Volume 00. 面向考试

1. 数据类型会溢出

2. 转义字符

'\t' '\n' '\r' '\0' 都是合法的字符。

注意无特殊表示，\后的数字是**八进制。**

因此'\123''\256''\300'都合法，但 '\377'不合法，因为 ASCII 码最大是 255.

'\x23'表示 ASCII 码是十六进制下 23（十进制下 35）的字符。

'\'不合法，'\\'表示\本身，是合法的。

3. 输出浮点数：去末尾 0（限 C++）

建议看一看 printf 格式化输出表和<cmanip>的操作符。但考了顺便问候一下出卷人全家。

%d 整数

%5d 整数，5 位场宽（如果整数位数小于 5，用空格补足五位）

%5.2f 浮点数，5 位场宽，两位小数

%.2f 浮点数，两位小数

4. 常量表示

0 开始 → 八进制	0123 合法，0288 不合法（288 不可能是八进制数）
0x 开始→ 十六进制	0x123 合法，0xff 合法，0x21g 不合法

5. 变量名

不能以数字开头，不能包含运算符，当然_可以随便出现

Volume 01.面向过程（面向结构）

【程序基础】

1. 左加与右加

i++（表达式的值是 i，传递值并完成相关操作后 i=i+1）

++i（先 i=i+1，再传递 i 的值，表达式的值是加完 1 后的 i）

从左到右即可

2. 强制类型转换

double (x/y)　先做 x/y 再转 double

(double) x/y　先把 x 转 double→把 y 转 double 再除

(double)(x/y)　先做 x/y 再转 double

3. ==!!!　“=”是赋值符

4.【短路求值 ShortCut】

&&优化（一个 F 直接退出判断）

|| 优化　（一个 T 直接退出判断）

[e.g.]

```
int p = 0, q = 3;
if ((!p)&&(++q))  q++;    //q=4，(++q)直接被无视。
```

~~5.（不会考的东西）　仅限 x 个 expr 为真　Sigma(expr)==x~~

~~[e.g.] if ((x==9) + (y!=0) + (z!=3) == 2)　count++;　//三个条件中两个成立即可~~

~~一个肯定不会考的东西：y!=0 不能改成(!y), bool(!y)也不行，因为这两个语句本质在做位运算，返回值未必是 0/1。~~

if 语句判断的本质是：零（False）或非零（True）

6.　浮点数不能用==比较相等　(使用 a-b<EPS 判断该精度下相等)

7. do-while

do-while<expr>　和　[PASCAL] repeat until<expr>中 expr 的要求真值正好相反。

do-while　直到 expr=F 停止；

~~[PASCAL] repeat..until　直到 expr=T 停止。~~

do-while()　至少会做一次循环中的操作。

~~8.　（不太可能考）逗号运算符，本质也是<expr>（一个表达式）~~

~~expr1, expr2, expr3, ..., expr n~~

~~从左到右依次处理，整个表达式的值为 expr n 的值。~~

9. for()　的　expr3　实际上就是在“}”后面加个语句。

10. break 的本质

跳出当前 block（整个{}，跳到}后，跳出使用该 Block 的内容（for/while/switch/do-while））

continue

仅限循环。跳到 } 然后继续循环。

11. 进制输入：cin>>oct>>x>>x2>>hex>>y>>dec>>z>>bin>>a;

x、x2 输入形式八进制，存储形式十进制

y 输入形式十六进制，存储形式十进制

z 输入形式十进制，存储形式十进制

a 输入形式二进制，存储形式十进制

[oct] [hex] [dec] [bin] 的作用域一直持续到下一个[oct] [hex] [dec] [bin]出现。

12. 系统库文件（库文件→ 【结构体/模块化设计】6）

C++新增： <iostream> ~~<crandom>（限 C++11）~~ 等

C：<cstdlib> <ctime> <cstdio>等

13. 名字空间 using namespace std 或 mine 或 whatshisname; 表示之后的变量、函数、特定操作符等都是该名字空间下定义的那些。

也可以不用这句话，然后用[namespace]**::**<varivable/functions/operator/……>。

[e.g.] std**::**cout << mine**::**a << std**::**a << whatshisname**::**a << "\n";

14. 比较简单且不怎么随机的【随机数】生成方式

<cstdlib>

只使用 rand()，每次运行程序时程序中所有随机数都是同一个。

srand(seed)，其中 seed 为 unsigned int.

一般 IDE 默认的 seed 为 1

seed 一般选取 unsigned int(time(NULL))

<ctime>

clock();

time(); //两者都能获取时间，但精度（单位）不同，time()精度更高

15. 以下程序段是否会陷入死循环?

```
int k=1000;
do
{  ++k; }
while (k>=1000);
```

不会。k 会加到数据溢出然后变成-2147483647，于是循环终止。

【字符串/字符数组】

1. 字符串结尾自动加"\0". **实际存储多一位**。

2.

a[]="" <==> a[1]='\0'。

'a' 一个字符

"a" 数组，{'a','\0'}

字符串的有效长度不包括'\0'（strlen()）

3. cin.getline

使用方式：cin.getline(str,len,ch);

ch 可以省略，默认'\n''\r'和 EOF。

最多读入(len-1)位（需要最后加'\0'）

对越界情况不同 IDE 处理不同。

4.【流读取】

cin>> 一般会跳过**空白字符（white space）,i.e** ' '（空格）,'\t','\n','\r'.

cin.eof （end of file）

直到读取到文件结束符（EOF）为止。即[PASCAL]EOF

※实际是在判断输入是否正常

可以用((ch=cin.get())!=EOF)判断是否到达文件尾

不同系统的 EOF：

^Z (Ctrl+Z) ~ Windows, MS-DOS

^D (Ctrl+D) ~ Linux, Unix

~~5. C++11 后推荐初始化写法：~~

~~char s[100]{};~~

~~<==> char s[100]={'\0'};~~

【函数】

1. 函数的形式参数每一个都需要声明类型，用","间隔，**不能省略类型**。
2. C/C++似乎不喜欢全局变量，所以尽量不用就好
3. **变量作用域→Block(块)**

```
int main()
{
   int a;
   {
       int a;
       a = 1;
          //屏蔽了外面那个块的 a，只对当前块中的 a 操作。编译器的处理：给 a 一个内部命名，类似函数名重载。
          （局部变量/内 Block 变量屏蔽同名全局变量/外 Block 变量）
   }
}
```

4. 局部变量与全局变量同名时访问**同名的全局变量** "**:: <变量名>**" **(只代表同名全局变量)**

[e.g.]

```
int a=0;            //假设 IDE 把它处理成 a_global（内部命名）
int f()
{
    int a = 2;      //假设 IDE 把它处理成 a_f
    a++;            // a_f++;
    ::a = ::a + a;  // a_global = a_global+a;
    {
        int a = -121;          //假设 IDE 把它处理成 a_f_
        ::a++;                 //a_global++;
    }
}
```

如果没有同名全局变量，使用"::a"会报错（"::a" is not declared）。

5. extern：

① 捆绑后面的全局变量;

