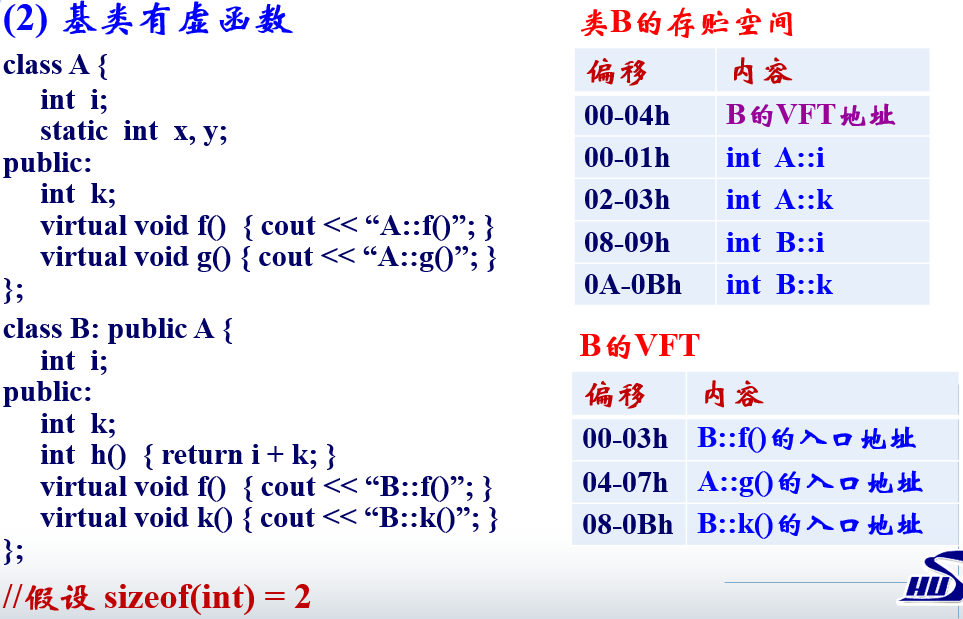
 

3. 访问对象 (无虚函数)中的数据成员

4. 有虚函数的简单类存贮空间

5. 有虚函数的派生类存贮空间

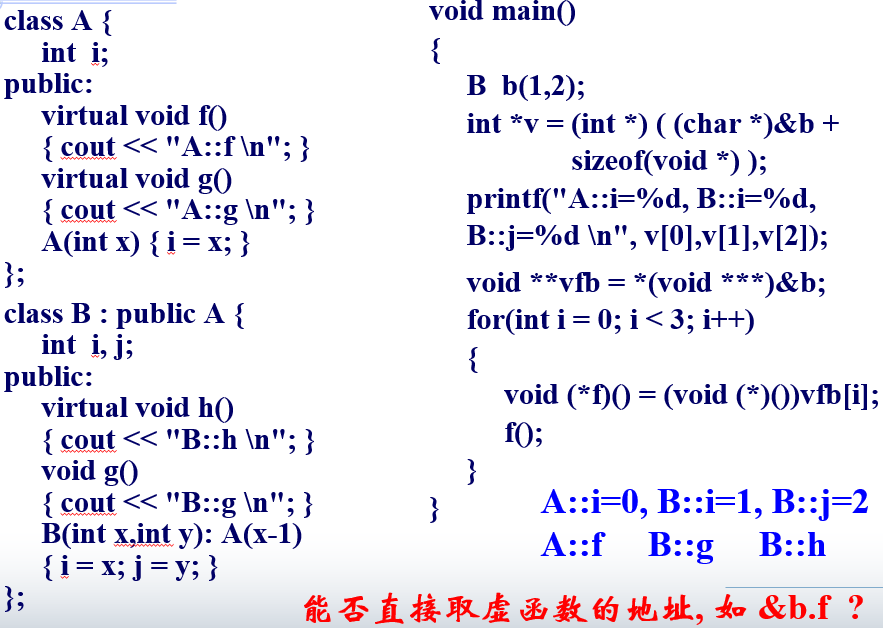
派生类VFT构造方法：

(1) 将基类A的VFT复制到派生类B的VFT的开始处；

(2) 若派生类B对基类A的某个虚函数f()进行了重定义，则在刚拷贝的基类VFT中，用B::f()的地址替换A::f()函数的入口地址；

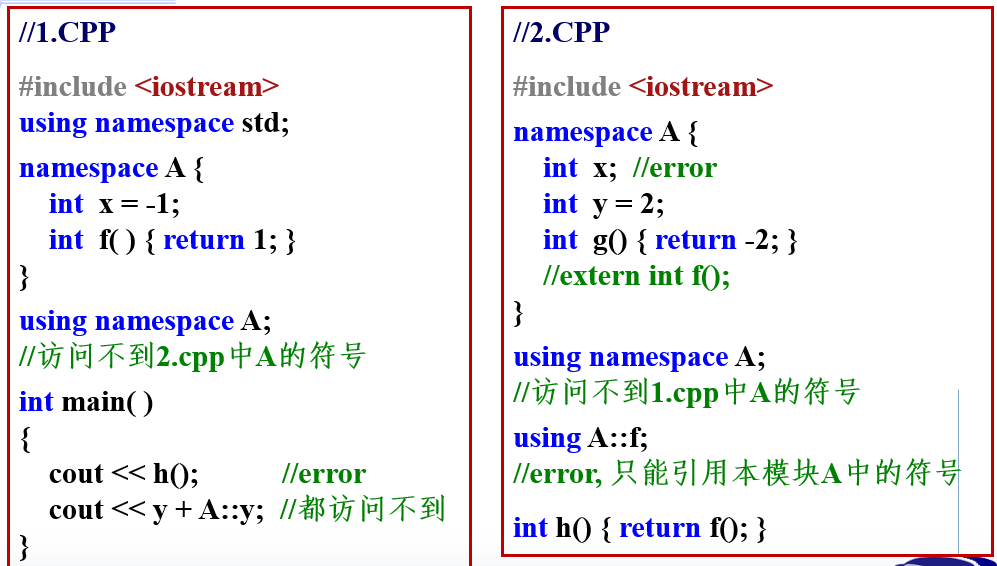
