C++的标准库

華中科技大學

C++ 标准函数库

STL (Standard Template Library)

1. C++ 标准库 (1)

C++语言规范提供了一套标准的函数库，主要由3部分组成：

- C函数库
- IO流及本地化
- STL（Standard Template Library）

  STL由类模板和函数模板组成，约占C++标准库的80%。

1. C++ 标准库 (2)

- C++标准库的所有头文件都没有扩展名。
- C++标准库包含50个头文件，其中18个提供了C库的功能。
- 在C++标准库中，与宏相关的名称(如max等)在全局作用域中定义，其他名称则在std命名空间中声明。
- 在C++中还可以使用C库的头文件（带.h后缀）。

2. STL介绍 (1)

- STL（Standard Template Library），由类模板和函数模板组成，是C++标准库的子集，约占了C++标准库的80%。
- 在C++标准库中，STL被组织为13个头文件：<algorithm> 、 <deque> 、 <functional> 、<iterator> 、 <vector> 、 <list> 、 <map> 、<memory>、<numeric>、<queue>、<set>、<stack>、<utility>。

2. STL介绍 (2)

- STL包含6大组件：
  - 容器（Container）
  - 迭代器（Iterator）
  - 算法（Algorithm）
  - 仿函数（Functor）
  - 适配器（Adaptor）
  - 内存分配器（Allocator）

2.1 容器 (1)

- 容器是一种数据结构，用来存贮数据。如list、vector、deques等，以类模板的方法提供。为了访问容器中的数据，可以使用容器类中定义的迭代器。
- STL 提供 3 类标准容器：序列容器、排序容器、哈希容器。后两类容器有时也统称为关联容器。

2.1 容器 (2)

容器类别	描述
序列容器	包括 vector（向量）、list（列表）、deque（双端队列）。元素在容器中的位置同元素的值无关，即容器不是排序的。将元素插入容器时，指定在什么位置，元素就会位于什么位置。
排序容器	包括 set（集合）、multiset（多重集合）、map（映射）、multimap（多重映射）。排序容器中的元素默认是由小到大排序好的，即便是插入元素，元素也会插入到适当位置。
哈希容器	C++11 新加入 4 种关联式容器：unordered_set（哈希集合）、unordered_multiset（哈希多重集合）、unordered_map（哈希映射）、unordered_multimap（哈希多重映射）。哈希容器中的元素是未排序的，元素的位置由哈希函数确定。

2.2 迭代器 (1)

- 迭代器提供一种方法，顺序访问一个聚合对象中各个元素，不需暴露该对象的内部表示。每种容器类都会定义自己的迭代器。
- 迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。
- 迭代器也可以是那些定义了operator*()以及其他类似于指针操作的类对象。
- 迭代器重载了 *，++，--，==，!=， = 运算符。

2.2 迭代器 (2)

迭代器可分2类：

- 容器迭代器：访问容器中的数据
- 流迭代器：访问流数据

迭代器可实现为：正向访问、双向访问、随机访问。

- 流迭代器一般实现为正向迭代器。
- 容器迭代器一般具有双向访问功能，而有些流迭代器还具有随机访问功能。

2.2 迭代器 (3)

- **正向迭代器**

  假设p是一个正向迭代器（每次正向移动一个元素），则p支持以下操作：++p，p++，*p。两个正向迭代器可以互相赋值，还可以用 == 和 != 运算符进行比较。

- **双向迭代器**

  双向迭代器除了具有正向迭代器的功能外，还具有-- (前置和后置) 功能 (每次正向或反向移动一个元素)。若p是一个双向迭代器，则 ++p、p++、--p、p--、==、!= 都可以用。

2.2 迭代器 (4)

随机访问迭代器 (1)

随机访问迭代器具有双向迭代器的全部功能，同时还具有随机访问的功能（每次正向或反向移动若干个元素）。若p是一个随机访问迭代器，i是一个整型变量或常量，则：

++p、p++	向前移动1个元素（改变p）
--p、p--	向后移动1个元素（改变p）
p += i	p往后移动i个元素（改变p）
p -= i	p往前移动i个元素（改变p）
p + i	返回p后面第i个元素的迭代器（不改变p）
p – i	返回p前面第i个元素的迭代器（不改变p）
p[i]	返回p后面第i个元素的引用(不改变p), 等价 *(p + i)

2.2 迭代器(5)

随机访问迭代器(2)

- 2个随机访问迭代器 p1、p2 还可以用 <、>、<=、>= 运算符。p1<p2 表示p2的位于p1之后，即p1需要经过若干次（至少1次）++操作后，才会等于 p2。p2-p1表示 p2 和 p1所指位置之间的元素个数。

- 不是所有容器都支持上面这些迭代器。stack、queue 和 priority_queue 没有迭代器，但这些容器适配器提供一些成员函数去访问容器中的元素。

容器	迭代器功能
vector	随机访问
deque	随机访问
list	双向
set / multiset	双向
map / multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

2.2 迭代器 (6)

(1) 容器迭代器

表1. 常用的容器迭代器

迭代器类型	迭代器定义方法
正向迭代器	容器类名::iterator 迭代器名;
常量正向迭代器	容器类名::const_iterator 迭代器名;
反向迭代器	容器类名::reverse_iterator 迭代器名;
常量反向迭代器	容器类名::const_reverse_iterator 迭代器名;

2.2 迭代器 (7)

对于迭代器的 ++ 和 —运算，一般使用前置方式(++i)，而不建议使用后置方式(i++)，这是因为后置运算需要多生成一个临时的局部对象。

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    for(int n = 1; n <= 5; n++) {
        v.push_back(n);          //在容器v的尾部添加一个元素
    }
    vector<int>::iterator i;   //定义正向迭代器
    for(i = v.begin(); i != v.end(); ++i) {    //用迭代器遍历容器
        cout << *i << " ";        //1 2 3 4 5
        *i *= 10;                 //每个元素*10
    }
    //用反向迭代器遍历容器
    vector<int>::reverse_iterator j;
    for(j = v.rbegin(); j != v.rend(); ++j) {
        cout << *j << " ";        //50 40 30 20 10
    }
    return 0;
}
```

2.2 迭代器 (8)

(2) 流迭代器 (1)

输入流迭代器、输出流迭代器 (头文件：iterator)

输入流迭代器

istream_iterator <数据类型> 迭代器名(绑定的流=cin);

istream_iterator <数据类型> 迭代器名; //指向流的结束位置

输出流迭代器

ostream_iterator <数据类型> 迭代器名(绑定的流, 分隔符);

支持的操作

*p、++p、p++、==、!= 。

2.2 迭代器 (9)

(2) 流迭代器 (2)

```
#include<iostream>
#include<iterator>
#include<string>
using namespace std;
int main( ) {
    istream_iterator<string> in_iter(cin), eof;         //eof = Ctrl^Z
    ostream_iterator<string> out_iter(cout,"\n");  //元素分隔符为换行
    while(in_iter != eof) {
        string s = *in_iter;
        *out_iter = s;
        out_iter++;
        in_iter++;
    }
}
```

2.3 算法 (1)

- 算法是用来操作容器中的数据的模板函数。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关. STL提供了大约100个实现算法的模版函数，定义在3个头文件 <algorithm>、<numeric> 和 <functional> 中。
- <algorithm> 由很多模版函数组成的，可以认为每个函数在很大程度上都是独立的，比较常用的有：比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并，等等。
- <numeric> 包括几个在序列上进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。
- <functional>中则定义了一些模板类，用以声明函数对象。

2.3 算法 (2)

- STL中算法大致分为四类：
  - 非可变序列算法：不直接修改容器的内容（元素）。
  - 可变序列算法：可以修改容器的内容。
  - 排序算法：对序列进行排序、合并、搜索。
  - 数值算法：对容器内容进行数值计算。

2.4 仿函数 (1)

- 仿函数就是一个实现了operator()的类（这个类的对象就是函数对象），这样的类具有类似函数的行为（将类名当函数名一样使用）。
- C++在<functional>头文件中定义了如下三类仿函数：
  - 算术类仿函数： plus<T>、 minus<T>、 multiplies<T>、 divides<T>、negate<T> 等
  - 关系运算类仿函数： equal_to<T>、 not_equal_to<T>、 greater_equal<T> 等
  - 逻辑运算仿函数： logical_and<T> 、 logical_or<T> 、 logical_not<T> 等
- 调用方式：multiplies<int>()(10, 123)，结果为1230.

2.4 仿函数 (2)


