

C++程序设计

谭浩强编著

清华大学出版社

课件制作：南京理工大学 陈清华 朱红

第一章C++概述

C++语言发展历史

自从1946年第一台电子数字计算机ENIAC问世以来，随着计算机应用领域的不断扩大，促进了计算机技术的高速发展，尤其是近年来计算机的硬件和软件都是日新月异。作为应用计算机的一种工具——程序设计语言，得到不断的充实和完善。每年都有新的程序设计语言问世，老的程序设计语言不断地更新换代。

20世纪60年代，Martin Richards为计算机软件人员在开发系统软件时，作为记述语言使用而开发了BCPL语言(Basic Combined Programming Language)。1970年，Ken Thompson在继承BCPL语言的许多优点的基础上发明了实用的B语言。到了1972年，贝尔实验室的Dennis Ritchie和Brian kernighan在B语言的基础上,作了进一步的充实和完善，设计出了C语言。当时，设计C语言是为了编写UNIX操作系统的。以后,C语言经过多次改进,并开始流行。C++是在C语言的基础上发展和完善的，而C是吸收了其它语言的优点逐步成为实用性很强的语言。

C语言的主要特点是：

- 1、C语言是一种结构化的程序设计语言，**语言本身简洁、使用灵活方便**。既适用于设计和编写大的系统程序，又适用于编写小的控制程序，也适用科学计算。
- 2、**它既有高级语言的特点，又具有汇编语言的特点**。运算符丰富，除了提供对数据的算术逻辑运算外，还提供了二进制的位运算。并且也提供了灵活的数据结构。用C语言编写的程序表述灵活方便，功能强大。用C语言开发的程序，其结构性好，目标程序质量高，程序执行效率高。

3、**程序的可移植性好**。用C语言在某一种型号的计算机上开发的程序，基本上可以不作修改，而直接移植到其它型号和不同档次的计算机上运行。

4、**程序的语法结构不够严密，程序设计的自由度大**。这对于比较精通C语言的程序设计者来说，可以设计出高质量的非常通用的程序。但对于初学者来说，要能比较熟练运用C语言来编写程序，并不是一件容易的事情。与其它高级语言相比而言，调试程序比较困难。往往是编好程序输入计算机后，编译时容易通过，而在执行时还会出错。但只要对C语言的语法规则真正领会，编写程序及调试程序还是比较容易掌握的。

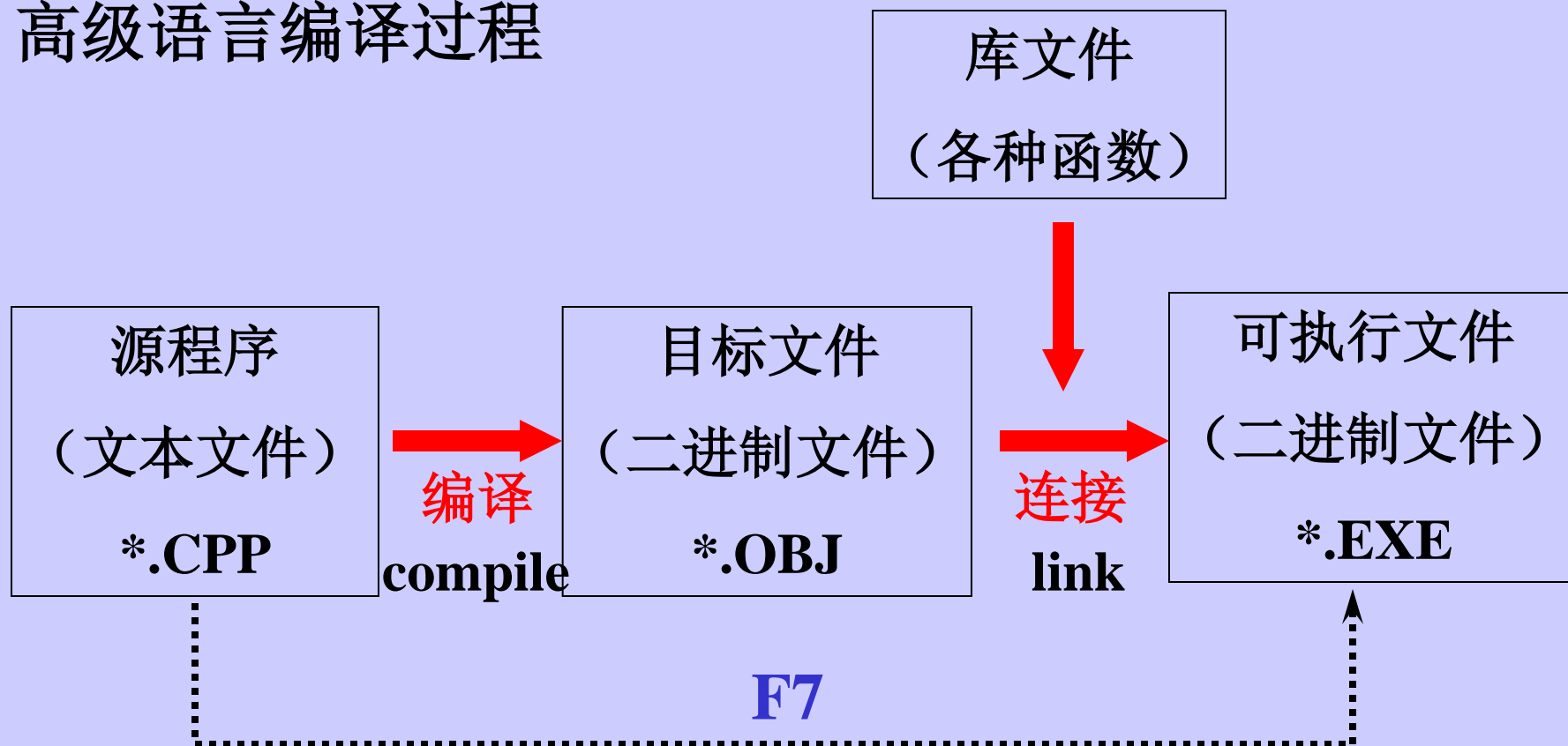
随着C语言应用的推广，C语言存在的一些缺陷或不足也开始流露出来，并受到大家的关注。如：**C语言对数据类型检查的机制比较弱；缺少支持代码重用的结构；随着软件工程规模的扩大，难以适应开发特大型的程序等等。**