(3) 若在派生类B中定义了新的虚函数，则依次将这些新的虚函数的入口地址尾加到B的VFT。

6. 利用存贮空间访问变量和虚函数

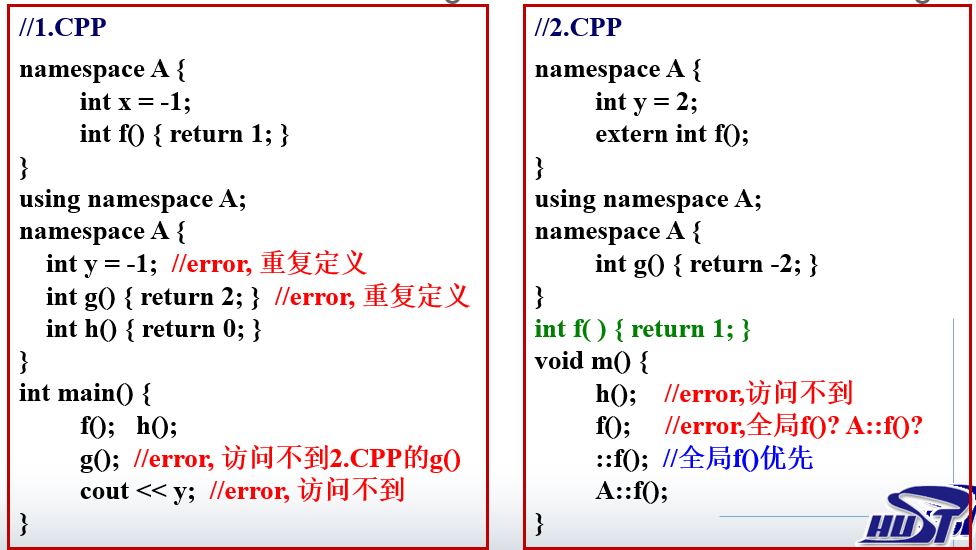


《名字空间》

* 可以在不同的模块中定义同名的名字空间。编译器会将不同模块中定义的同名的名字空间整合为一个空间。
* 每个模块中名字空间内的符号，作用域是当前模块。
* 引用其他模块中同名的名字空间内的符号时，需要在本模块的名字空间中用extern声明。
* 用using引用名字空间的符号时，只能引用本模块的名字空间中的符号。