② 其他源文件里的全局变量

6. 变量使用空间释放顺序：先定义的后消失（栈）。

（1）各函数内局部变量

（2）函数静态变量

（3）全局变量

7. static 全局变量：仅限本文件使用。

8. C++ 形式参数为数组时，**实际传递的是数组的地址（数组名指针）**，因此在函数中对数组形参进行操作时会影响到实际的数组。

换言之：不存在"数组形参"这种形参。其实都是实际参数。

9. 【默认参数】 void print(int value,int base=10)

此时 print(20) 等价于 print(20,10)。

编译机制：遇到一个参数省略自动省略所有后面的参数并取默认值。

所以默认参数必须放最后。

默认参数需要在声明中就写明，否则编译出错。

10. 内联函数：inline 类型 名称（形参）<语句> **（必须声明即定义）**

把后面的语句直接"粘贴"到程序中调用它的地方，生成程序代码副本。(实现方式：编译器将其加入符号表，用代码代替函数调用)

必须在调用前定义。

不适合 inline：函数内有循环，递归等。

11.【重载函数】

C++认为形参表不同（个数或类型不同）而名称相同的函数是两个函数。

编译器对此的实现：以内部名称代替函数名。

重载函数的编译处理

int max(int,int) → max_int_int_int

double max(double,double) → max_double_double_double

12.【函数模板】

template <模板形式参数表>

（template <class/typename Type>）

调用前定义。相当于定义了函数类型。

显式模板形参：template <**class** T1, class T2, class T3>

T3 calc(T1 x, T2 y);

调用：calc<char,int,int>(a,b) 得到类型为 char。

※后来面向对象用 class 表示“类”，因此新 IDE 下保留字替换为“typename”。当然 class 在此处的含义仍得以保留。

编译原理：根据参数确定模板参数的类型，再生成对应函数。

【指针/引用/动态数组】

1. 指针类型：存储地址，四字节十六进制

0x0061fe44

若输出，**省略前导零**，即 0x61fe44.

2. 字符指针与常量指针：

```
char *q;                    //定义指针时最好先赋值 NULL;
const char *p="abcde"; //ok
char *r="abcde";    //在比较严格的 IDE 下会报错
*p = “c”;              //error
p = q;                  //ok
q = new char[10];
strcpy(q,"abcde");
cout << q << endl;        //abcde
cout << r << endl;        //abcde
cout << *q <<endl;        //a（q 存的是'a'的地址）
cout << *r << endl;       //a
cout << (void*)q << endl;   //地址。
q++;
r++;                    //是常量指针但不是指针常量，所以指针本身还是可以改变的
cout << q << endl;       //bcde
cout << r << endl;       //bcde
cout << *q <<endl;        //b
cout << *r << endl;       //b
```

3. 为什么定义指针时需要声明其指向地址所存储的数据的类型？既然指针本身只占一个 4 字节单元（存储地址）（以 32 位系统为例），只要一个“pointer”（指针）类型不就够了吗……为什么还要分 int*， char*，long*，char这么多种指针类型？**

我们知道指针有这样的操作：p++，p--，比如 p 指向一个数组的某个元素的时候就可以用这种操作。它的作用是前移/后移一个单元格，本质是**前移/后移 sizeof(类型)个字节**！

因此指针类型实际上保存了移动的字节数（即 sizeof（类型)），所以指针类型包括了其指向地址所存储的数据类型，

4. 实际程序调用的存储空间：（C++）

↓ 整个内存

[OS] ——— 操作系统用的部分，我们的程序用不了

[DataSec] ——— 常量区 Data Section（Java 等语言称之为常量池），**程序结束后自动释放。**

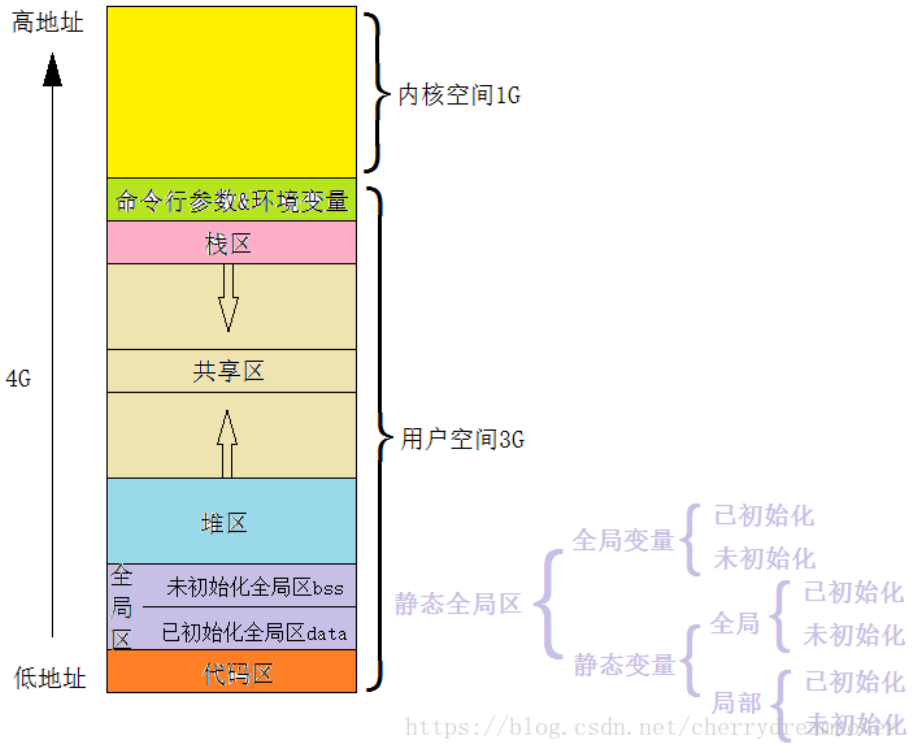
[Stack（栈）] ——— 局部变量

[Heap（堆）] ——— malloc 分配的空间，用 free 释放空间

[※自由存储区] ——— C++的动态分配区，内存较大，指针动态开辟的空间（用 delete 释放）。**本质就是 Heap 和 Static 的一部分**，并不是单独分配的一块区域。一般在 Heap 或 Static 区域里开辟新空间。

[Static] ——— 静态区/全局区，静态局部变量和全局变量

[Programm] ——— 存代码（二进制存储）



5. 数组名指针

int a[1000]

a 保存的是 a[]的地址，**可以看成一个“常量指针”（不可改变，但可以知道地址）。**

如：

int *p;

p=a; 等价 p=&a[0];

p=a+1; 等价 p=&a[0+1];

实际上&a[1]-&a[0]表示该数组一个“单元”占用内存大小。

对比：**cout << (p-a);　//输出 1 而不是 4（32 位系统下一个 int 单元的内存大小）**

理解方式：(p-a)和 a,p 同类型，　p-a = (a+1)-a = 1。

6. 指针返回值的函数→ 返回的是一个“变量”（实际是变量的地址），可用于找到某个变量对其直接进行操作。

e.g.

```
int* max(int* a, int* b)  {    if (*a>*b) return a; else return b;  }
int main()  {
   ……
   *(max(a,b)) ++;          //若 a<b  则 b++; 若 a>b  则  a++;
}
```