为了克服C语言本身存在的缺点，并保持C语言简洁、高效，与汇编语言接近的特点，1980年，贝尔实验室的Bjarne Stroustrup博士及其同事对C语言进行了改进和扩充，并把Simula 67中类的概念引入到C中。并在1983年由Rick Maseitti提议正式命名为C++（C Plus Plus）。后来，又把运算符的重载、引用、虚函数等功能加入到C++中，使C++的功能日趋完善。

当前用得较为广泛的C++有：VC++（Visual C Plus Plus）、BC++（Borland C Plus Plus）、AT&T C++等。

简单的C++程序介绍

高级语言编译过程



在Vitual C++系统中，可直接从源程序编译连接至可执行程序，但依然要生成*.OBJ及*.EXE这两个文件。

一个简单的C++程序

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

包含文件

函数体
开始

主函数

```
int main(void )
```

分号，一条完整
语句的结束符

```
{ cout<<"I am a student.\n"; //输出字符串
```

```
}
```

函数体
结束

输出流，在屏幕上打
印引号内的字符串

注释或说明

本程序编译执行后，在DOS屏幕上打印出

I am a student.

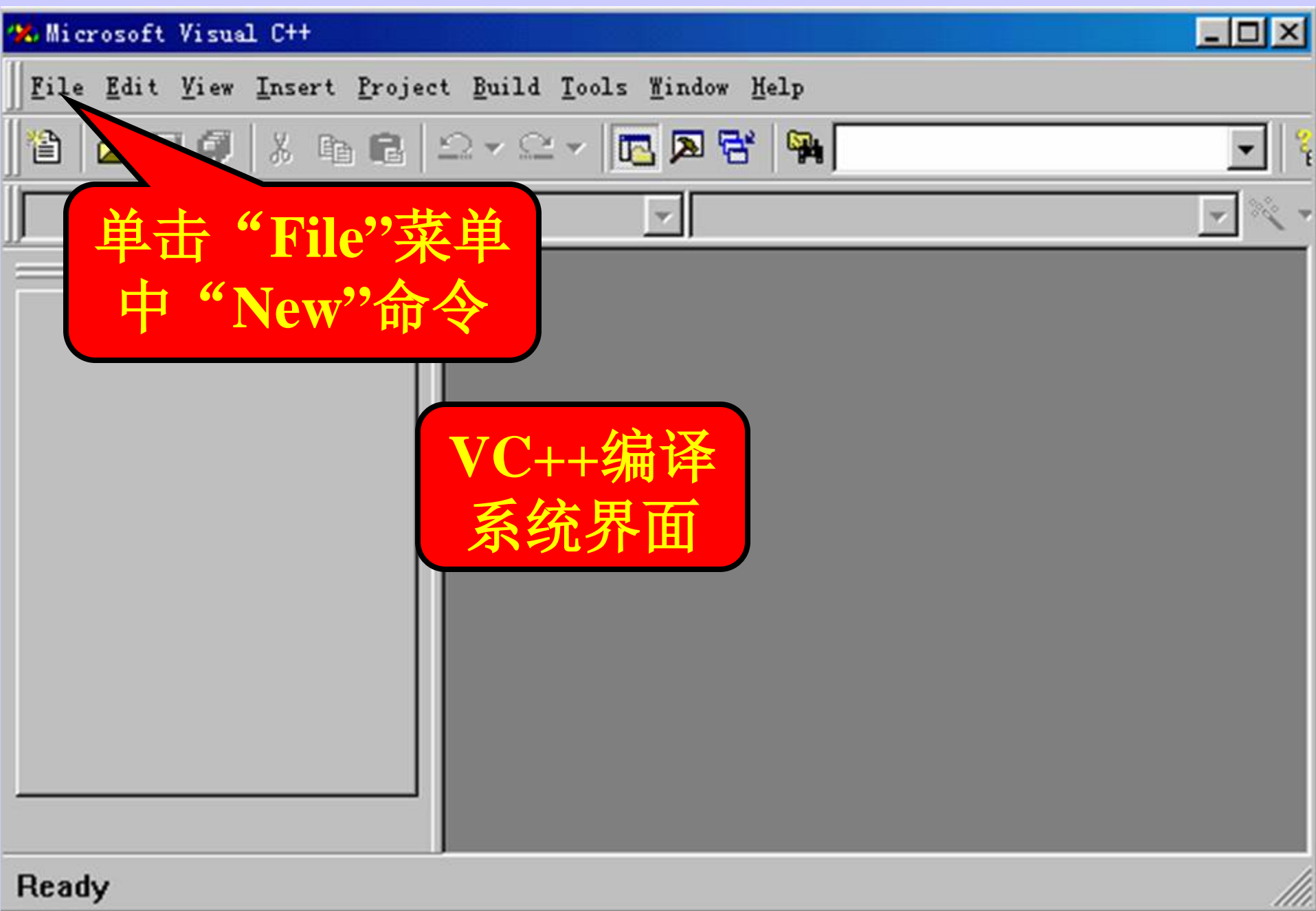
编译过程:

- 1) 启动**Visual C++**,选择“文件”菜单中的“新建”命令, 选择“文件”标签中的“**C++ Source File**”选项。
- 2) 选择源程序存放的目录和输入源程序名, 单击“确定”。
- 3) 在编辑器中编写源程序。
- 4) 单击**F7**或“编译”中的“重建全部”编译源程序, 若编译通过, 单击“执行”, 在**DOS**屏上看结果, 任按一键返回编辑器。



启动VC++
编译系统





选择“Files”选项卡

Files Projects Workspaces Other Documents

选择C++源
文件命令

输入文件名

File

c1-1.cpp

Location:

D:\C++

输入文件
存放位置

单击选择
驱动器

选择驱动
器或目录

Choose Directory

Directory name:

C:\Documents and Settings\zhu

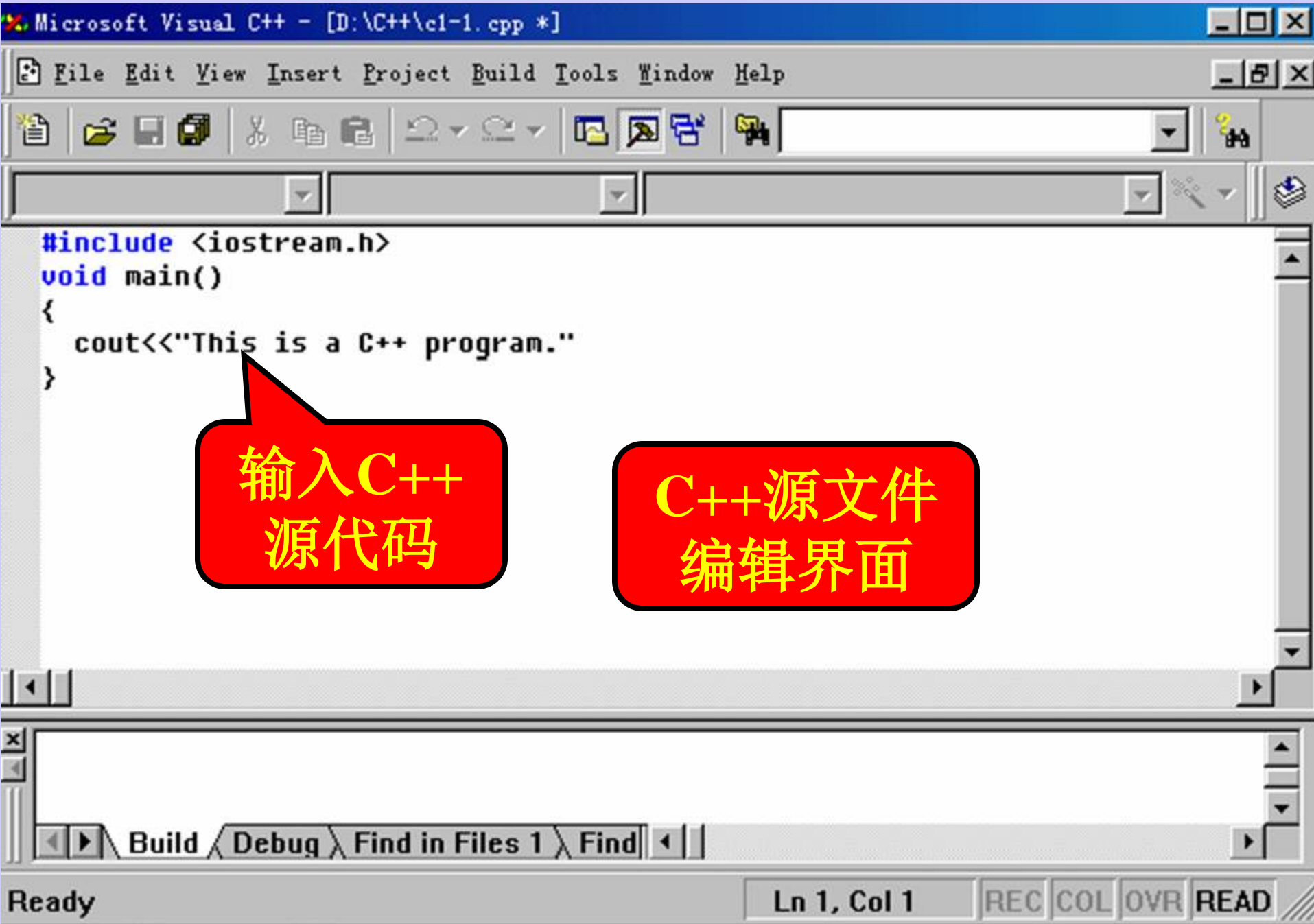
c:\
Documents and Settings
zhu
「开始」菜单
Cookies
Favorites
My Documents
桌面