* 名字空间 (包括匿名名字空间) 可以分多次定义。

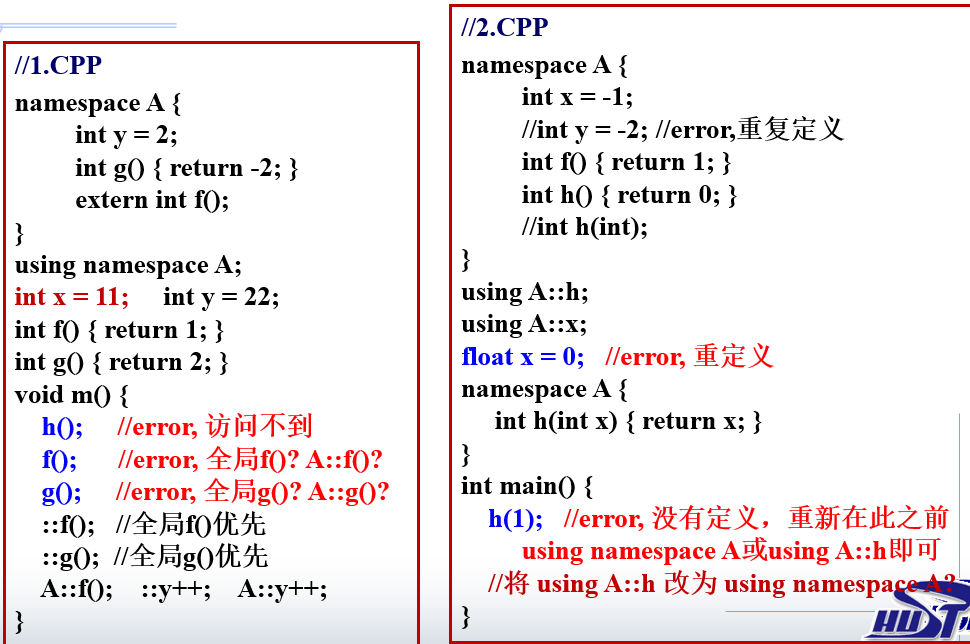


(1) using namespace 名字空间名称；

本模块可以访问这个名字空间中的所有符号 (即使该名字空间是多次定义的)。可以在当前模块中再定义和名字空间中同名的标识符。

(2) using 名字空间名称::成员名称；

* + 只能在本模块中访问这个成员。编译器将该成员的定义加入当前模块，不能在当前模块中再定义和该成员同名的标识符。
  + 引用的成员必须在用using引用前已经声明 (即使还没有定义也可以)。