7.【引用】（限 C++，比指针好用）

```
int i;
```

int &j=i;　必须定义即赋值，**不能重新赋值**，因为两者保存于同一个内存单元。

引用本质：取别名，捆绑在一起。

可以嵌套引用，即 int &j=i; int&k=j; int &l=k;

本质：看到 int i;　　开辟空间：　i【　】

然后遇到 int& j=i;　　我也叫 j

遇到 int& k = j;　　我也叫 k

引用不会开辟新的空间　→ 比指针好用 1

8. 指向数组/指针的引用：把数组名和指针名改成(&i)。

e.g. int a[3]={};

```
   int (&b)[3]=a;          //&b[3]是错误的，因为会翻译成&(b[3]). 优先级问题
   int *p=a+1;
   int *(&x)=p;       //或 int *&x=p;（两者都是定义指针的引用）
```

9.【引用传递】

函数参数能用引用不用指针（而且还好看，普通函数的参数上加个&就完事了）。

→ 比指针好用 2

e.g.

```
void f(*a,*b) { *a,*b }        f(&x,&y)
void f(&a,&b) { a,b }          f(x,y)
```

形式参数与实参共用内存空间，交换即实参交换，形参的改变即实参改变。

10. 返回引用的函数：也加个&就完事，功能和返回指针的函数一样。

```
int &max(int &a, int &b, int &c)
{
   if (a>b)
   {
      if (a>c) return a; else return c;
```

```
    }
    if (b>c) return b; else return c;
}
```

返回的是最大的那个变量本身。

此时 max(a,b,c)++; max(a,b,c)=3; 这样的语句是合法的。

11.【最小特权】

能加 const 就给 const（只读取不改变不写入）

const 后面的变成常量，拒绝写入。

e.g.

int *const ptr; → ptr 拒绝写入，可以改*ptr（ptr 指向的空间内容）

const int *ptr; → *ptr 拒绝写入（ptr 指向的空间内容拒绝写入），可以改 ptr 本身.

e.g. const int a[]

const int *const ptr; → ptr 动不了了

~~12. main() 参数~~

~~argc 参数个数~~

~~argv[] 一个指向字符串的指针数组。~~

~~运行时要求输入，不需要 cin。~~

~~一般形式：main(int argc, char **argv)~~

13.【指针开辟动态数组】

*A 代替一维数组，动态一维数组。

```
int* A = new int[size];
delete []A;
```

**a 代替二维数组，动态二维数组。

```
int** a = new int *[row]
a[i] = new int[col]

delete []a[i];    //删除动态数组
delete []a;

delete p; //删除指针
```

特别注意指针开辟的动态数组赋初值情况

```
int a[10];
int *p = new int[n];
int i;
```

p = a; //错误。p 指向了 a 指向的内存空间（a 对应的数组）。

```
delete []a;
```

//如果之前是 p=a，则出现错误：**“the poiner freed was not allocated.”**（因为 p 指向

了 a，a 不是动态数组，并非由指针开辟）

正确写法：for (i=0; i<n; i++)　　p[i] = a[i];

14. 【指向函数的指针】

函数名即是当前函数的指针，它指向函数体**代码在内存中保存的地址**（保存在 Program 区）。

定义指针类型：
void (*func)()= NULL 或 func;　（func 必须是 void func()）
函数指针数组：
void (*func[5])(int, int, int) = {NULL, func1, func2, func3, func4};
要求 func1, func2, func3, func4 都是 void(int, int, int)类型的函数。

[e.g.]
声明定义部分：
typedef bool (*FPtr)(const int x, const int y)
bool incINT(const int x, const int y) {……};

FPtr cmp;
template <class T>
void Sort(,　,　bool (*cmp)(T,T))

调用部分：
Sort(,　,　incINT);
　　↑使用了（指向）函数（的）指针

【结构体/模块化设计】

1. 类似[PASCAL]record（记录类型）

定义声明：
struct <类型名>
{
　　已定义类型 1　变量 1;
　　已定义类型 2　变量 2;
　　……
　　已定义类型 n　变量 n;
};
注意结尾有冒号（整个 struct 是一句话。）
内容一个个声明类型，不可使用变量表。

2. 函数返回结构体地址的实现方式

函数体里新建结构体指针，然后在函数调用后 **删除指针**（可行）
（因为局部变量会被自动释放，无法返回地址。）

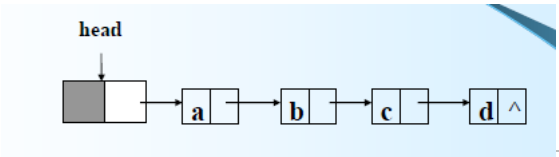
3. pointer being freed was not allocated

被释放的指针并非动态开辟。出现该问题的情况在【指针】12 已阐述。

4. 结构体嵌套 → **链表**

```
struct T1
{
    ...
    T1* next;
};
```

做题的时候画画图就可以了，比如↓



（^表示 NULL）

5. 为了**节省空间**，结构体作为函数参数时**最好使用引用传递**。
   若需要“值传递”，可以 **const <typename> &**x 作为“伪值传递”。（最小特权原则）

6.【模块化设计】

.h 文件（库的接口）

```
#ifndef _name_h              //if not defined （yet）的简写
#define _name_h             //定义宏
//若还没有定义.h 的说明书，则定义该宏
函数原型声明;

符号常量;

自定义类型;

#endif                                  // end of "if(ndef)"
```

.cpp 文件（库中函数的定义）

```
函数原型定义;
```

[e.g.]

random.h

```
#ifndef _random_h
#define _random_h
```

```
void randomseed();
            //注释解释函数用法、作用、注意事项
int randomint(int low, int high);

#endif
```
[EOF]　//这里我用这个表示文件末尾，实际编程并不需要自己加（系统会自己加的）

random.cpp

```
#include "random.h"          //""默认引用的.h 和.cpp 在同一目录
#include <cstdlib>            //<>引用系统路径下的库文件
#include <ctime>

void randomseed()
{       srand(unsigned long long(time(NULL)));       }

int randomint(int low, int high)
{      return( low + rand() % (high-low+1) ) ;   }
```
[EOF]

7. 如何隐藏库的实现细节？→游戏引擎，专利

【.o 文件】库的源文件→二进制代码

所以只要 .h + .a/.lib 文件即可。

静态链接库和动态链接库（实验 T3）

8. 【函数放入结构体】

若函数都围绕结构体展开，可直接放入结构体中。→可以省去结构体参数

放入函数的结构体 ⟷ “类”

Volume 02.面向对象

【类】

1. **【类】**

把函数放进结构体。

class <ClassNAME>

{

private/~~protected~~：私有数据成员；私有成员函数。

//protected 一般不用，后面派生中再讲。

[TypeofFunc] <FuncNAME>(参数表);

public: 公有数据成员；公有成员函数声明 **←外部可调用**

[TypeofFunc] <FuncNAME>(参数表);

};

private 和 public 顺序随意，出现次数随意。

私有：只能在类的函数中对其进行修改，不能在主程序或其他函数中对其调用、修改。（调用（如输出）也不行!!!!!）

2. **成员函数定义**

[TypeofFunc] <ClassNAME>**::**<FuncName>(参数表)

{ };

该 FuncNAME 只定义在<ClassNAME>的空间中

VS. 名称空间 std，mine 等

3. class：未标识自动视作 private。

区别于 struct：**自动视作 pubilc**。

e.g. 120301.dev 中的 DynArr 和 DynArr_test ↓.