```
#include <iostream>
#include <algorithm>
using namespace std;
class Sort {
public:
      bool operator()(int x, int y) const {
      return x > y;
   }
};
class Display {
public:
     void operator()(int x) const { cout << x; }
};
int main( ) {
    int a[5] = { 4, 1, 2, 5 };
    sort(a, a+4, Sort());   //Sort()是函数名, sort()内部会调用 Sort()(v1,v2)
    for_each(a, a+4, Display());
    //for_each(p1, p2, Display())  <=>  while(p1 != p2) Display()(*p1++)
    return 0;
}
```

2.5 适配器 (1)

- 容器适配器对容器进行包装，使其表现出另外一种行为。
- C++标准库提供了3种顺序容器适配器：queue (FIFO队列)、priority_queue (优先级队列)、stack (栈)。
- 适配器对容器进行包装，就可实现其他相应的功能。例如，利用适配器 stack 去包装容器 vector<int>（即 stack<int, vector<int>>）就实现了整型栈的功能。

2.5 适配器 (2)

華中科技大學

基础容器：能够被适配器进行装配的容器，基础容器必须满足一定的条件。

容器适配器	基础容器筛选条件	默认使用的基础容器
stack	基础容器需包含以下成员函数： • empty() • size() • back() • push_back() • pop_back() 满足条件的基础容器有 vector、deque、list。	deque
queue	基础容器需包含以下成员函数： • empty() • size() • front() • back() • push_back() • pop_front() 满足条件的基础容器有 deque、list。	deque
priority_queue	基础容器需包含以下成员函数： • empty() • size() • front() • push_back() • pop_back() 满足条件的基础容器有vector、deque。	vector

2.6 内存分配器

内存分配器为容器类模板提供自定义的内存申请和释放功能。一般情况下，只有高级用户才有改变内存分配策略的需求，因此内存分配器对于一般用户来说，并不常用。

2.7 程序示例 (1)


```
//程序 1：动态 1 维数组及排序
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> v = { 4, 2, 3, 1 };
   v.push_back(100);              //4, 2, 3, 1, 100
   sort(v.begin(), v.end());      //从小到大排序
   reverse(v.begin(), v.end());  //倒装数据
   for(int k = 0; k < v.size(); k++) {
      cout << v[k] << "    ";     //100 4 3 2 1
   }
   return 0;
}
```

```
//程序 2：动态 1 维数组及排序
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool compare(int x, int y)
{
      return x > y;    //降序排序
}
int main()
{
   vector<int> v = { 4, 2, 3, 1 };
   v.push_back(100);    //4, 2, 3, 1, 100
   sort(v.begin(), v.end(), compare);
   //reverse(v.begin(), v.end());
   for(int k = 0; k < v.size(); k++) {
         cout << v[k] << "    "; //100 4 3 2 1
   }
   return 0;
}
```

2.7 程序示例 (2)


```
//程序 3：动态 2 维数组
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int M = 3, N = 4;
    vector<vector<int>>
    v(M, vector<int>(N));
    int k = 1;
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            v[r][c] = k++;
            //v[r].push_back(k++);
        }
    }
    return 0;
}
```

```
//程序 4：不规则 2 维数组
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int M = 4;
    vector<vector<int>> v(M);
    for(int r = 0; r < v.size(); r++) {
        v[r].resize(r+1);
    }
    int k = 1;
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            v[r][c] = k++;
        }
    }
    for(int r = 0; r < v.size(); r++) {
        for(int c = 0; c < v[r].size(); c++) {
            cout << v[r][c] << "    ";
        }
        cout << "\n";      // 1
    }                       // 2 3
    return 0;               // 4 5 6
}                           // 7 8 9 10
```

C++的变量、常量、程序空间

華中科技大學

- 计算机内存
- 程序内存空间
- 变量
- 常量

华中科技大学 计算机学院

金良海 2021.09

1. 内存 (1)

- 计算机的主存贮器是用来存放程序和数据的。
- 主存贮器是以字节为单位的，每个字节能存贮8位二进制数据。
- 80X86的主存贮器一般是由内存条组成的。

内存条：

32K
640K
1M
128M
512M
1G
……

1. 内存 (2)

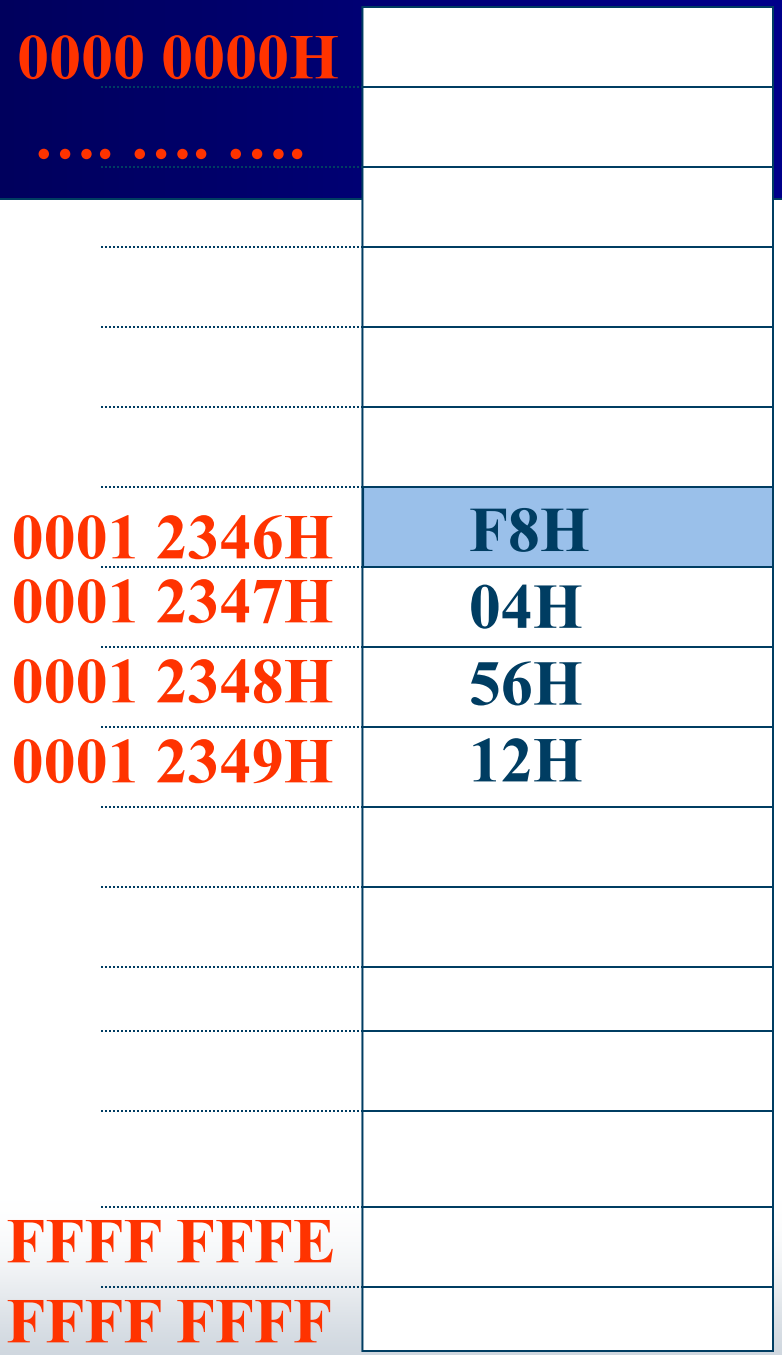
- 为了访问内存中的每个存贮单元（字节），需要给每个存贮单元一个编号，这就是**地址**。
- 内存的地址一般从0开始编号，编号是连续的。例如，1M内存表示有1M (1024*1024)个字节，其地址范围为 00000～FFFFF (16进制)。对于win32 (x86)，能管理4G内存，地址范围为：0000 0000～FFFF FFFF。

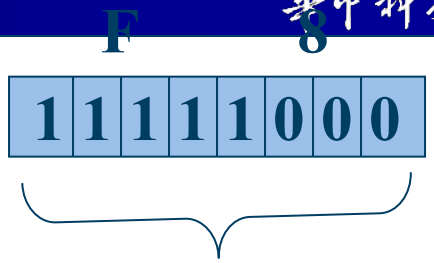
1. 内存 (3)

华中科技大学

每一个字节都有一个地址

字节是最小的寻址单位





8个位组成一个字节 BYTE

Q: 1M字节内存，地址编码需要多少二进制位？

Q:32位地址对应的内存大小可达到多大？

2. 程序内存空间

- 正在运行中的程序所占用的内存空间包括：程序代码指令空间、各种变量占用的空间、程序动态申请的空间 (new、malloc等)。
- 不管程序是正常退出 (main()中的return)还是异常退出 (exit、abort)，程序代码指令空间和各种变量占用的空间 都会被OS自动释放，而 程序动态申请的空间 则需要显式用指令去释放。

3. 变量 (1)

- 高级语言中的任何变量，编译器都会给它分配内存单元，内存单元的大小由变量类型决定。（变量＝内存单元）