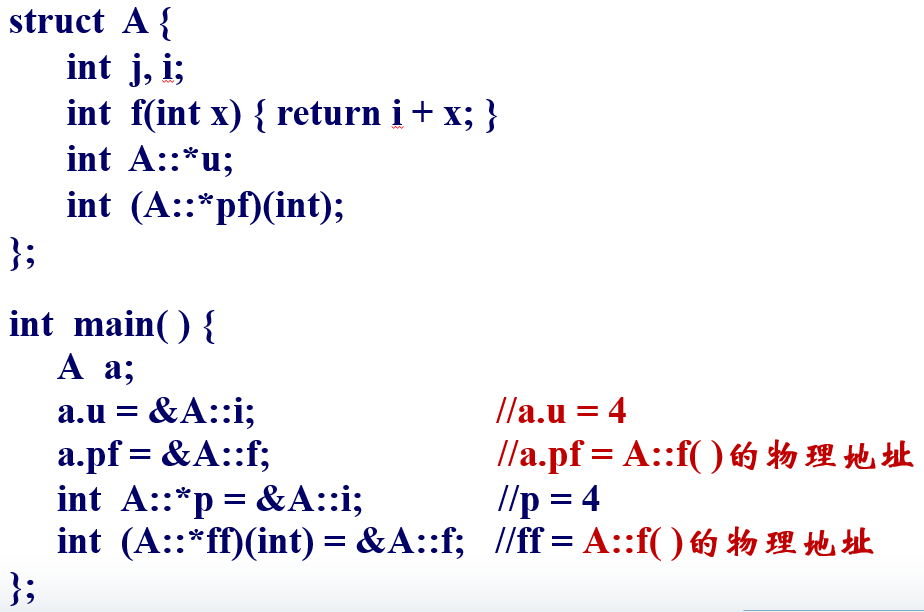
《深入理解成员指针》

(1) 定义实例成员指针

**实例数据成员指针：变量类型 类名::\*变量名；**

**实例函数成员指针：返回类型 (类名::\*变量名) (参数原型)；**

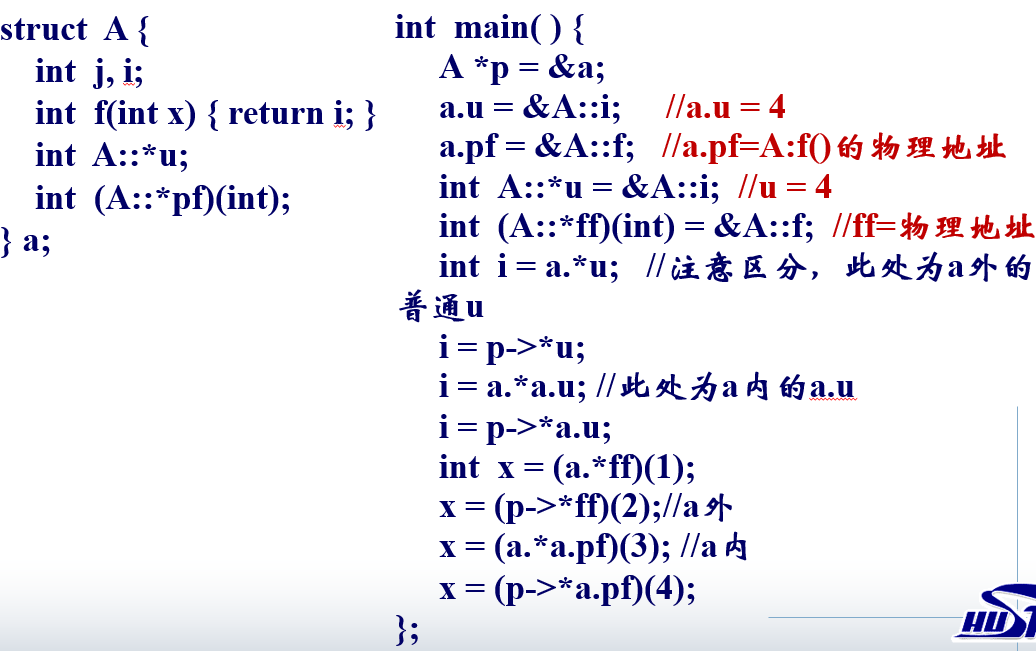
* 实例数据成员指针的值是数据成员（相对对象首地址）的偏移值
* 实例函数成员指针实际上是函数的物理地址