【DynArr.h】

```
#ifndef DYNARR_H
#define DYNARR_H

class DynArr
{
    int low;
    int high;
    int* arr = NULL;
  public:
    bool create(int lh, int rh);
    bool insert(int index, int value);
    bool inquiry(int index, int &value);
    void free();
};

struct DynArr_test
```

```
{
        int low;
        int high;
        int* arr = NULL;
  public:
        bool create(int lh, int rh);
        bool insert(int index, int value);
        bool inquiry(int index, int &value);
        void free();
        void abababababa() {    return;};
};

#endif
```

【DynArr.cpp】略

DynArr 由 class 定义，DynArr_test 由 struct 定义。在 low 数据成员的定义前不加 private/public.

main.cpp

```
DynArr a, b;
DynArr_test c;
std::cout << a.low;       //DynArr.h 中报错
std::cout << c.low;       //可以
```

4. 可以函数声明即定义，视作内联函数（可以不加 inline）

类似原本程序在 main 前定义。

5. 类的存储大小?

≥所有变量类型存储空间的和。

函数对整个类的所有变量都一样，因此属于类而不属于每个变量，**只有一份拷贝，**所以并不会占用新的存储空间。

6. 对象：类定义的变量。

指向对象的指针：Rational *p;　p->fun() <==> (*p).fun()

7. 类定义中的 this 指针

实际上 IDE 编译后会在函数参数表**左侧**增加 this 指针

如：**[.h]**

```
class DynArr{
……
void create(int lh, int rh);
};        [.cpp]
void create(int lh, int rh)
{ arr = new int[rh-lh+1]; low=lh; high=rh; }
```

编译后：

```
void create(DynArr* this,int lh, int rh)
{
    this->arr = new int[rh-lh+1];   this->low = lh;    this->high = rh;
};
```

8. 注意对象赋值对象时也会出现指针开辟动态数组赋初值问题

```
DynArr a,b;
*****
a = b;        //则 a.arr 保存了*(b.arr)的地址，两者指向同一个数组！
```

【后续部分引言】

我们想方设法想让所定义新类型的“对象”和系统内置类型的变量使用方式长得差不多，因此有了下面这些内容：

9. 构造函数：以类名称命名的函数，常常用以初始化

想达到的效果：NewType x = 2; 或 AnotherNewType x (2,3);

```
e.g. DynArr(int lh, int rh)
{
    low = lh;   high = rh;
    arr = new int [rh-lh+1];
}
```

DynArr a(2,4)语句 包含以下操作：

①开辟 a.low, a.high, a.arr

②调用构造函数。

int x(10); <=> int x=10;

构造函数依然可用函数的默认值，可以带有缺省值（默认构造函数）

10. 重载构造函数 拷贝构造函数

想达到的效果：NewType y; NewType x=y; 或 NewType x (y);

e.g. **拷贝构造函数**

```
    DynArr(const DynArr &x)
        //类和结构体都推荐常量引用传递，若值传递需要调用拷贝构造函数构造新的空间保存值，在结束后又需要析构函数→费时费空间
    {
        low = x.low;
        high = x.high;
        arr = new int[high-low+1];
        *arr = *(x.arr);
    }
```

调用两种构造函数：

```
DynArr a(20,30);
DynArr b(a);
DynArr c = a;        //与 DynArr c(a)本质相同。
DynArr d;
d = a;               //与 DynArr d(a)不同！
d = DynArr(a);       //ok
```

11. 类的动态开辟 DynArr*p = new DynArr(2,43);

12. 析构函数："~类名"的函数，使用完毕后清理类的函数。

void，无参数，不可重载，不可默认参数
无析构函数时，系统自动生成：将几个定义变量释放。（但是指针对应空间不释放）
~DynArr() { delete []arr; }

注意：
只有（重载）构造函数、析构函数的声明和定义可以省略 void（因为只能是 void），其它的 void 仍然要写！

13. **同类型后定义的先消失（auto/static/全局）**
函数值传递对象：
进入函数体→拷贝构造函数→函数体变量定义、空间释放→析构函数→回到调用该函数体的函数

14. 再看函数原型声明
声明部分用 A(int =0); A(int id=0); A(int ID=0); 三种形式**等价**
（实现部分如下：A(int ID=0) { … }; ）

编译器并不在意声明和定义的变量名是否一样，只在意参数表的类型是否一致。
※类型也包括"const"的声明！将 const <type> 和 <type>视为两个不同类型。
[e.g.]

```
class A
{
    public:
    A(int = 0);
    A(const &A copy);
    ~A();
    private:
    int x;
}
A::A(int ID) { x=ID; cout << x<<"\n"; }
```

15. 常量成员与静态成员：

1）常量对象

class A{ };

const A obj; //只能初始化赋值，其余时候都不能改变内容

const A obj2 (12);

const A obj3 = 2;

obj = obj2 = obj3 = obj; //error

2）常量成员函数：不能修改成员变量的值（否则编译出错）

int getx() **const**; //一般都是：输出某些东西，返回某些值之类的

//定义实现的时候也不能省略 const。

可以重载：int getx();

（const 和无 const 是两个函数）

编译器处理： int getx() const; → getx_const

int getx(); → getx

3）常量数据成员

class A{

const int SIZE; //正确

~~const int SIZE=200; //不可在类定义中初始化！~~

};

初始化列表：DynArr::DynArr(int lh, int rh): low(lh), high(rh) { arr = new int [rh-lh+1]; }

常量数据成员只能用初始化列表赋值。

e.g. A::A(int x): SIZE(x) { };

A1(10); // SIZE=10

A2(2121); //SIZE=2121

4）静态成员变量

一个对所有该类变量都一样的数据 → 最好能独立存储，不用每个变量保存一遍。

引入**静态成员变量**：不需要给每个变量都分配该成员的对应空间，该成员和类的函数一样归类所属。**只有一个拷贝。**

class Account

{ ……

static double rate;

…… };

静态成员需要在使用前初始化，初始化方式如下：

在类的声明中，double Account::rate = 0.05; //类似定义成员函数。

[调用方式] ①Account**::**rate ②Account obj; obj.rate

5）**静态成员函数：**归类所有，为了处理静态成员变量。

普通成员函数：编译中自动加入 this 指针。

静态成员函数：**不作任何处理**。（毕竟静态成员是整个类共用）

[声明方式]