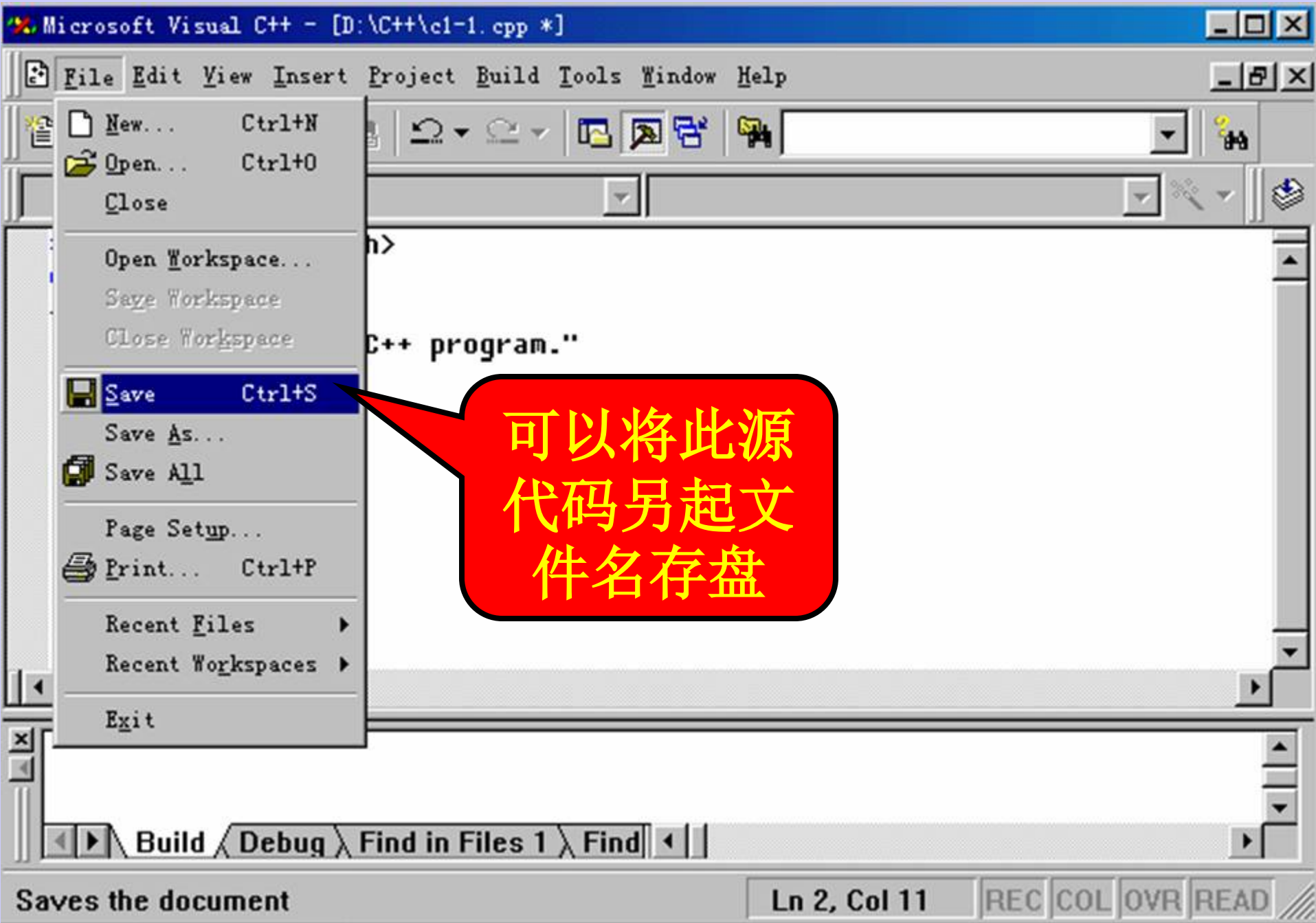
驱动器:

c:
a:
c:
d: ZHU1
e:
f:
g:

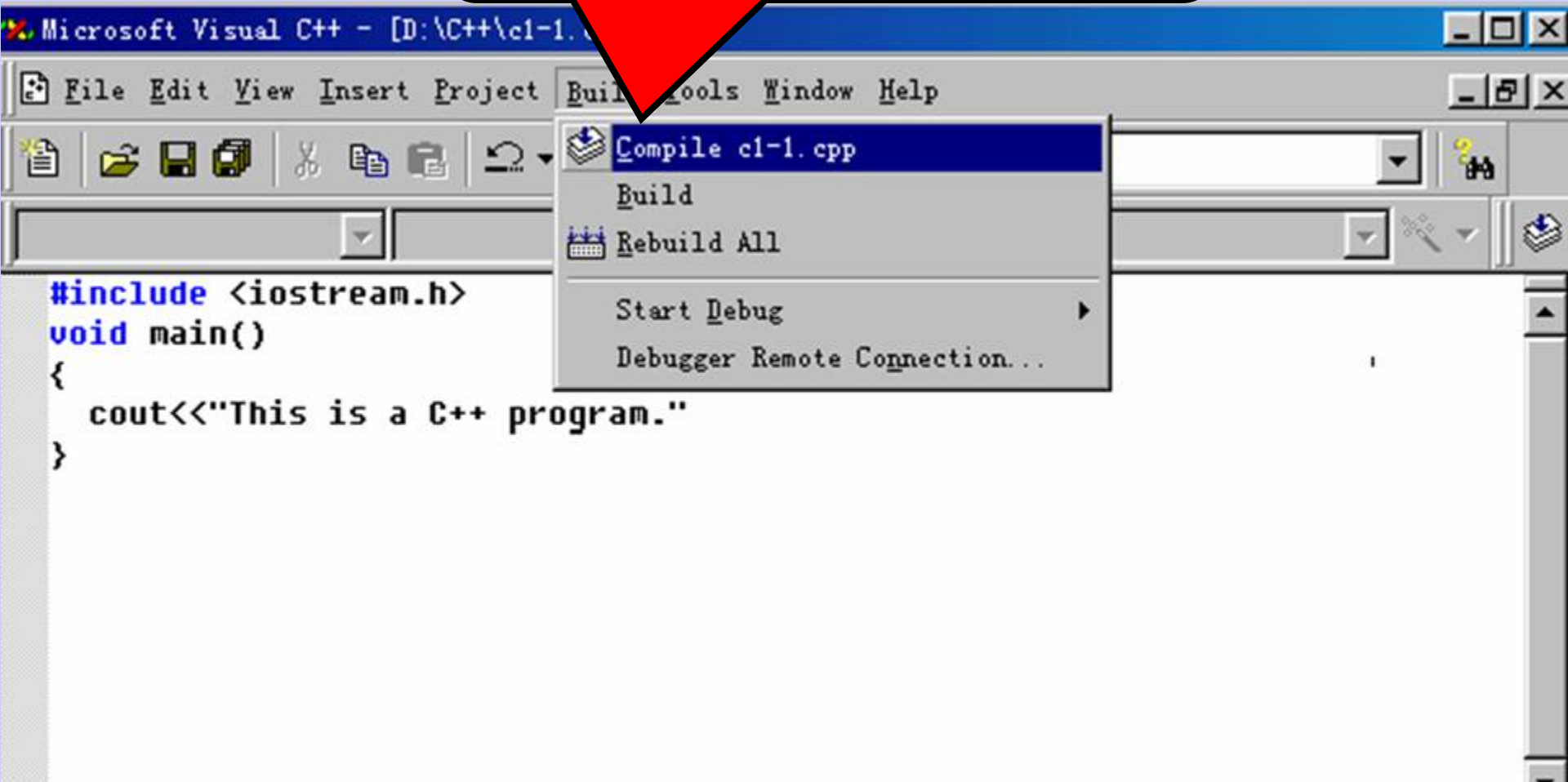
OK

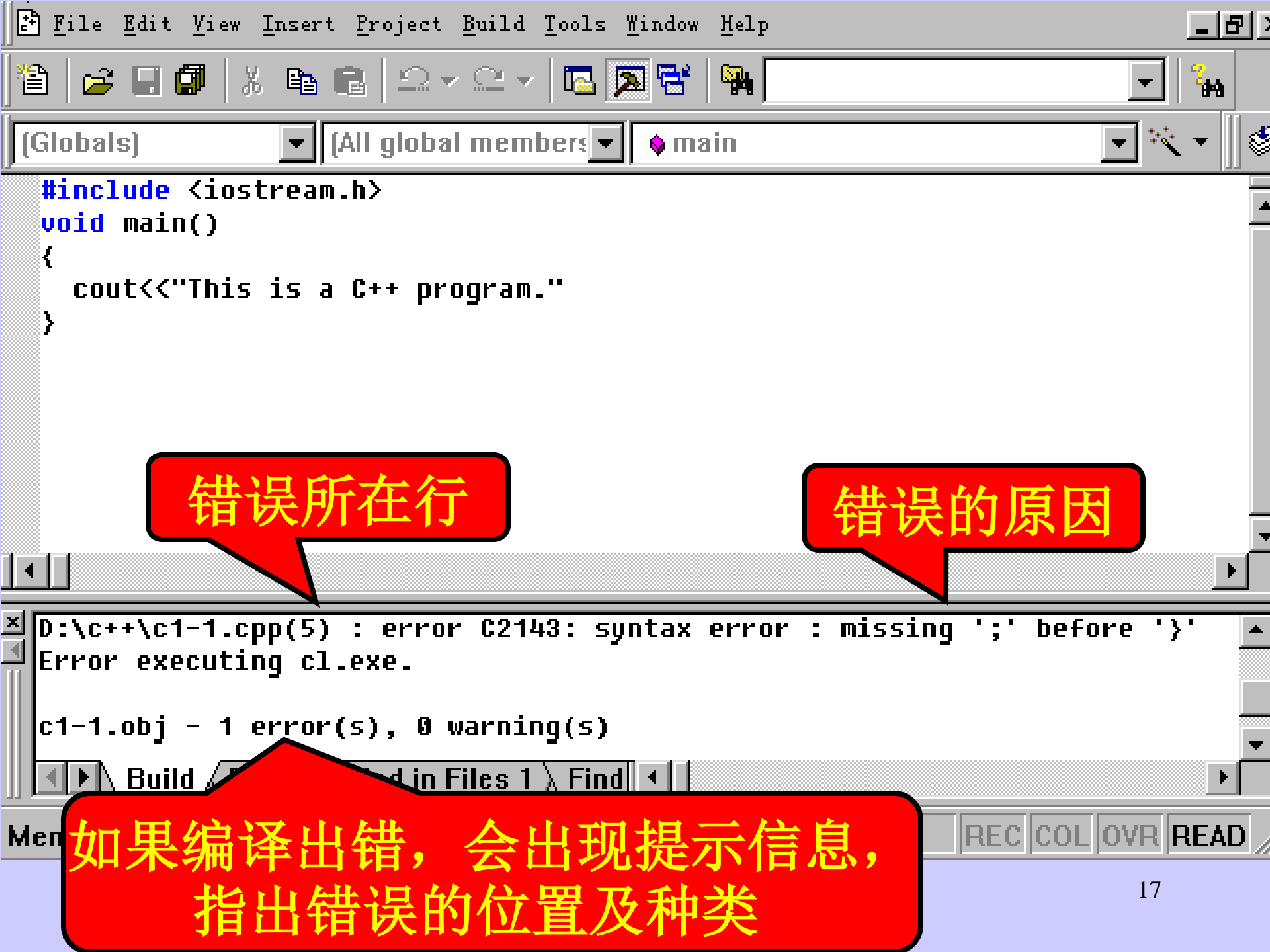
Cancel





选择编译命令，将源文件.cpp生成.obj文件





错误所在行

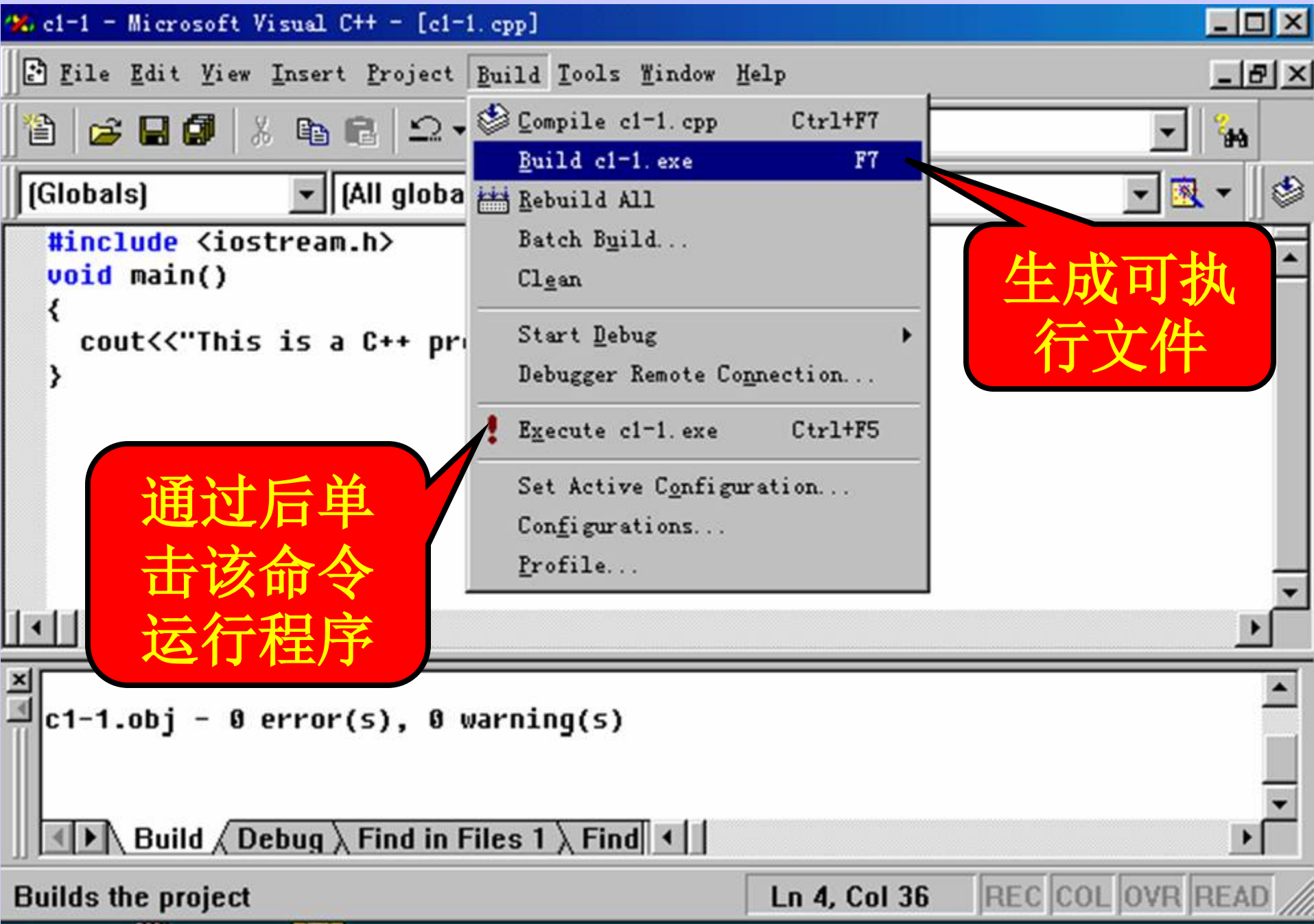
错误的原因

D:\c++\c1-1.cpp(5) : error C2143: syntax error : missing ';' before '}'
Error executing cl.exe.

c1-1.obj - 1 error(s), 0 warning(s)

如果编译出错，会出现提示信息，
指出错误的位置及种类





MS
DOS c1-1

自动



This is a C++ program.
Press any key to continue

运行结果显示
在DOS屏上

注意：不可以在软盘上
运行程序！应该把保存
在软盘中的源文件拷贝
到硬盘的目录中再运行！

例子

文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)

后退 搜索 文件夹

文件夹

- 本地磁盘 (E:)
 - 2解答
 - 2003_12文件
 - 2003计算机基础_孙一平book
 - 2004-8_课程设计新版
 - 2004_5_毕业设计
 - 2004_VC++备课讲义
 - 2004_上机作业
 - C++_EXSE
 - Debug
 - 例子
 - 2004毕业设计
 - 2005毕业设计
 - ADSL
 - C_Lecture
 - C++
 - C课程设计
 - daily_writing
 - MASM
 - MFC讲义Copy
 - NeuralNetwork神经网络资料
 - Novel_feeling
 - PYjj
 - VC++上机作业

C++_Lecture_1_简单
例子.cpp
C++ Source file

未编译前，只有一个源程序

源程序所在目录

例子

文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)

后退

搜索

文件夹

文件夹

本地磁盘 (E:)

2解答

2003_12文件

2003计算机基础_孙一平book

2004-8_课程设计新版

2004_5_毕业设计

2004_VC++备课讲义

2004_上机作业

C++_EXSE

Debug

例子

Debug

2004毕业设计

2005毕业设计

ADSL

C_Lecture

C++

C课程设计

daily_writing

MASM

MFC讲义Copy

NeuralNetwork神经网络资料

Novel_feeling

nv...