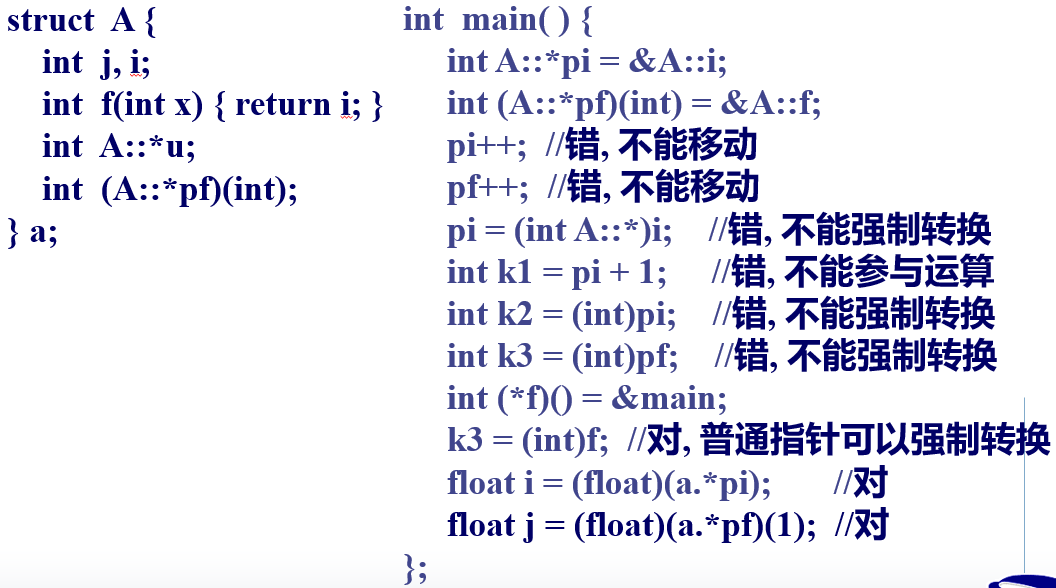
(2) 使用实例成员指针

必须结合对象(引用)使用 .\*、或结合对象指针 使用 ->\*, 来访问对象中的成员，用法是：

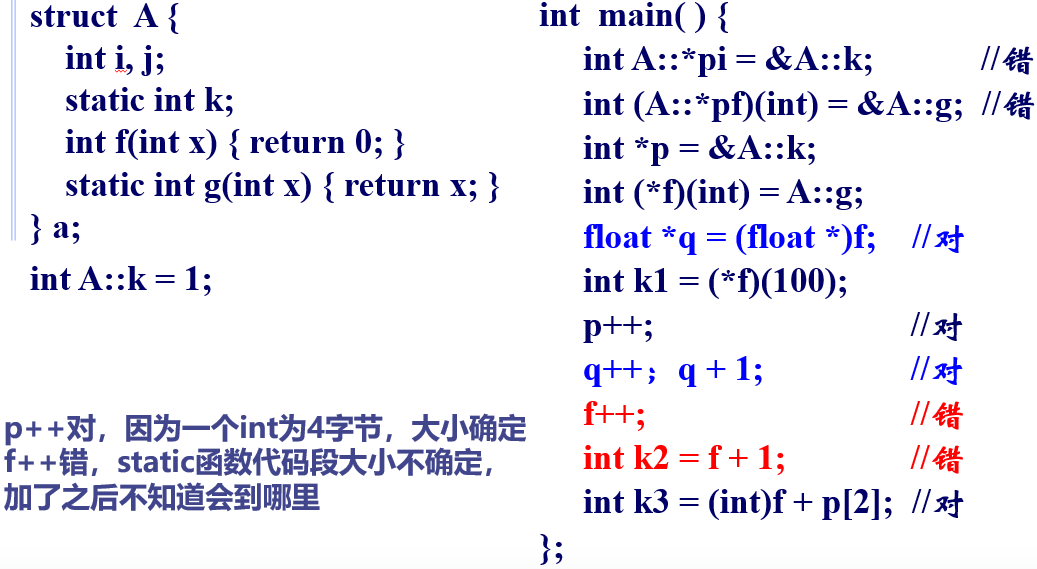
* 对象.\*数据成员指针；对象指针->\*数据成员指针；(对象.\*函数成员指针) (参数)；(对象指针->\*函数成员指针) (参数)