在类里声明：static void SetRate(double x) { rate = x; }

在类外声明：void SetRate(double x) { rate = x; }

[定义方式]

不需要 static，直接像正常函数定义就可以了

因此在静态成员函数里只能出现静态成员变量。

[调用方式] ①Account::SetRate(x); ②Account obj; obj.SetRate(x);

6）静态常量成员：

```
class A{
    static const int SIZE=100;
    ……
};
```

不同于普通常量成员（不同对象有不同普通常量，但普通常量对一个对象始终保持不变（恒常）），静态常量成员对整个类型恒常，对所有对象都一样。

16. 友元

类的私有成员理论上只能通过其成员函数访问。有的时候其他类需要访问当前类的数据。

友元关系需要**显式声明**（“授予”）。→ 不具有对称性和传递性。

友元包括：友元函数（一般函数），友元成员（类的成员函数），友元类（整个类）。

声明：friend +具体声明（函数声明/类声明/成员声明）

可以在类里各个部分（private/public）里声明。**习惯性写在最上面。**

1）友元函数声明：类似函数声明。

[e.g.]

```
class boy { char name[10]; int age; friend void display(const boy &x); public:…… };
void display(const boy &x); { …… x.name …… x.age };    //可行
```

2）友元成员声明：类似成员声明。

[e.g.]

```
class boy;
class girl { char name[10]; int age; public: void display(const boy &x); …… };
void girl::display(const boy &x); { …… x.name …… x.age };    //可行
class boy
    { char name[10]; int age; friend void girl::display(const boy &x); public:…… };
```

3）友元类声明

```
class Y{…};
class X{
    ….
    friend Y;//或 friend class Y
    ……};
```

【运算符重载】

1. 全局函数：非类中定义的函数。

2.【※运算符重载】重载系统运算符

目的：让自己定义的类和系统已有的类高度相似。

e.g. 定义复数类 complex a,b,c; c=a+b;

[限制] 并非所有运算符能重载，不能改变优先级、结合性；不能改变运算符是几元运算符（操作对象个数）

+	-	*[二元]	/	%	^	&[二元]	|	~	=
加法	减法	乘法	整除/除法	取模	异或	位运算与	位运算或	位运算非	赋值

<<	>>	==	<	>	<=	>=	!=	!	&&	||	++	--
左移	右移	相等	小于	大于	小于等于	大于等于	不等于	非	与	或	左/右加	左/右减

<<相当于×2，>>相当于÷2（整除）

*[一元]	&[一元]	+=	-=	*=	/=	%=	^=	&=	|=	>>=	<<=
指针访问	取地址	复合赋值符（运算后赋值）									

,	->	->*	<<[左侧 ostream]	>>[左侧 istream]
逗号运算符	访问指针成员	访问指针的指针成员	输出流	输入流

[]	()	new	delete	new[]	delete[]
下标符	括号	动态开辟空间	释放空间	动态数组开辟	释放动态数组空间

一些运算符：>>=，<<=（左移右移后赋值），^=（异或后赋值）

不能重载的运算符：.[成员关系]　.*[通过指针访问成员，不怎么用的一个运算符]

::[作用域符，优先级最高]　?:[三元运算符]　sizeof

强制类型转换相关（typeid，const_cast，dynamic_cast，static_cast，reinterpret_cast）

3.operator@（重载@这个运算符）两种方式：

[1] 友元重载（全局函数）

```
class Complex
{
    friend Complex operator+(const Complex &x, const Complex &y);
    private: int r; int im;
    public: Complex(int x=0, int y=0){r=x; im=y;}
};
Complex operator+(const Complex &x, const Complex &y)
{
     Complex tmp;
     tmp.r = x.r+y.r;
     tmp.im = x.im+y.im;
}
```

[2] 成员重载：　重载运算符放进类里。注意编译会加 this 指针。

```
class Complex
{
    private: int r; int im;
    public: Complex(int x=0, int y=0){r=x; im=y;}
    Complex operator+(const Complex x) const;
};
```

编译后形式：　Complex operator+(**Complex* this,** const Complex &y);

```
Complex Complex::operator+(const Complex &y)
{
    Complex tmp;
    tmp.r = r+y.r;
    tmp.im = im+y.im;
}

调用：c = a.operator+(b);          //operator 不可省略
```

4. =、[]、()、-> **必须重载为成员函数（对对象本身操作）**。
复合赋值符、++、--最好定义为成员函数。（因为会对对象本身进行操作）
其它两元或多元运算符最好定义为友元全局函数。

5.重载赋值符
重载为成员函数。注意返回值类型是**引用**。
e.g. x=y=z; <==> y=z; x=y; "="希望返回的是 y 而非那个值。**右结合性。**
防止返回对象（返回对象也可以进行连续赋值（常规的情况，如 a = b = c，而不是（a = b）= c））的时候调用拷贝构造函数和析构函数导致不必要的开销，降低赋值运算符的效率。

[重载=函数原型]

```
class X
{   …… public: X &operator=(const X &source)
};

X &X::operator=(const X &x) const
{
    if (this==&x) return *this;          //要防止自己复制自己!
    X tmp;
    其它操作;
    return X;
}
```

6. //为什么可以返回一个局部变量/局部变量的引用?

```
创建局部 X 变量→调用构造函数
消失的时候       →调用析构函数
class A{int ID; public:……}
A::A(int) 创建输出
A::A(const A &x) +10，复制（拷贝）输出
A::operator=(const A &x) +100，赋值输出
A::~A()  析构输出

A f2()
{
   A a(50);
```

```
    return a;
}

int main()
{
    进 f2 输出;
    A a1=f2();          //*//
    出 f2 输出;
    return 0;
};
```

老编译器：

看到//*//，先看看 f2()，进 f2，看见 A a(50);

遇到 return a; 开辟空间［fsadsxa］（给函数返回值用），拷贝构造函数，［fsadsxa］Id=60. 把 a 析构。

回到 main 看看有没有人要用这个东西（a1 = ），a1 要用。把这个空间给 a1。

新编译器：

……遇到 return a; 返回了说明 a 没用了，回到 main 看看有没有人要用这个东西，a1 要用。把 a 的空间给 a1。

```
A a1;
a1 = f2();
拷贝 f2 里的 a。Id=60.
```

不同编译器处理方式不同。

7. 重载下标符

```
int &DynArr::operator[](int index) const        //可不加 const 但建议加
{
    if ((index<low)||(index>high)) return(-2147483648);
    return arr[index-low];
}
```

8. 重载++

注意有 x++和++x，作用不同。重载函数时至少需要有两个函数。

解决方式：**哑元**（不发挥作用的一个参数）

++x：前（缀）加——一元运算符 operator++(X x); X.operator++() 返回值是引用（最后返回的是操作后的 x 的值，最好返回 x 本身）

x++：后（缀）加——二元运算符 operator++(X x, **int**); X.operator++(int) 返回值是值（只要返回一个值就可以了）

一般调用时，int 赋值为 0.