C++_Lecture_1_简单
例子.cpp
C++ Source file



C++_Lecture_1_简单
例子.dsp
Project File



C++_Lecture_1_简单
例子
HTML Document



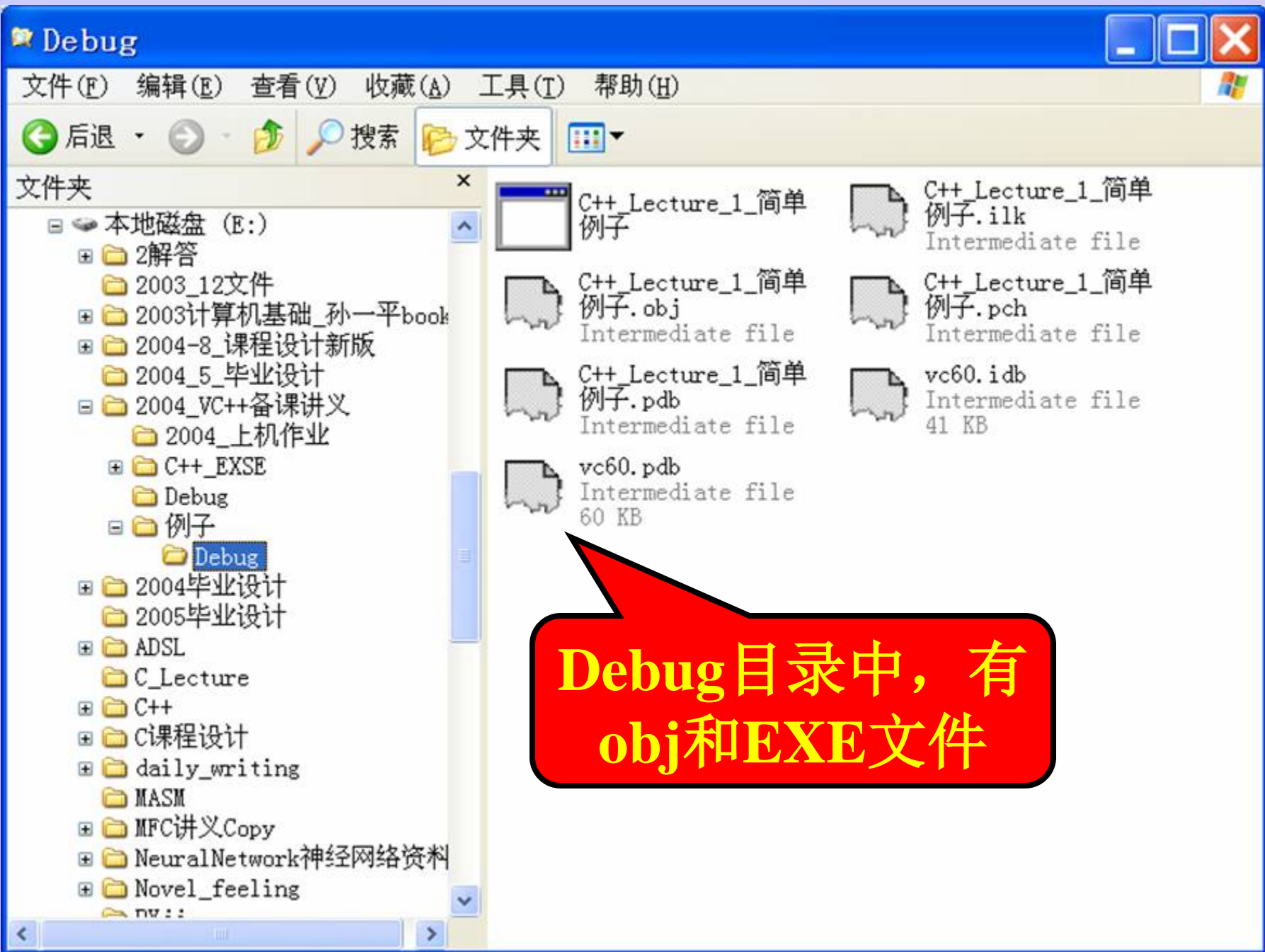
Debug



C++_Lecture_1_简单
例子
NCB 文件

编译运行后，出现众多附加文件

同时，产生一个子目录Debug



另一个例子

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

```
void main(void)
```

```
{
```

```
    cout << "i="; //显示提示符
```

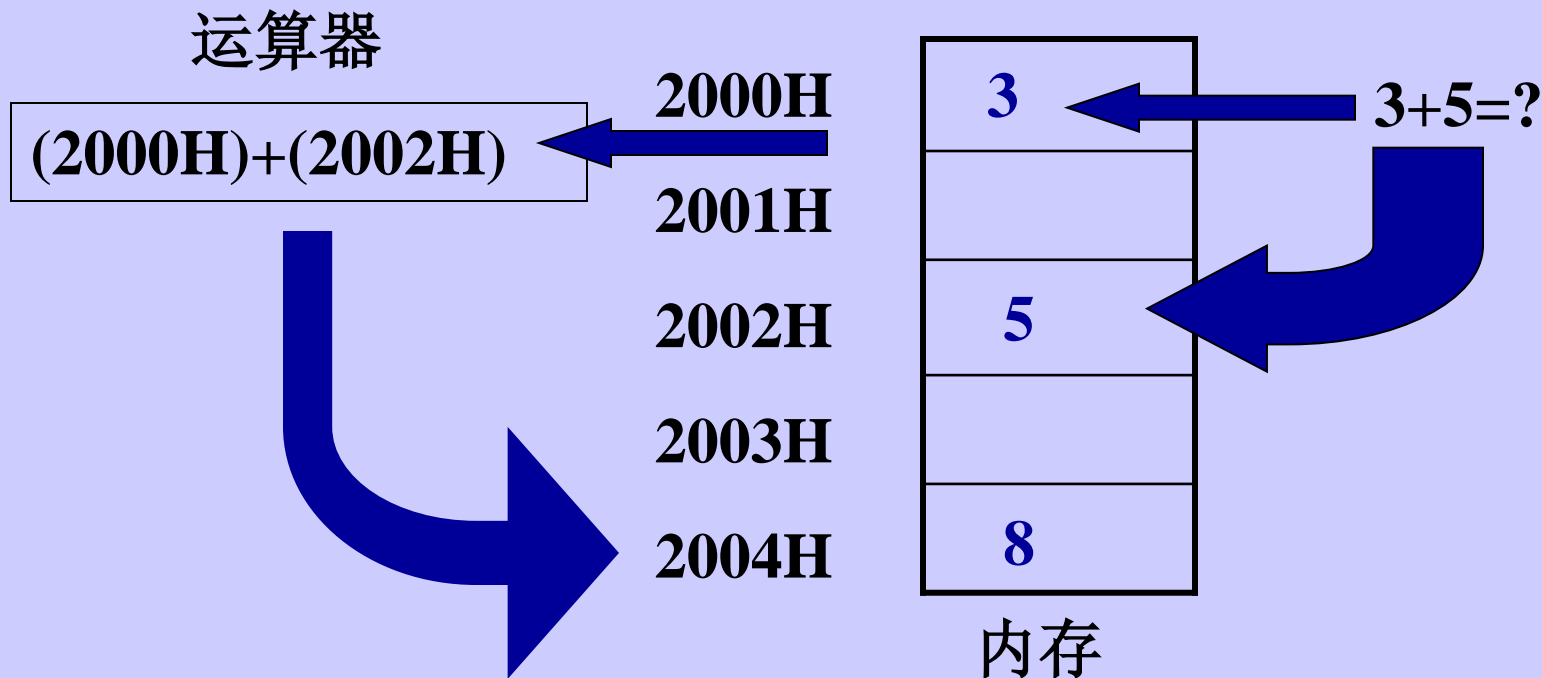
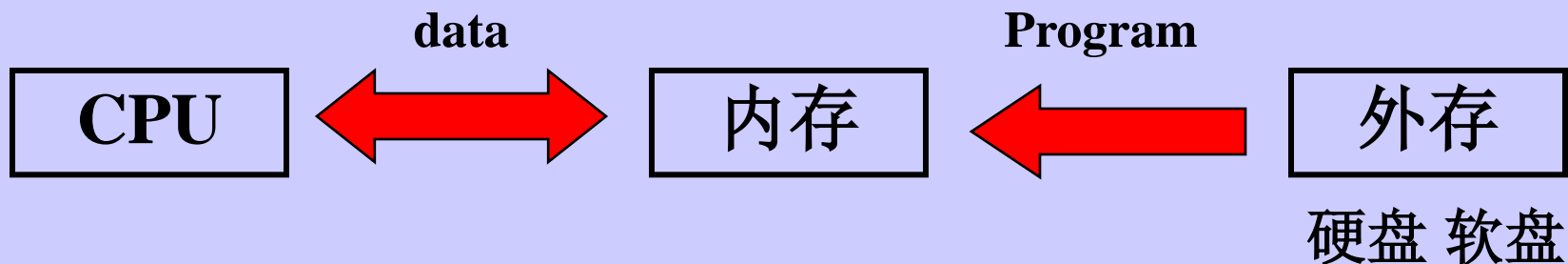
```
    int i;        //说明变量i
```

```
    cin >>i;      //从键盘上输入变量i的值
```

```
    cout << "i的值为: " <<i<<"\n"; // 输出变量i的  
    值
```

```
}
```