```
char c;        //给c分配1个字节的内存单元
short x;       //给x分配2个字节的内存单元
int y;         //给y分配4个字节的内存单元
int a[10];     //给a分配40个字节的内存单元
int b[3][5];   //给b分配60个字节的内存单元
```

3. 变量 (2)

- 指针。指针（不管多少重指针）也是变量，所以，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。

3. 变量 (3)

对于win32：

```
char *p1;   int **p2;
double ***p3;   int (*p4)[100];
```

编译器会给每个指针变量 p1、p2、p3、p4 分配4个字节的内存单元。

```
struct A {
    long  i, j;
    char  buf[10];
} a[10];                    //180个字节的内存单元
struct A *p5 = a;           //4个字节的内存单元
```

3. 变量 (4)


```
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。

4. 常量 (1)

- **常量包括:**
  - 常量字符串。
  - const变量，包括全局和局部const变量、所有的静态const变量 (包括类的静态const变量)。

4. 常量 (2)

- **字符串常量**

  字符串常量的类型是 const 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。因为“abc”的类型是 const char *，所以可以 char c = “abc”[2]，但不能“abc”[2] = ‘1’。

4. 常量 (3)

- const变量

  const变量包括：全局和局部const变量(不包括类内定义的实例const变量)、所有的静态const变量（包括类内的静态const变量）。

4. 常量 (4)


```
const int x = 1;
int *const p = &y;  //int y = 0
static const int z = 2;
void f( ) {
    const int m = 11;
    static const int n = 12;
    … …
}
class  A {
     const int k;   //k不是常量
     static const int j;
     …  …
}
```

这些const变量 (除了y和k)，即 x、p、z、m、n、j，都是常量。

4. 常量 (5)

- **常量数据的存贮**

  OS为每个程序开辟一个只读的内存区域，用于保存常量数据 (常量字符串和 const变量)。也就是说，常量占据内存单元，但这些内存单元是不能写入的。

```
const int x = 1;
*(int *)&x = 2;   //语法正确 (因此x是左值)，但修改不了x的值
```

4. 常量 (6)


```
const int x = 1;
struct  A {
      int  k;
      const int  i;   //i不是常量
      static const int  j;
      A( ): i (-1) { }
} a;
const int  A::j = 2;

int main( ) {
    *(int *)&x = 0;        //语法正确,但修改不了x的值
    *(int *)&A::j = 0;     //语法正确,但修改不了A::j的值
    *(int *)&a.i = 0;      //正确, a.i = 0
}
```

想一想, why?

为什么可以修改类的const实例数据成员?

The end.

C++的变量

華中科技大學

- 计算机内存
- 变量

华中科技大学 计算机学院

金良海 2021.09

1. 内存 (1)

- 计算机的主存贮器是用来存放程序和数据的。
- 主存贮器是以字节为单位的，每个字节能存贮8位二进制数据。
- 80X86的主存贮器一般是由内存条组成的。

内存条：

32K
640K
1M
128M
512M
1G
……

1. 内存 (2)

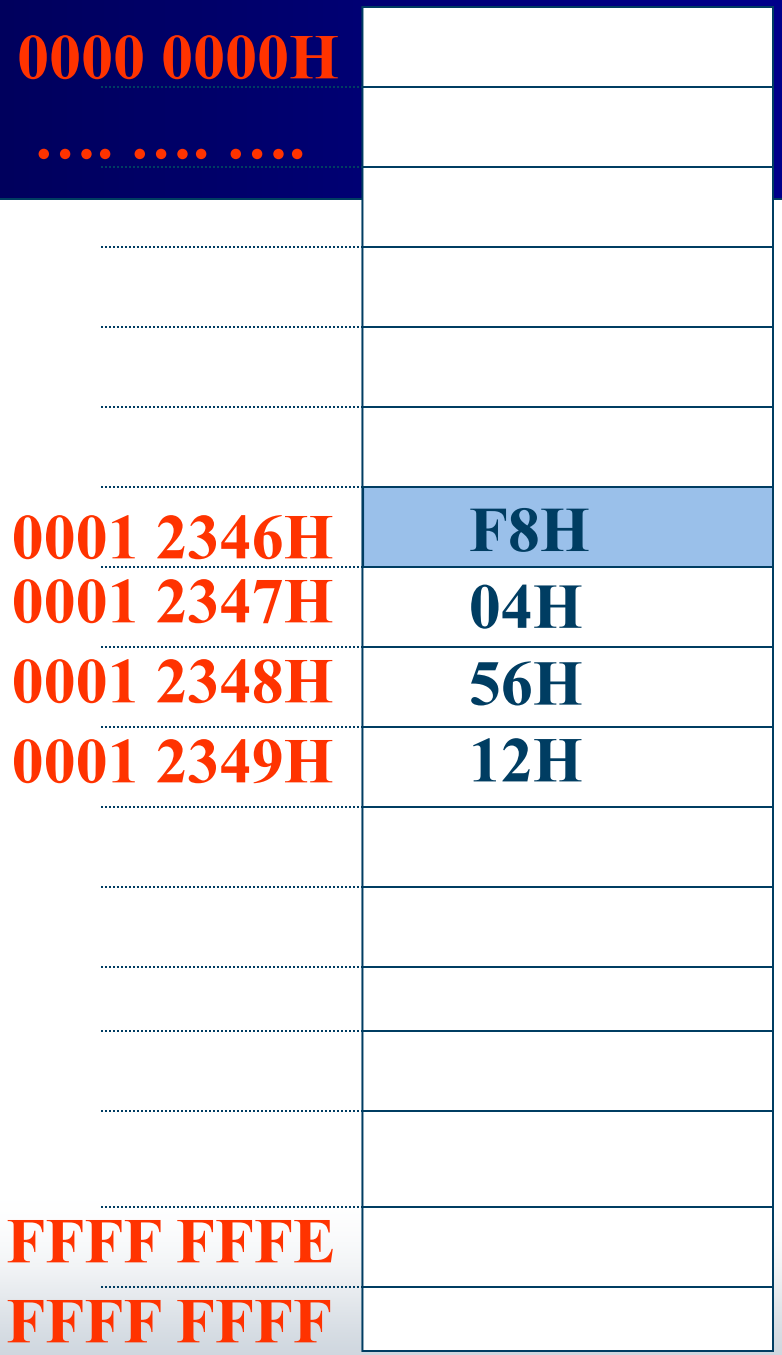
- 为了访问内存中的每个存贮单元（字节），需要给每个存贮单元一个编号，这就是**地址**。
- 内存的地址一般从0开始编号，编号是连续的。例如，1M内存表示有1M (1024*1024)个字节，其地址范围为 00000～FFFFF (16进制)。对于win32 (x86)，能管理4G内存，地址范围为：0000 0000～FFFF FFFF。

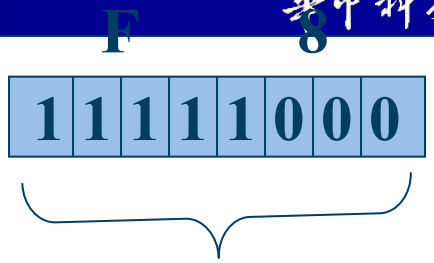
1. 内存 (3)

华中科技大学

每一个字节都有一个地址

字节是最小的寻址单位





8个位组成一个字节 BYTE

Q: 1M字节内存，地址编码需要多少二进制位？

Q:32位地址对应的内存大小可达到多大？

2. 变量 (1)