(3) 实例成员指针说明：不能移动，不能参与运算，不能强制类型转换。

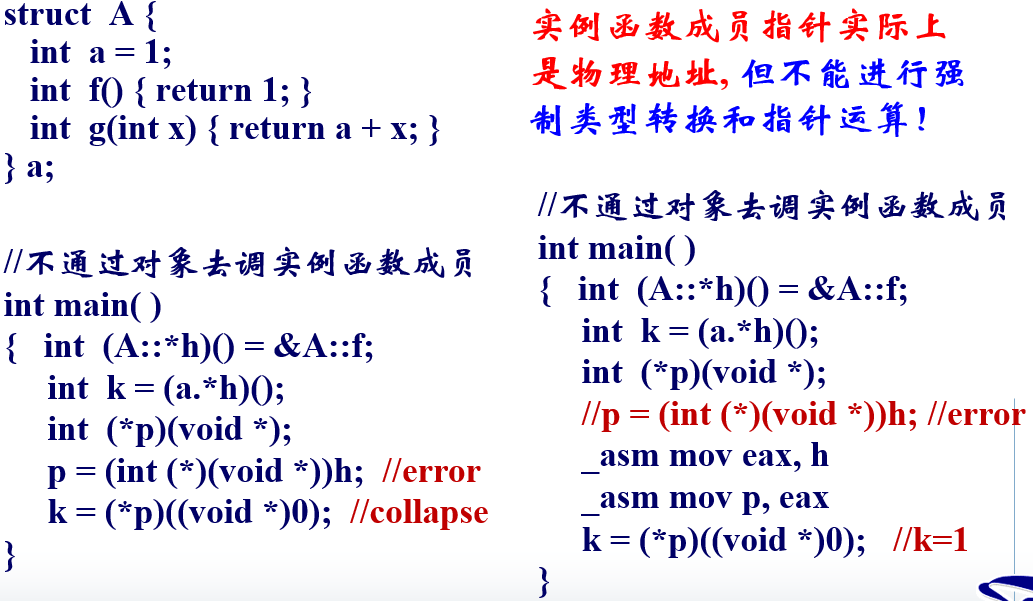


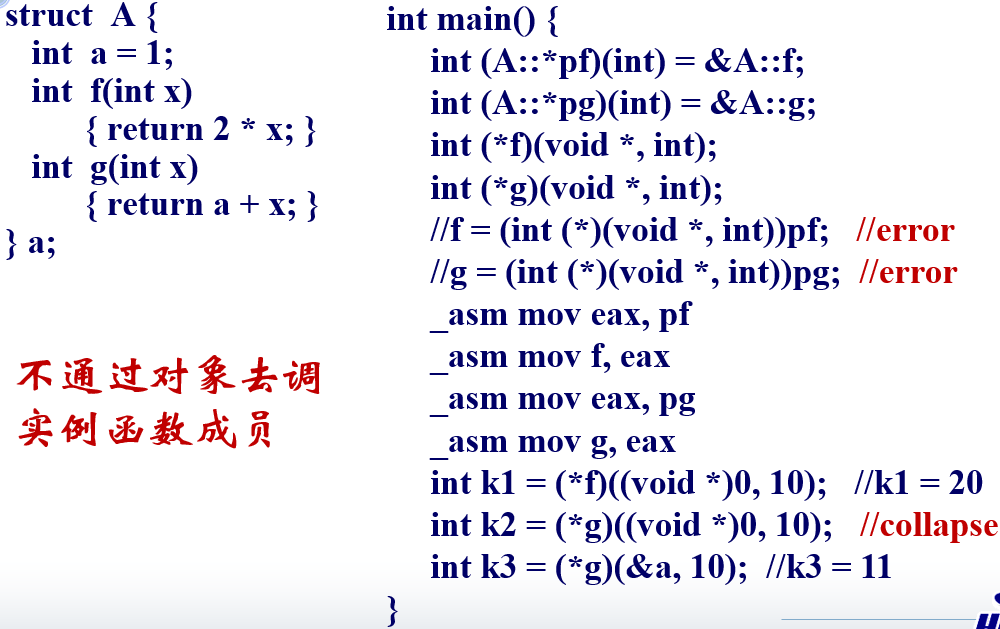
* 静态成员指针本质上是普通指针，需要以普通指针的形式去定义和访问，不能使用“类名::\*变量名”实例成员的方式去定义和访问。
* 静态成员指针的值是静态成员的物理地址。



* 类的函数成员的代码是不依赖于任何对象而独立存在的, 实例函数成员指针实际上是成员函数的物理地址。
* 能否不通过对象而直接调用实例函数成员呢？

(不通过“(a.\*f)(实参) ”的形式去调用实例成员函数)





《static\_extern\_friend等限定符》

static, extern 指明存贮模型, 修饰变量和函数, static 和 extern 不能同时使用

const, volatile 指明属性, 修饰变量和函数

mutable 只能用在类里面修饰非const的实例数据成员

virtual 只能用在类里面修饰实例函数成员

friend 修饰一个函数(普通函数、成员函数)

inline 修饰一个函数 (普通函数、成员函数)

* 对于用static声明的普通函数, 在定义函数体时static可以省略 (也可以不省略);
* 对于用static声明的类内成员函数, 在类外定义函数体时一定省略 static (像初始化类内定义的static变量一样)。
* static 和 extern 不能连用
* 只有对于 [static] const 变量，先声明后定义，才有意义 (先定义后声明 ⬄ 只定义不声明)，表明将该变量的作用域修改为整个程序。

例：extern const int z; //其他模块可以访问z

static const int z = 3;

static void h();

struct A {

static const int k; // = 1 ?;

static void f();

friend static void h();

//extern friend static void h(); //error

};

const int A::k = 10;

void A::f() { }

//static void A::f() { } //error

void h() { } //OK

//static void h() { } //OK

int main() { return 0; }

* static/extern 互斥(不能连用), 可以与const, volatile, inline, friend 相互连用; 不能与virtual, mutable连用.
* const, virtual, friend, inline 可以相互连用.
* mutable 只能与 volatile 连用, 其他限定词都不能与mutable 连用.