（注意不可能给 int 设置默认值。否则 operator++(x)究竟调用谁？）

［ 函数声明原型举例 ］

```
class Counter{
```

```
        int value;
        int alarm;
    public:
        Counter &operator++();
        Counter operator++(int);
};

Counter &Counter::operator++()
{
    Counter tmp;
    value++;
};
```

9. 输入输出流重载

【输出重载函数原型】

```
ostream& operator<<(ostream &os, const T& X)
                                    //因为 this 指针自动加在左边，所以不能成员函数
                                   //operator<<的系统原型规定了这一点
{    };
//ostream 为输出流类型，如 cout 就是一个 ostream。
[e.g.] {   os << X.num << "/" << X.den; return os; }

main()中调用:       [cout]<<[x]
                     &os    &X
```

【输入重载函数原型】

```
istream& operator>>(istream &in, T& obj)   //不可能设置成常量，否则输入了寂寞
{    };
[e.g.] { in >> X.num >> X.den; return in; }
```

10. 类型转换

【内置类型到类类型的转换】

1）隐式转换

编译器会自动寻找是否存在带缺省值的构造函数能构造一个类类型变量。

e.g. RaT r1;（有理数类型）

使用“2+r1”：2 调用构造函数，构造“2”.num=2; "2".den=1; //有缺省值

本质：调用 RaT r=2;　只要该语句合法，隐式转换就够了。

有的时候隐式转换会出错，**e.g.** 2×2 矩阵构造函数缺省值是 0，但“2+Matrix”应该把 2 转化成 2E（E 是单位阵）。

如果隐式转换会出错，不允许隐式转换→ expilicit RaT(int x=0, int y=1);

//不允许系统在强制类型转换时使用构造函数，此时调用 2+r1 会编译报错。

因此，我们建议显式转换。

2）显式转换： 二元运算符重载为友元全局函数。

不用成员重载？

[e.g.] r1=2+r2→ 若成员重载，出错。(要求左操作数必须是 this 指针，即 RaT 类型)

友元全局函数：2+r2 和 r2+2 都是合法的表达式。

【类类型到其它类型（内置类型/类类型）的转换】

重载类型转换函数。**必须重载成成员函数!**

operator <目标类型名>() const

[e.g.] operator double() const { return(double(num)/den); } 调用：double(r);

重载的目的：让上层（main 函数和其他调用函数）中对类类型的使用和系统内置类型高度相似。

【组合/继承】

1. 组合：类的套娃，类作为类的成员。

```
class ComplexR
{
    RaT real;
    RaT imag;
    public:
        ComplexR(int r1=0, int r2=1, int i1=0, int i2=1): real(r1,r2), imag(i1,i2) { …… };
};
```

2. 继承：在已有类的基础上增加或修改少量代码直接得到/构造新的类。→解决代码重复

要修改大量的话这边建议重写呢。

基类/父类 （源）

派生类/子类/导出类 （目标）

派生类定义

class <派生类名> : <派生方法> <基类名>

{ //派生类新增的数据成员和成员函数

};

派生方式：public/private/protected 约束了基类中的函数调用权限。

公有/私有/保护派生

(*保护派生 protected 现在用得比较少)

~~protected 成员是一类特殊的私有成员，它不可以被全局函数或其他类的成员函数访问，但能被派生类的成员函数访问~~

~~protected 成员破坏了类的封装，基类的 protected 成员改变时，所有派生类都要修改~~

[e.g.]

```
class RaT : public inT          //inT 是自己定义的整型
{
```

```
    int y;
    public: func(int k);
};
```

内存中大概长这样：　RaT[　　[inT]　　y;　　func;　　]

基类中	继承方式（派生方式）	派生类中
public	public	public
public	protected	protected
public	private	private
protected	public	protected
protected	protected	protected
protected	private	private
private	public	派生类不可见
private	protected	派生类不可见
private	private	派生类不可见

基类成员的访问说明符	**继承类型**		
	public 继承	**protected 继承**	**private 继承**
public	在派生类中为 public	在派生类中为 protected	在派生类中为 private
	可以由任何非 static 成员函数、友元函数和非成员函数访问	可以直接由任何非 static 成员函数、友元函数访问	可以直接由任何非 static 成员函数、友元函数 s 访问
protected	在派生类中为 protected	在派生类中为 protected	在派生类中 private
	可以直接由任何非 static 成员函数、友元函数访问		
private	在派生类中隐藏	在派生类中隐藏	在派生类中隐藏
	可以通过基类的 public 或 protected 成员函数或非 static 成员函数和友元函数访问		

3. 派生类的构造函数和析构函数

派生类不继承基类的构造函数、析构函数和赋值运算符

但是可以调用后者。

1）派生类构造函数

派生类构造函数名（参数表）：**基类构造函数名（参数表）**｛ 自己的构造;｝

↑ 派生类的初始化列表

2）派生类析构函数

先析构自己的东西，再调用基类的析构函数析构基类。

[e.g.]

```
class Pond{  ……  };
class Pool: public Pond
 { ……
   Pool( 参数表 ): Pond(参数) { …… };
   ~Pool() { …… };                          //只析构自己的内容
```

…… };

4. 组合 vs 继承

构造顺序：按照类中的定义顺序。

先**基类**，再组合中的成员类**（按定义顺序）**，再自己的成员构造

析构顺序相反。

[e.g.]

```
class A { };
class B { };
class D: public B { A   a1;   A   a2; };
class A
{   int objID;
   public: A(int id=0) {objID =id; cout <<“A”<<objID<<“被构造\n”;}
   ~A() {cout <<“A”<<objID<<“被析构\n”;}
};
class B
{   int objID;
   public: B(int id=0){objID =id; cout <<“B”<<objID<<“被构造\n”;}
   ~B() {cout <<“B”<<objID<<“被析构\n”;}
};
class D: public B
{   int objID; A a2,a1;
   public: D(int id=0,int idb=0,int ida1=0,int ida2=0):a1(ida1), B(idb), a2(ida2)
        {objID =id; cout <<“D”<<objID<<“被构造\n”;}
   ~D() {cout <<“D”<<objID<<“被析构\n”;}
};
int main()    {   D d(10,20,30,40); return 0; }
```

开始构造 D → 无论初始化列表里顺序如何（**a1(ida1), B(idb), a2(ida2)**），先调用基类 B 的构造函数构造基类 → 再调用类成员 A 的构造函数构造组合的类成员（按定义顺序，与初始化列表顺序无关）→ 构造自己的成员 → 析构自己的成员 → 调用组合的类成员 A 析构函数析构 → 析构基类。

输出：

```
B20 被构造
A40 被构造      //按照定义顺序，a2 先，a1 后。
A30 被构造
D10 被构造
D10 被析构
A30 被析构
A40 被析构
B20 被析构
```