- 高级语言中的任何变量，编译器都会给它分配内存单元，内存单元的大小由变量类型决定。（变量＝内存单元）

```
char c;        //给c分配1个字节的内存单元
short x;       //给x分配2个字节的内存单元
int y;         //给y分配4个字节的内存单元
int a[10];     //给a分配40个字节的内存单元
int b[3][5];   //给b分配60个字节的内存单元
```

2. 变量 (2)

- **指针。**指针（不管多少重指针）也是变量，所以，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。

2. 变量 (3)

对于win32：

```
char *p1;   int **p2;
double ***p3;   int (*p4)[100];
```

编译器会给每个指针变量 p1、p2、p3、p4 分配4个字节的内存单元。

```
struct A {
    long  i, j;
    char  buf[10];
} a[10];                    //180个字节的内存单元
struct A *p5 = a;           //4个字节的内存单元
```

1. 内存 (4)


```
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。

The end.

类内数据成员的缺省值

华中科技大学 计算机学院

全良海 2021.09

类内数据成员的缺省值(1)

- 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。

类内数据成员的缺省值(2)

- 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)。

类内数据成员的缺省值 (3)


```
class A { /**... **/};
class B {
    const A o = A();
    int  i = 0;
    int  &j = i;
    int  num[10] = { 1, 2, 3 };
    int  *const p = num;
    const int x = 1;
    volatile const float y = 1.0f;
    static int a;                 //static非const的数据成员不能设定初始值
    static volatile int b;        //static非const的数据成员不能设定缺省值
    static const int c = 2;       //static const 的整型变量可以设定缺省值
    static const int d;           //可以设定d的缺省值
    static const float PI;        //static const 的非整型变量不能设定缺省值
    static const int e[2] = { 1, 2 };  //错, 静态const数组不能设定缺省值
    int y(1);                          //错, 只能 int y = 1;
};

int B::a = 1;
volatile int B::b = 1;
const float B::PI = 3.14159f;
int main() {
    cout << B::d;   //错, B::d 不存在
    cout << B::b << B::c;   //对
}
```

The end.

函数的调用与返回 (1)


```
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！

程序有问题吗?

函数的调用与返回 (2)

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。

函数的调用与返回 (3)


```
struct A {
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(a) "; }
    ~A() { cout << "~A() "; }
};
int f(A *a) { return 0; }
int g(A &a) { return 0; }
int h(A a)  { return 0; }
int main()
{
    A  a;
    A b = a;
    f(&a);
    g(a);
    h(a);
}
```

```
int main()
{
    A  a;        //A()
    A b = a;     //A(a)
    f(&a);       //
    g(a);        //
    h(a);        //A(a), ~A()
}                //~A(), ~A()
```

如果去除 A(const A &)，将怎样？

```
struct A {
    A() { cout << "A() "; }
   ~A() { cout << "~A() "; }
};
int main()
{
    A  a;        //A()
    A b = a;     //
    f(&a);       //
    g(a);        //
    h(a);        //~A()
}                //~A(), ~A()
```

函数的调用与返回 (4)

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。

函数的调用与返回 (5)


```
struct A {
    A() { cout << "A() "; }
    A(const A &a) { cout << "A(a) "; }
    ~A() { cout << "~A() "; }
    A &operator=(const A &a) {
        cout << "=() "; return *this; }
};
int f(A *a) { return 0; }
int g(A &a) { return 0; }
A h(A a)  { return a; }
```

```
int main()
{
    A  a;
    A  b = a;
    f(&a);
    g(a);
    b = h(a);
    A  c = h(a);
}
```

```
int main()
{
    A  a;          //A()
    A  b = a;      //A(a)
    f(&a);         //
    g(a);          //
    b = h(a);      //A(a), A(a), ~A(), =(), ~A()
    A c = h(a);    //A(a), A(a), ~A()
}                  //~A(), ~A(), ~A()
```

函数的调用与返回 (6)


```
class A {
   int  i, j, k;
public:
   A(int x) {  i = x;
               cout << "Ad" << i;
            }
   A(const A &a) {  i = a.i;
                    cout << "Aa" << i;
                 }
   ~A() { cout << "~A" << i; }
   friend A abs(A a);
};

A  abs(A a)
{
  A  b( a.i < 0? -a.i : a.i );
  return b;
}
```

```
int main( )
{
    A a(-1), b(2);
    b = abs(a);
    return 0;
}
```

```
A a(-1), b(2);  //Ad-1 Ad2
b = abs(a);     //Aa-1 Ad1 Aa1 ~A1 ~A-1 ~A1
return 0;       //~A2 ~A-1
```

函数的调用与返回 (7)


```
class A {
    int  i, j, k;
public:
    A(int x) {  i = x;
                cout << "Ad" << i;
             }
    A(const A &a) {  i = a.i;
                     cout << "Aa" << i;
                  }
    ~A() { cout << "~A" << i; }
    friend A abs(A &a);
};

A  abs(A &a)
{
   A  b( a.i < 0? -a.i : a.i );
   return b;
}
```

```
int main( )
{
    A a(-1), b(2);
    b = abs(a);
    return 0;
}
```

```
A a(-1), b(2);  //Ad-1Ad2
b = abs(a);     //Ad1Aa1~A1~A1
return 0;       //~A2~A-1
```

1. 简单类的存贮空间

(1) 只有普通数据成员才占空间

(2) 静态数据成员和函数成员不占空间

```
class A {
        static long a;
        int  i;
public:
        int  k;
        static int b;
        friend int h() { return  -1; }
        int f() { return 0; }
        static int g() { return 1; }
        A() { }
} x;
```

x的存贮空间

int i	
int k	

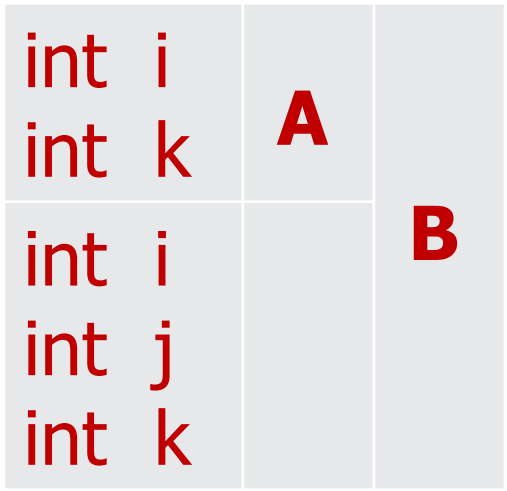
Question: 能否将h()写成 { return k; } ?

2. 无虚函数的派生类存贮空间

派生类对象需要把基类空间包含进去

```
class A {
        static long a;
        int i;
public:
        int k;
        static int b;
        int f() { return 0; }
        static int g() { return 1; }
};
class B : A {
     int  i, j, k;
     static  int m;
} x;
```

x的存贮空间



多继承和虚基类怎么处理？

如何访问x中A的数据成员？

3. 访问对象(无虚函数)中的数据成员


```
class A {
   static long a;
   int  i;
public:
   float  k;
   static int b;
   int f() { return 0; }
   static int g()
   { return 1; }
   A(int x,int y)
   { i = x; k = y+0.5f; }
};
```

```
class B : A {
   int  i, j;
public:
   int  k;
   static int m;
   B(int x, int y, int z): A(x-1,y-1)
   { i = x; j = y; k = z; }
};
void main()
{
   B  b(1,2,3);
   int    *p1 = (int *)&b;
   float *p2 = (float *)(p1+1);
   int    *p3 = (int *)(p2+1);
   cout << p1[0] << p2[0] <<
   p3[0] << p3[1] << p3[2];
}
```

Results: 0 1.5 1 2 3

4. 有虚函数的简单类存贮空间

偏移 00 - 03：虚函数表VFT的地址
偏移04开始：非static的数据成员

```
class A {
    int  i, j;
    static  int  x, y;
    virtual void f( ) {  cout << "f()"; }
public:
    int  k;
    virtual void g() { cout << "g()"; }
    int h() { return i + k; }
};
```

//假设 sizeof(int) = 2

类A的存贮空间

偏移	内容
00-03h	VFT地址
03-04h	int i
05-06h	int j
07-08h	int k

VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

5. 有虚函数的派生类存贮空间 (1)

(1) 基类没有虚函数(派生类有)

```
class A {
    int  i;
    static  int  x, y;
public:
    int  k;
    int  h() { return i + k; }
};
class B: public A {
    int  i;
public:
    int  k;
    int  h() { return i + k; }
    virtual void f()  { cout << "B::f()"; }
    virtual void g() { cout << "B::g()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-01h	int A::i
02-03h	int A::k
04-07h	B的VFT地址
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

5. 有虚函数的派生类存贮空间 (2)

(2) 基类有虚函数

```
class A {
    int  i;
    static  int  x, y;
public:
    int  k;
    virtual void f()  { cout << "A::f()"; }
    virtual void g() { cout << "A::g()"; }
};
class B: public A {
    int  i;
public:
    int  k;
    int  h()  { return i + k; }
    virtual void f()  { cout << "B::f()"; }
    virtual void k() { cout << "B::k()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-04h	B的VFT地址
00-01h	int A::i
02-03h	int A::k
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	B::f()的入口地址
04-07h	A::g()的入口地址
08-0Bh	B::k()的入口地址

5. 有虚函数的派生类存贮空间 (3)

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

華中科技大學

```
class A {
    int  i;
public:
    virtual void f()
    { cout << "A::f \n"; }
    virtual void g()
    { cout << "A::g \n"; }
    A(int x) { i = x; }
};
class B : public A {
    int  i, j;
public:
    virtual void h()
    { cout << "B::h \n"; }
    void g()
    { cout << "B::g \n"; }
    B(int x,int y): A(x-1)
    { i = x; j = y; }
};
```

```
void main()
{
    B  b(1,2);
    int *v = (int *) ( (char *)&b +
                sizeof(void *) );
    printf("A::i=%d, B::i=%d,
    B::j=%d \n", v[0],v[1],v[2]);
    void **vfb = *(void ***)&b;
    for(int i = 0; i < 3; i++)
    {
        void (*f)() = (void (*)())vfb[i];
        f();
    }
}
```

A::i=0, B::i=1, B::j=2

A::f B::g B::h

能否直接取虚函数的地址，如 &b.f ？

深入理解成员指针

成员指针

- 实例成员指针(数据成员和函数成员)
- 静态成员指针(数据成员和函数成员)
- 深入理解实例成员指针

1. 实例成员指针 (1)

(1) 定义成员指针

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的偏移值；
- 实例函数成员指针实际上是函数的物理地址。

1. 实例成员指针 (2)


```
struct  A {
      int  j, i;
      int  f(int x) { return i + x; }
      int  A::*u;
      int  (A::*pf)(int);
};

int  main( ) {
     A  a;
     a.u = &A::i;                    //a.u = 4
     a.pf = &A::f;                   //a.pf = A::f( )的物理地址
     int  A::*p = &A::i;             //p = 4
     int  (A::*ff)(int) = &A::f;     //ff = A::f( )的物理地址
};
```

1. 实例成员指针 (3)

(2) 使用成员指针

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：

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)

1. 实例成员指针 (4)


```
struct  A {
    int  j, i;
    int  f(int x) { return i; }
    int  A::*u;
    int  (A::*pf)(int);
} a;
```

```
int  main( ) {
    A *p = &a;
    a.u = &A::i;      //a.u = 4
    a.pf = &A::f;    //a.pf=A:f()的物理地址
    int  A::*u = &A::i;  //u = 4
    int  (A::*ff)(int) = &A::f;  //ff=物理地址
    int  i = a.*u;
    i = p->*u;
    i = a.*a.u;
    i = p->*a.u;
    int  x = (a.*ff)(1);
    x = (p->*ff)(2);
    x = (a.*a.pf)(3);
    x = (p->*a.pf)(4);
};
```

1. 实例成员指针 (5)

(3) 成员指针说明

- 实例数据成员指针是数据成员相对于对象首地址的偏移。
- 实例函数成员指针是函数成员的物理地址。
- 实例成员指针：不能移动；
  不能参与运算；
  不能强制类型转换。

1. 实例成员指针 (6)


```
struct  A {
    int  j, i;
    int  f(int x) { return i; }
    int  A::*u;
    int  (A::*pf)(int);
} a;
```

```
int  main( ) {
    int A::*pi = &A::i;
    int (A::*pf)(int) = &A::f;
    pi++;  //错, 不能移动
    pf++;  //错, 不能移动
    pi = (int A::*)i;    //错, 不能强制转换
    int k1 = pi + 1;     //错, 不能参与运算
    int k2 = (int)pi;    //错, 不能强制转换
    int k3 = (int)pf;    //错, 不能强制转换
    int (*f)() = &main;
    k3 = (int)f;  //对, 普通指针可以强制转换
    float i = (float)(a.*pi);       //对
    float j = (float)(a.*pf)(1);  //对
};
```

2. 静态成员指针 (1)

- 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。

2. 静态成员指针 (2)


```
struct  A {
    int i, j;
    static int k;
    int f(int x) { return 0; }
    static int g(int x) { return x; }
} a;
int A::k = 1;
```

```
int  main( ) {
    int A::*pi = &A::k;            //错
    int (A::*pf)(int) = &A::g;     //错
    int *p = &A::k;
    int (*f)(int) = A::g;
    int k1 = f(100);
    p++;                           //对
    f++;                           //错
    int k2 = f + 1;                //错
    int k3 = (int)f + p[2];        //对
};
```

3. 深入理解函数成员指针 (1)

- 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)

3. 深入理解函数成员指针 (2)


```
struct  A {
   int  a = 1;
   int  f() { return 1; }
   int  g(int x) { return a + x; }
} a;


//不通过对象去调实例函数成员
int main( )
{   int  (A::*h)() = &A::f;
    int  k = (a.*h)();
    int  (*p)(void *);
    p = (int (*)(void *))h;  //error
    k = (*p)((void *)0);  //collapse
}
```

实例函数成员指针实际上是物理地址，但不能进行强制类型转换和指针运算！

```
//不通过对象去调实例函数成员
int main( )
{   int  (A::*h)() = &A::f;
    int  k = (a.*h)();
    int  (*p)(void *);
    //p = (int (*)(void *))h; //error
    _asm mov eax, h
    _asm mov p, eax
    k = (*p)((void *)0);   //k=1
}
```

3. 深入理解函数成员指针 (3)


```
struct  A {
   int  a = 1;
   int  f(int x)
        { return 2 * x; }
   int  g(int x)
        { return a + x; }
} a;
```

不通过对象去调
实例函数成员

```
int main() {
    int (A::*pf)(int) = &A::f;
    int (A::*pg)(int) = &A::g;
    int (*f)(void *, int);
    int (*g)(void *, int);
    //f = (int (*)(void *, int))pf;   //error
    //g = (int (*)(void *, int))pg;  //error
    _asm mov eax, pf
    _asm mov f, eax
    _asm mov eax, pg
    _asm mov g, eax
    int k1 = (*f)((void *)0, 10);   //k1 = 20
    int k2 = (*g)((void *)0, 10);   //collapse
    int k3 = (*g)(&a, 10);  //k3 = 11
}
```

The end.

虚函数与多态 (1)

(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; }
};
```

虚函数与多态(2)

(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。

虚函数与多态(3)

(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**。

虚函数与多态(4)

(4) 虚函数的多态性(1)

假定如下的继承关系：

C0 ← C1 ← … Ck ← … ← Cn

即，C0是祖先类，Cn是子孙类。

有如下的语句：

```
Ck  c;
C0  *p = (C0 *)&c;
p->f( );
int  k = p->i;
```

或者：

```
C0 &q = c;
q.f( );
int  k = q.i;
```

虚函数与多态(5)

(4) 虚函数的多态性(2)

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中重定义了f()，则调用Cm::f()。

虚函数与多态(6)

(4) 虚函数的多态性(3)

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。

The end.

1. 类的存贮空间

(1) 只有普通数据成员才占空间

(2) 静态数据成员和函数成员不占空间

1.1 简单类的存贮空间

(1) 只有普通数据成员才占空间

(2) 静态数据成员和函数成员不占空间