5. **重定义基类中函数：原型完全相同。**派生类函数覆盖基类函数。

此时调用基类的这一函数需要使用作用域符（::）

[e.g.]

```
class Base { …… public void Asshole(int) const {cout<<"Fxxk you.\n"};} …… };
class Damn: public Base
{ ……
  public void Asshole(int) const          //原型完全一致，覆盖 Base 的 Asshole()函数
   {
    cout<<"Mothxxxxxking"};
    Base::Asshole(0);   // 调用了 Base 的 Asshole()函数
   }
  public void Asshole(int) { cout << "Whatever.\n"; }    //原型不一致，不覆盖
}
```

6. **派生类的赋值符**

派生类不继承基类的构造函数、析构函数和赋值运算符

但是可以调用后者。

[e.g.]

```
class People { public: …… private: char *name; int age; };
class Student: public People {public: …… private: int s_no; char *class_no; };
People &operator=(const People &other)
{   if (this == &other) return *this;
   delete name;   name = new char[strlen(other.name) + 1];
   strcpy(name, other.name);
   age = other.age;
   return *this;
}
Student &operator=(const Student &other)
{   if (this == &other) return *this;
   s_no = other.s_no;
   delete class_no;
   class_no = new char[strlen(other.class_no) + 1];
   strcpy(class_no, other.class_no);
   People::operator=(other);                    //调用基类的赋值符
   return *this;
}
```

7. **派生类也可作为基类。**凑个字数，没什么需要具体展开的。

8. **派生类隐式转换为基类：把派生类自己的都扔了，只保留基类的部分。可扔不可补。**

1）**赋值：**只复制基类部分。 Derive a; Base r = a; //将 a 的基类部分赋值给 r

2）**指针：**基类指针访问派生类的基类部分，间接修改派生类的基类部分。

[e.g.] Derive a; Base* bp = a; bp->setx(10); //a 的基类部分被修改

3）引用：基类引用就是派生类的基类部分的别名。

[e.g.] Derive a; Base& bp = a;　bp.setx(10);　//就是在修改 a 的基类

对派生套娃会一层层往下直到找到和自己一个类型的为止，找不到就是编译报错。

[e.g.]

```
class Base{};
class Derive1:public Base {};
class Derive2:public Derive1 {};
class Derive3:public Derive2 {};

Derive3 a;
Derive1* p = a;     //访问 a 的 Derive2 基类的 Derive1 基类部分
Base& q = a;        //就是 a 的 Derive2 基类的 Derive1 基类的 Base 基类部分
```

9. 派生类与基类的强制类型转换：一般只允许派生类到基类的强制类型转换。

```
Class Base {};
Class Derive: public Base {};

Base b, *p;
Derive d, *q;

p=&b;    q=&d;      p=&d;    p=q;    //ok
q=&b;    q=p;                        //error，可扔不可补
q=reinterpret_cast<Derive*>(p);
q=reinterpret_case<Derive*>(&b);      //ok，但不推荐（IDE 随便给你补了随机的东西）
```

10. 虚函数/多态性

简单易懂版：

指针和引用指着的是派生类的基类部分，但是他们想用派生类（最外层）的函数。

于是我们把这个函数设置成了虚函数，用到函数的时候就去找派生类的这个函数。

即：在编译的时候才决定调用哪个函数。（同一内容面对不同对象会调用不同函数→**多态性**）

显然要求基类和派生类中这个函数的函数原型完全一致。

严谨版：当把一个函数定义为虚函数时，等于告诉编译器，这个成员函数在派生类中可能有不同的实现。必须在执行时根据传递的参数来决定调用哪一个函数。当基类指针指向派生类对象或基类对象引用派生类对象时，对基类指针或对象调用基类的虚函数系统会到相应的派生类中寻找此虚函数的重定义。

【声明方式】virtual +函数声明。

只要基类有声明，所有派生类与之原型完全相同的函数都被认为是虚函数。

[e.g.]

```
class Shape {    public: virtual void print() const{cout<<“Shape”<<endl;} };
class Point: public Shape {
        public: virtual void print() const {cout<<“Point”<<endl;}     }
class Circle: public Point {
        public:    virtual void print () const{cout<<“Circle”<<endl;}    }
class Cylinder:public Circle {
        public:    virtual    void print () const {cout<<“Cylinder”<<endl;}    }
int main() {
   Circle Cr; Cylinder Cy; Point Pt; Shape* x;
   x = &Cr;     x->print();
   x = &Cy;     x->print();
   x = &Pt;     x->print();     return 0; }
```

输出： Circle↙Cyclinder↙Point↙

三次都是”x->print()”但调用的函数不同 → **多态性**。

11. 静态联编/动态联编

静态联编：编译之前确定好了。 Cylinder Cy; 则 Cy.print();是静态联编

动态联编：编译时才能确定调用谁。

Shape* x = &Cr, & y = Pt; 则 x->print(); y.print();是动态联编。

动态联编条件：1）要有虚函数声明；2）类型继承关系的建立；3）调用虚函数操作的是指向对象的指针或者对象引用；或者是由成员函数调用虚函数

12. 成员函数调用虚函数：和外部调用一样，直接到最外层。

想调用自己的函数：Myself::Act();

讨论

```
class A{
 public:
      virtual void act1( );
      void act2( ) {this->act1( );}
};
void A::act1( )
{
    cout << “A::act1( ) called.\n" ;
}
```

B::act1( ) called.

```
class B : public A{
 public:
      void act1( );
};
void B :: act1( )
{
    cout << "B::act1( ) called\n";
}
int main ( )
{
   B b;
   b.act2( );
   return 0;
}
```

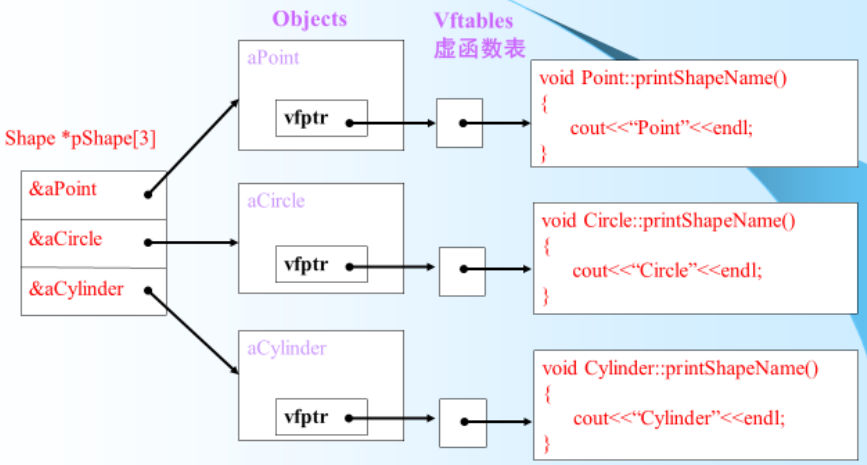
讨论（续）

```
class A
{
 public:
      virtual void act1( );
     void act2(){A::act1():}
};
void A::act1( )
{
    cout << “A::act1( ) called.\n" ;
}
```

A::act1( ) called.

```
class B : public A
{
 public:
      void act1( );
};
void B :: act1( )
{
    cout << "B::act1( ) called\n";
}
int main ( )
{
   B b;
   b.act2( );
   return 0;
}
```