```
class A {
        static long a;
        int  i;
public:
        int  k;
        static int b;
        friend int h() { return  -1; }
        int f() { return 0; }
        static int g() { return 1; }
        A() { }
} x;
```

x的存贮空间

int i
int k

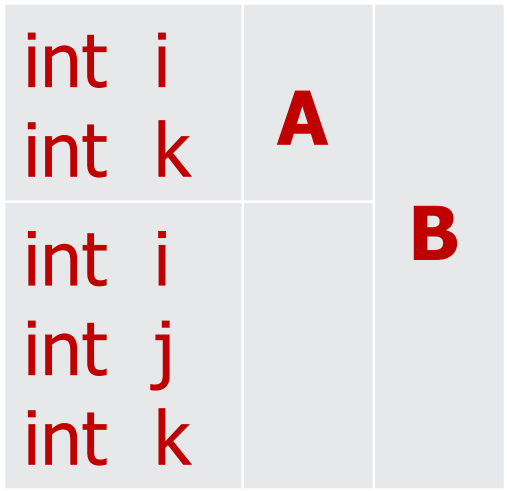
Question: 能否将h()写成 { return k; } ?

1.2 无虚函数的派生类存贮空间

派生类对象需要把基类空间包含进去

```
class A {
        static long a;
        int  i;
public:
        int  k;
        static int  b;
        int f()  { return 0; }
        static int g() { return 1; }
};
class B : A {
     int  i, j, k;
     static  int  m;
} x;
```

x的存贮空间



多继承和虚基类怎么处理？

如何访问x中A的数据成员？

1.3 访问对象(无虚函数)的数据成员


```
class A {
   static long a;
   int  i;
public:
   float  k;
   static int b;
   int f() { return 0; }
   static int g()
   { return 1; }
   A(int x,int y)
   { i = x; k = y+0.5f; }
};
```

```
class B : A {
   int  i, j;
public:
   int  k;
   static int m;
   B(int x, int y, int z): A(x-1,y-1)
   { i = x; j = y; k = z; }
};
void main()
{
   B  b(1,2,3);
   int    *p1 = (int *)&b;
   float *p2 = (float *)(p1+1);
   int    *p3 = (int *)(p2+1);
   cout << p1[0] << p2[0] <<
   p3[0] << p3[1] << p3[2];
}
```

Results: 0 1.5 1 2 3

1.4 有虚函数的简单类存贮空间

偏移 00 - 03：虚函数表VFT的地址
偏移04开始：非static的数据成员

```
class A {
    int  i, j;
    static  int  x, y;
    virtual void f( ) {  cout << "f()"; }
public:
    int  k;
    virtual void g() { cout << "g()"; }
    int h() { return i + k; }
};
```

//假设 sizeof(int) = 2

类A的存贮空间

偏移	内容
00-03h	VFT地址
03-04h	int i
05-06h	int j
07-08h	int k

VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

1.5 有虚函数的派生类存贮空间 (1)

(1) 基类没有虚函数(派生类有)

```
class A {
    int  i;
    static  int  x, y;
public:
    int  k;
    int  h() { return i + k; }
};
class B: public A {
    int  i;
public:
    int  k;
    int  h() { return i + k; }
    virtual void f()  { cout << "B::f()"; }
    virtual void g() { cout << "B::g()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-01h	int A::i
02-03h	int A::k
04-07h	B的VFT地址
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	f()的入口地址
04-07h	g()的入口地址

1.5 有虚函数的派生类存贮空间 (2)

(2) 基类有虚函数

```
class A {
    int  i;
    static  int  x, y;
public:
    int  k;
    virtual void f()  { cout << "A::f()"; }
    virtual void g() { cout << "A::g()"; }
};
class B: public A {
    int  i;
public:
    int  k;
    int  h()  { return i + k; }
    virtual void f()  { cout << "B::f()"; }
    virtual void k() { cout << "B::k()"; }
};
```

//假设 sizeof(int) = 2

类B的存贮空间

偏移	内容
00-04h	B的VFT地址
00-01h	int A::i
02-03h	int A::k
08-09h	int B::i
0A-0Bh	int B::k

B的VFT

偏移	内容
00-03h	B::f()的入口地址
04-07h	A::g()的入口地址
08-0Bh	B::k()的入口地址

1.5 有虚函数的派生类存贮空间 (3)

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。

1.6 利用存贮空间访问变量和虚函数


```
class A {
    int  i;
public:
    virtual void f()
    { cout << "A::f \n"; }
    virtual void g()
    { cout << "A::g \n"; }
    A(int x) { i = x; }
};
class B : public A {
    int  i, j;
public:
    virtual void h()
    { cout << "B::h \n"; }
    void g()
    { cout << "B::g \n"; }
    B(int x,int y): A(x-1)
    { i = x; j = y; }
};
```

```
void main()
{
    B  b(1,2);
    int *v = (int *) ( (char *)&b +
                sizeof(void *) );
    printf("A::i=%d, B::i=%d,
    B::j=%d \n", v[0],v[1],v[2]);
    void **vfb = *(void ***)&b;
    for(int i = 0; i < 3; i++)
    {
        void (*f)() = (void (*)())vfb[i];
        f();
    }
}
```

A::i=0, B::i=1, B::j=2

A::f B::g B::h

能否直接取虚函数的地址，如 &b.f ？

2. 类成员的访问


```
class A {
        static long a;
        int  i;
protected:
        int  j;
public:
        int  k;
        static int b;
        int f() { return 0; }
        A(int a) { }
};

class B {
public:
        int  j;
        B(int b) { }
};
```

```
class C : protected  A, B {
        int  i;
protected:
        int  j;
public:
        C   c;   //错误, 递归定义
        int  k;
        int f() { return 0; }
        C() :  B(2), A(1) { }
} x;
```

对于不同的继承属性，如何访问对象x中所有成员（包括A和B的成员和C的新成员）？

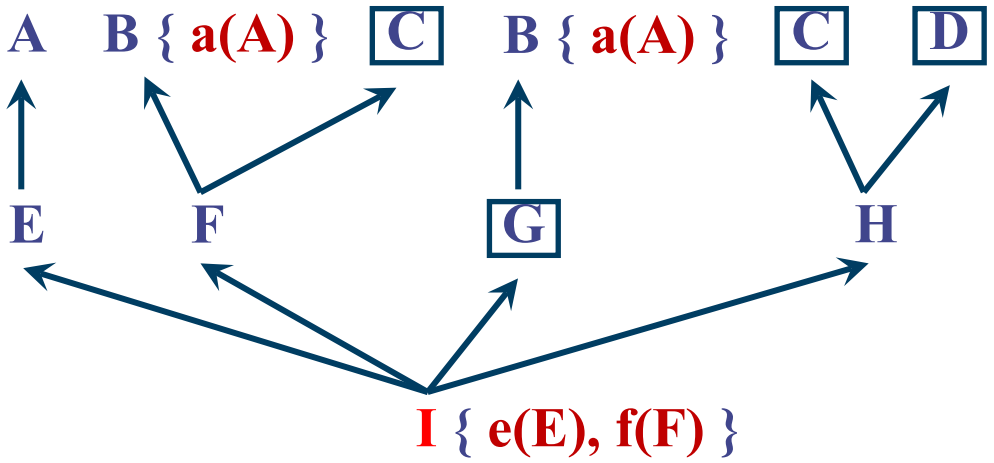
3. 构造、析构及初始化 (1)

- 析构和构造的顺序相反，派生类对象的构造顺序：
  - 按自左至右、自下而上地构造倒派生树中所有虚基类
  - 按定义顺序构造派生类的所有直接基类
  - 按定义顺序构造（初始化）派生类的所有数据成员，包括对象成员、const 成员和引用成员
  - 执行派生类自身的构造函数体
- 如果构造中，虚基类、基类、对象成员、const 及引用成员又是派生类对象，则派生类对象重复上述构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。
- 由派生类（根）、基类和虚基类构成一个派生树的节点，而对象成员将成为一棵新派生树的根。

3. 构造、析构及初始化 (2)