C++动态联编实现机制

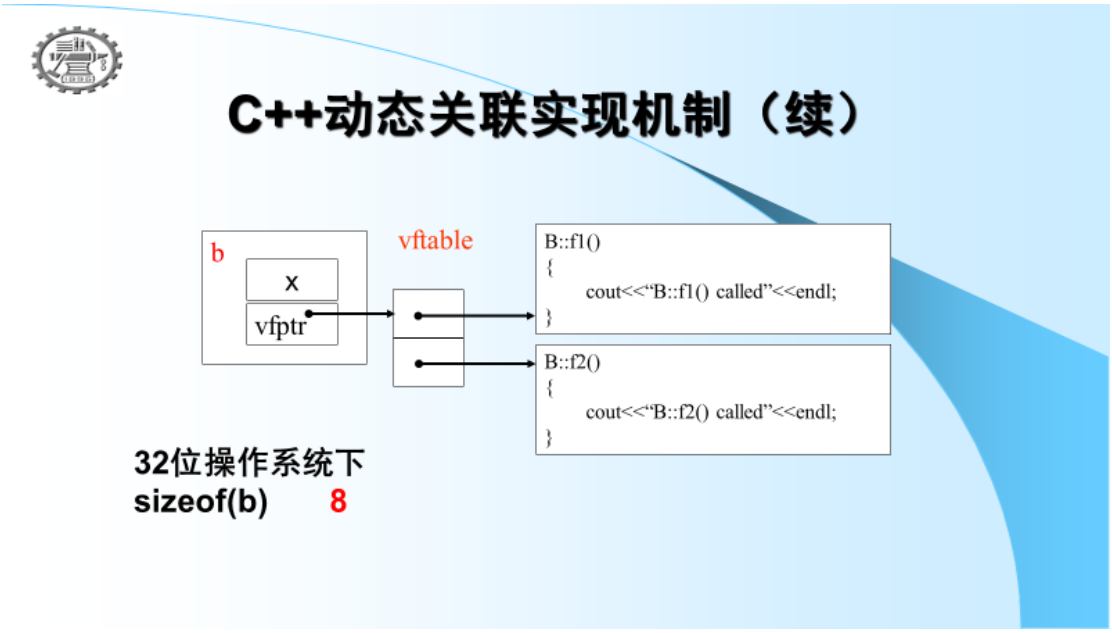


虚函数表(Virtual Function Table)

```
class A
{
public:
   virtual void f1() { cout<< "A::f1() called" << endl;}
   virtual void f2() { cout<< "A::f2() called" << endl;}
private:
    int x;
};

class B : public A
{
public:
    virtual void f1() { cout<< "B::f1() called" << endl;}
    virtual void f2() { cout<< "B::f2() called" << endl;}
};
```

```
int main()
{
   B b;
   cout << sizeof(b);
   return 0;
}
```



13. 虚析构函数

基类指针/引用动态开辟了派生类，然后要 delete[]，发现只释放了基类空间。所以需要把析构函数搞成虚函数。

[e.g.]

```
class Base{ …… virtual ~Base(){……}};
class Derive{ …… ~Derive() {……}};
                    //因为基类的析构函数已经声明是虚函数，可以不再写 virtual
int main() {
   Base *bp = new Derive;
   delete bp;                    //ok
return 0; }
```

14. 纯虚函数与抽象类

有的时候基类只是为了便于管理各个类（比如 Dog, People, Rose 然后便于管理在上面加个啥都没有的可爱的 Creature 类，这样 Creature *a 就可以随便用非人类动物、植物或者人类赋值。）

此时定义虚函数没有什么必要还占空间（由类管理的函数代码空间，属于一个类）。

引入**纯虚函数：**是一个在基类中说明的虚函数，它在该基类中没有定义，但要在它的派生类里定义自己的版本，或重新说明为纯虚函数。

【声明】 virtual 类型 函数名（参数表）=0

啥都没有就只有纯虚函数的类称为抽象类。

如果它有一个数据成员或者一个非纯虚函数的函数成员，则是具体类。

~~以下内容均不考~~

【模板】

1. **类模板：**类似函数模板，把一些类型替换成 T 就完事了。

```
template<class T1, class T2>
DynArr
{
   T1 low, high;
   T2* storage;
};
```

2. **模板实例化**

DynArr<int,int> a;　　//一个下标整型，内容整型的动态数组。
DynArr<char,int> b; //一个下标字符型，内容整型的动态数组。
　　→　可视为[PASCAL] var b:array['a'..'z'] of longint;的替代品
DynArr<int,char> c; //一个下标整型，内容字符型的动态数组。

3. **非类型参数：**

如果想在类里加上数组（不用指针动态数组实现方式），那么编译时就得知道数组大小，如何告诉模板？→ 非类型参数

```
template<class T1, class T2, T1 low, T1 high, int suibianxiede, char meishayong>
DynArr_2
{
   T1 lh, rh;
   T2 storage[int(high-low+1)];
};
```

调用：DynArr_2<char, int, 'a', 'z'> a;

【文件输入输出与异常处理】

1.【异常输入流】

输入异常处理　【实验 7】

cin.clear()：非法输入后，恢复输入（更改 cin 的状态标识符）否则无法继续输入，所有后续值由 IDE 随机分配

cin.sync()：清空输入缓存区，将前面已有的输入（除成功读入的内容已存储外）全部清空。

cout.put('A')　　返回值是 cout
所以有：cout.put('A').put(65).put('\n');　　//AA↙

cout.write(s,10);　　//输出 s 字符串的前 10 位

不带参数的 cin.get()函数：((ch=cin.get())!=EOF)
带一个参数的 cin.get()函数：cin.get(str[0]); //读入字符串后要自己加'\0'
带三个参数的 cin.get()函数：cin.get(str,10,ch);　　//ch 不会被存入 str，但 ch 仍会留在输入缓存区中，在下一次读入的时候被读入下一个变量中。

cin.read(str,10);　//从输入缓存区中读入 10 个字符（不管它们是什么）

格式化输入输出
整数流基数：dec，oct，hex，setbase
设置浮点数精度：precision，setprecision
设置域宽（场宽）：setw，width
设置域（场）填充字符：fill，setfill

文件输入输出
ofstream out("file.txt");　//与 file.txt 建立文件输出流
判断能否输出到文件（重名文件/只读文件）→ if (!out) { 处理; }
out.close();
　　//将输出缓存区（内存）内的内容放入对应文件，并在文件尾加 EOF；实际上不加在程序结束的时候也会给这个文件加上 EOF

ifstream in("file.txt");　　//也会准备输入缓存区（block / sector）
if (!in) { 处理; }
in.close();

异常处理
try{

可能异常抛出的代码

}
catch(<type> vari_name) { 处理异常; }
catch(<type> vari_name) { 处理异常; }
catch(<type> vari_name) { 处理异常; }
……

catch(...)　　//任意类型的异常，类似 switch 的 default。