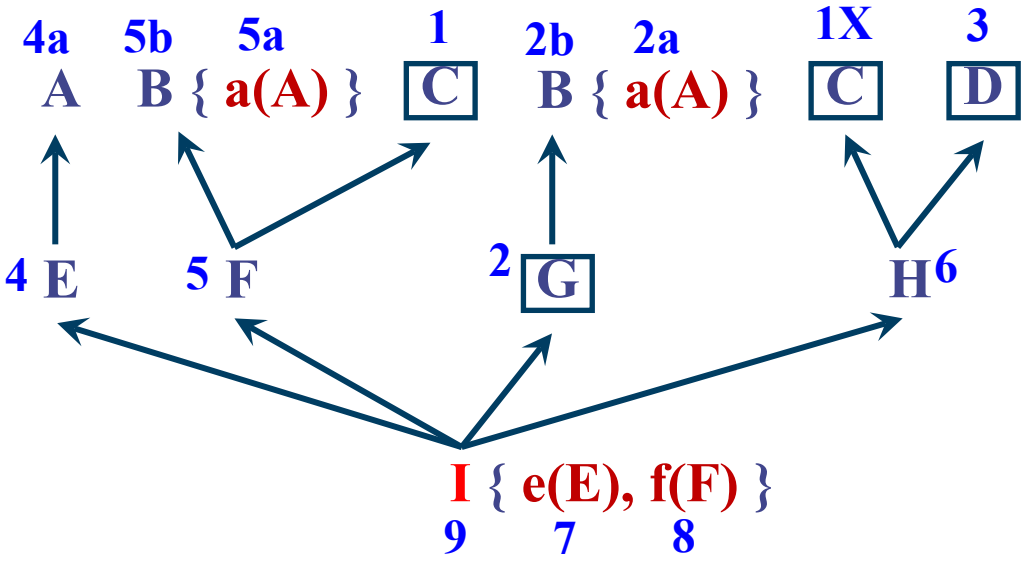

```
struct  A {  A( ) { cout<<'A'; } };
struct  B {  const  A  a;
             B( ) { cout<<'B'; }
           };
struct  C { C( ) { cout<<'C'; } };
struct  D { D( ) { cout<<'D'; } };
struct  E:  A { E( ) { cout<<'E'; } };
struct  F:  B, virtual C { F( ) { cout<<'F'; } };
struct  G:  B  { G( ) { cout<<'G'; } };
struct  H:  virtual C, virtual D { H( ) { cout<<'H'; } };
struct   I:  E, F, virtual G, H {
             E   e;
             F   f;
             I( ) { cout<<'I'; }
          };
void main(void) { I  i; }
```

3. 构造、析构及初始化 (3)



例9.7六棵派生树 (根红色)

3. 构造、析构及初始化 (4)



例9.7六棵派生树(根红色)，输出：CABGDAEABFHAECABFI

3. 构造、析构及初始化 (5)


```
struct  A {
    A(int a) { cout << a; }
} a(0);
struct  B : virtual A {
    B(int b) : A(1) { }
} b(1);
struct  C : public B {
    C( ): B(3), A(2) { }
} c;  //能否去掉红色的A(2)?
int main( ) { }   //输出: 012
```

如果虚基类没有无参的构造函数，则需显式指明虚基类的构造方式。

```
struct  A {
    A( ) { cout << "A"; }
    A(int a) { cout << a; }
};
struct  B : virtual A {
    B(int b) { }  //没有A的构造?
    A  a = A(1);
} b(1);
struct  C : public B {
    C( ): B(3), A(2) { }
} c;  //能否去掉红色的A(2)?
int main( ) { }   //输出???
                  //输出: A121
```

4. 虚函数 (1)


```
struct  A {
    int  a;
    void f() { }
};
struct  B : public A {
    int  a;
    void f() { }
};
struct  C : public B {
    int  a;
    void f() { }
};
```

```
void main()
{
    C  c;
    A  *p1 = &c;
    B  *p2 = &c;
    p1->f();              //???  A::f()
    p2->f();              //???  B::f()
    int x = p1->a;        //???  C::A::a
    int y = p2->a;        //???  C::A::a
    A  a = *(A *)&c;
    a.f();                //???  A::f()
    x = a.a;              //???  new C::A::a
}
```

C有3个a !!!

对于不同的继承属性，
main()程序有语法错误否?

4. 虚函数 (2)


```
struct  A {
    virtual  void f() { }
    void g() { }
};

struct  B : public A {
    void f() { }
    virtual void g() { }
};

struct  C : public B {
    void f() { }
    void g() { }
};
```

```
void main()
{
    C  c;
    A  *p1 = &c;
    B  *p2 = &c;
    p1->f();                //???
    p1->g();                //???
    p2->f();                //???
    p2->g();                //???
    A  a = *(A *)&c;
    a.f();  a.g(); //??, 若A::f()不是虚函数?
    c.f();  c.g();          //???
}
```

p1: C::f() , A::g()

p2: C::f() , C::g()

4. 虚函数 (3)


```
class A {
public:
    virtual void f() { cout << "A::f() \n"; }
    void g() { cout << "A::g() \n"; }
};
class B : public A {
public:
    void f()  { cout << "B::f() \n"; }
    virtual void g() { cout << "B::g() \n"; }
    void h() { cout << "B::h() \n"; }
} b;
class C : public B {
public:
    void g() { cout << "C::g() \n"; }
} c;
```

```
void main()
{
    A  *p = &b;
    p->f();              //B::f()
    p->g();              //A::g()
    //p->h();  //b.h();
    /**/
    p = &c;
    p->f();              //B::f()
    p->g();              //A::g()
    /**/
    B  *q = &c;
    q->f();              //B::f()
    q->g();              //C::g()
    /**/
    B  x = *(B *)&c;
    x.f();  x.g();   //??
}
```

4. 虚函数 (4)

深入理解虚函数：用派生类指针指向基类对象

```
class A {
    int a;
public:
    virtual void f()  { cout << "A::f"; }
    void g()          { cout << "A::g"; }
};
class B : public A {
    int b;
public:
    void  f()  { cout << "B::f"; }
    void  g() { cout << "B::g"; }
};
void main()
{
    A  a;
    B  *p = (B *)&a;
    p->f();              //A::f,  why?
    p->g();              //B::g
}
```

```
class A {
public:
    virtual void f()  { cout << "A::f"; }
    void g()          { cout << "A::g"; }
};
class B : public A {
    long  b
public:
    void  f()         { cout << "B::f"; }
    virtual void  g() { cout << "B::g"; }
    void  h()         { b = 0;  }
};
void main()
{
    A  a;
    B  *p = (B *)&a;
    p->f();      //A::f
    p->g();      //崩溃, g()的地址无效
    p->h();      //崩溃, 无效的B::b
}
```

5. new 和 delete

- new: (1) 申请对象所需内存空间
  (2) 调用构造函数构造对象
- delete: (1)调用析构函数销毁对象
  (2)释放对象所需内存空间

```
A *p = new  A;              delete   p;  //A有无参的构造函数
A *p = new  B(…, …);        delete   p;  // B只有带参数的构造函数
A *p = new  A[10];          delete   []  p;
A *p = new  B[10](…);       delete   []  p;  //error
int  *p = new int[10];      delete [] p;  delete p;   free(p)
```

6. this、const、volatile、static

- this: 每个普通成员函数的第1个参数（被隐含）
- const、volatile: 即可用于修饰数据成员，又可修饰函数成员
- static: 不能放到函数声明的右边，static限定的成员具有全局性质，类内声明的static数据成员只能进行全局初始化，访问类内静态成员要符合其属性规范

```
class A {
        static  int  x;
public:
        static  int  y;
        int  z;
};
```

```
int  A::x = 1;
int  A::y = 2;
void main() {
    A    a;
    int  i = A::x + A::y;  //error?
    int  j= a.x + a.y;     //error?
}
```

7. 函数中类对象参数的传递(1)

主程序先在堆栈中申请对象所需的缓冲区并调用构造函数构造一个对象，子程序使用完该对象后在返回时调用析构函数销毁堆栈中的对象。如果类定义了如A(A &a)形式的构造函数，则使用这个构造函数构造堆栈中的对象；否则使用系统缺省的构造函数（直接拷贝）。

```
struct A {
    int  z;
    char *p;
    A() { p=new char [10]; }
   ~A() { if(p) { delete p; p=0; } }
};
```

```
void  f(A a) { }

void main() {
    A  a;
    f(a);
    strcpy(a.p, "ABC");  //???
}
```

7. 函数中类对象参数的传递(2)


```
class A {
   int  i;
public:
   A(int x) {  i = x;
               cout << " Ad" << i;
            }
   A(const A &a) { i = a.i;
                   cout << " Aa" << i;
                 }
   ~A() { cout << " ~A" << i; }
   A add(A a) {  a = i + a.i;
                 return a;
              }
};
void main()
{
   A  a(1);
   a = a.add(a);
}
```

Ad1 Aa1 Ad2 ~A2 Aa2 ~A2 ~A2 ~A2

```
class A {
   int  i;
public:
   A(int x) {  i = x;
               cout << " Ad" << i;
            }
   A(const A &a) { i = a.i;
                   cout << " Aa" << i;
                 }
   ~A() { cout << " ~A" << i; }
   A add(A &a) {  a = i + a.i;
                  return a;
               }
};
void main()
{
   A  a(1);
   a = a.add(a);
}
```

Ad1 Ad2 ~A2 Aa2 ~A2 ~A2

8. 函运算符重载 (1)

- 不能重载（5个）：sizeof　.　.*　::　?:
- 只能重载为普通函数成员（4个）：=　->　()　[]
- 不能重载为普通函数成员的：new　delete
- 其他运算符：只能重载为普通函数成员和普通函数
- 运算符重载必须至少有一个参数是对象类型（A、const A、A &、volatile A等）
- 若运算符为左值运算符，则重载后运算符函数最好返回非只读引用类型（左值）
- 前置++和--重载为单目，后置++和--重载为双目

8. 函运算符重载 (2)


```
class A;
int operator=(int, A &);     //错误, 不能重载为普通函数
A &operator +=(A *p, A a[3]);      //p和a参数不代表对象
class A {
        friend int operator=(int, A &); //错误, 不存在operator=
        static  int operator()(A &, int); //错误, 不能为静态成员
        static  int operator+(A &, int);  //错误, 不能为静态成员
        friend A &operator+= (A &, A &);  //正确
        A &operator++( );    //隐含参数this代表一个对象
};
```

8. 函运算符重载 (3)


```
class A {
    int   a;
    friend A &operator-- (A &x) { x.a--; return x; } //自动内联, 返回左值
    friend A operator-- (A &, int);   //后置运算, 返回右值
public:
    A &operator++( ) { a++;  return *this; }   //单目, 前置运算
    A operator++(int) { return  A(a++); }      //双目, 后置运算
    A(int x) { a=x; }
}; //A m(3); (--m)--可以; 因为--m左值, 其后--要求左值操作数
A operator--(A &x, int) {  //x左值引用, 实参被修改
     return  A(x.a--);          //先取x.a返回A(x.a)右值, 再x.a--
}  //A m(3); (m--)--不可; 因为m--右值, 其后--要求左值操作数
```

9. 习题 (2.1)


```
char   c, *pc;
const  char    cc='a';
const  char    *pcc;
char   *const cpc=&c;
const  char   *const cpcc=&cc;
char   *const *pcpc;
```

下面的赋值哪些是合法的？

```
char   c, *pc;
(1) c = cc;          (10) *pc="ABCD"[2];
(2) cc=c;            (11) cc='a';
(3) pcc=&c;          (12) *cpc=*pc;
(4) pcc=&cc;         (13) pc=*pcpc;
(5) pc=&c;           (14) **pcpc=*pc;
(6) pc=&cc,          (15) *pc=**pcpc;
(7) pc=pcc;          (16) *pcc='b';
(8) pc=cpcc;         (17) *pcpc='c';
(9) cpc=pc;          (18) *cpcc='d';
(19) *pcpc = &c;  /???
```

9. 习题 (2.4)


```
int &f() {
    int  i = 10;
    int  &j = i;
    return  j;
}
int g() {
    int  j = 20;
    return  j;
}
void main() {
    int  &r = f();                          //r=10
    int  j  = g();                          //r=20, j=20
    printf("r=%d j=%d\n",                   //r=6, j=20
              r,j,1,2,3,4,5,6,7,8);
    r = f();                                //r=10
    j = g();                                //r=20, j=20
    printf("r=%d j=%d\n",r,j);              //r=20, j=20
}
```

9. 习题 (6.5)


```
class A {
    int  a1, a2;
protected:
    int  a3, a4;
public:
    int  a5, a6;
    A(int);
    ~A(void);
};
```

```
class B: A {
    int  b1, b2;
protected:
    A::a3;
    int  b3, b4;
public:
    A::a6;
    int  b5, b6;
    B(int);
    ~B(void);
};
```

```
struct C: B {
    A::a5;
    int  c1, c2;
protected:
    A::a4;
    int  c3, c4;
public:
    int  c5, c6;
    C(int);
    ~C(void);
};
```

A:
private: a1, a2
protected: a3, a4
public: a5, a6, ~A()

B:
no_access: A::a1,A::a2
private: b1,b2,
A::(a4,a5,~A())
protected: b3,b4, A::a3
public: b5,b6,~B(), A::a6

C:
no_access: A::(a1,a2),
B::(b1,b2)
public: c1,c2,c5,c6,~C(),
B::(b5,b6,~B()) , A::(a5,a6)
protected: c3,c4, B::(b3,b4),
A::(a3,a4)

9. 习题 (7.3)

指出程序的错误之处及其原因：

```
class A {
    int a;
    virtual int f();
    virtual int g()=0;
public:
    virtual A();
} a;

class B: A {
    long f();
    int  g(int);
} b;

A *p = new A;
B *q = new B;
int f(A, B);
A  g(B &);
int h(B *);
```

virtual A() => A(), 构造对象是确定的,不需多态性

A是抽象类, 不能定义对象a

long f() => int f()

B是抽象类, 不能定义对象b

A是抽象类, 不能申请对象

B是抽象类, 不能申请对象

A、B都是抽象类, 参数不能使A、B的对象

不能返回A的对象, 因为A是抽象类

正确

9. 习题 (8.2)


```
class A {
    int  a;
protected:
    int  b, c;
public:
    int  d, e;
    ~A( );
};
```

A:
pv: a
ptd: b, c
pub: d, e, ~A

```
class B: protected A {
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
    A::d;
};
```

B:
xx: A::a
pv: a
ptd: b, f
  A::(b,c,e,~A)
pub: e, g, A::d

```
class  C: A {
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
    A::d;
};
```

C:
xx: A::a
pv: a
  A::(b,c,e,~A)
ptd: b, f
pub: e, g, A::d

```
struct  D: B, C {
    int  a;
protected:
    int  b, f;
public:
    int  e, g;
};
```

D:
xx: B:(a, A::a)
  C:(a, A:(a,b,c,e,~A)
ptd: b, f
    C::(b,f), B::(b,f)
    A::(b,c,e,~A)
pub: a, e, g
  C::(e, g, A::d)
  B::(e, g, A::d)

9. 习题 (9.2-1)


```
int  x = m;  //m=学号
int  y = ::x+5;
struct A {
   int  x;
   static  int  y;
public:
   A &operator += (A &a)
   { x+=a.x; y+=a.y;
     return *this;  }
   operator int() { return x+y; };
   A(int x=::x+1, int y=::y+11)
   { A::x=x; A::y=y; }
};
int  A::y=20;
```

```
void main(void) {          ::x=m,  ::y=m+5
   A    a(2,5);            a.x=2,  A::y=5
   A    b(6);              b.x=6,  A::y=m+16
   A    c;                 c.x=m+1, A::y=m+16
   int   i;
   int  &j = i;
   int  *p = &A::y;
   i = b.y;                i=A::y=m+16
   j = b.x;                i=j=b.x=6
   i = *p;                 i=A::y=m+16
   j = c;                  i=j=c.x+A::y=2m+17
   i = a + c;              i=3m+35
   i = b += c;             i=3m+39
   i = ((a+=c)=b)+9;       i=5m+80
}
```

9. 习题 (9.2-2)

i = a+c;

i = (int)a+(int)c
  = (a.x+A::y)+(c.x+A::y)
  = (2+m+16) + (m+1+m+16)
  = 3m + 35

i = b += c;

b+=c: b.x=b.x+c.x=6+m+1=m+7
        A::y=A::y+A::y=2m+32;
i = (b+=c) ⇔ i=b=b.x+A::y=m+7+2m+32
  = 3m + 39

i = ((a +=c)=b)+9;

a+=c: a.x=a.x+c.x=2+m+1=m+3
        A::y=A::y+A::y=2*(2m+32)=4m+64;
(a+=c)=b ⇔ a.x=b.x=m+7, A::y=A::y=4m+64
i=((a+=c)=b)+9 ⇔ i=a+9 = (a.x+A::y)+9 = (m+7+4m+64)+9
 =5m+80