第二章 数据类型、运算符与表达式



用一个字节表示整数，范围为-128~127；用两个字节表示整数，范围为-32768~ 32767。一般用四个字节表示整数。(举例)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

有符号数 无符号数

32767 **32767**

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

32766 **32766**

.....

.....

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1 **1**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0 **0**

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-1(补码) **65535**

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-2 **65534**

.....

.....

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-32767 **32769**

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

-32768 **32768**

常量与变量

常量：在程序运行过程中，其值一直保持不变的量为常量。

常量也区分不同的类型：30，40 为整型，30.0，40.0为实型，编辑器只是根据其表面形式来判断其类型。

变量：在程序运行过程中，其值可以改变的量称为变量。

变量在程序的执行中能够赋值，发生变化。变量有一个名字，并在使用之前要说明其类型，一经说明，就在内存中占据与其类型相应的存储单元。

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

```
#define PRICE 30 //常量，在程序中保持不变
```

```
void main(void)
```

```
{ int num, total; //定义变量,在内存中开辟区间
```

```
    num=10;      //变量赋值,10为常量
```

```
    total=num*PRICE;
```

```
    cout<<"total="<<total; //输出结果
```

```
}
```

num	total
10	300

PRICE
30

其中: num=10

total=num*PRICE

是赋值号，不同于数学意义上的等号。

C++中有多种数据类型，均有常量与变量之分，各占不同的内存空间，正确定义与使用数据是编写程序的基本前提。

变量名的命名方法：

变量名、数组名、函数名...称为**标识符**。

标识符只能由**字母、数字、下划线**这三种字符组成，且第一个字符必须为字母或下划线，长度不大于247个字符，**大小写不通用**。（关键字不能作为标识符）。

关键字即是**VC++**的语法要求中使用的字。

如 `int` `if` `while` 等。

正确的标识符：`INT`，`sum`，`de12`，`SUM`等。**变量必须使用前定义，以分配空间。**

举例说明

abc *English* 2xy *x-y* if *Else* *b(3)* ‘def’
***Chine_bb* b3y AbsFloat float**

一般变量都是用匈牙利命名法命名的。

int nCount;

char chChoice;

整型数据

整型常量:

常量是根据其表面形式来判定，整型量即是没有小数点的整数，范围： $-2^{31} \sim (2^{31}-1)$ ，有三种形式：

- 1) 十进制（默认方式） 43 1345 87654
- 2) 八进制 以0开头 043, 056, 011
- 3) 十六进制 以0x开头 0x12 0xa3 0x34 0xdf
(举例说明)

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

```
void main(void)
```

```
{
```

```
    int int10,int8,int16; //定义3个整型变量
```

```
    int10=10;           //默认为十进制
```

```
    int8=010;           //八进制
```

```
    int16=0x10;         //十六进制
```

```
    cout<<"int10="<<int10<<endl;
```

```
    cout<<"int8="<<int8<<endl;
```

```
    cout<<"int16="<<int16<<endl;
```

```
}
```

输出

int10=10

int8=8

int16=16

整型变量:

分为有符号型与无符号型。

有符号型:

short 在内存中占两个字节, 范围为 $-2^{15} \sim (2^{15}-1)$

int 在内存中占四个字节, 范围为 $-2^{31} \sim (2^{31}-1)$

long在内存中占四个字节, 范围为 $-2^{31} \sim 2^{31}-1$

无符号型: **最高位不表示符号位**

unsigned short 在内存中占两个字节, 范围为 $0 \sim 2^{16}-1$

unsigned int 在内存中占四个字节, 范围为 $0 \sim 2^{32}-1$

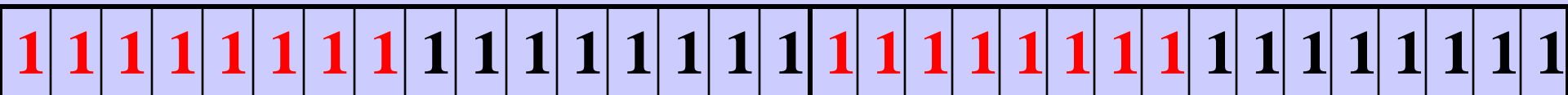
unsigned long在内存中占四个字节, 范围为 $0 \sim 2^{32}-1$

1) **整型常量**亦有长短之分，常量中无unsigned型，但一个非负的整型常量可以赋给unsigned型的变量。

2) 若一个常量定义为长整型数，则在其后加l或L进行区分。

如：32l 32L 564L等，内存为其分配四个字节存储。

一个数在内存中为



当这个数为有符号数时，是-1；为无符号数时，是 $2^{32}-1$

内存中的数是以**补码**的形式存放的。（举例说明）

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

```
void main()
```

```
{ unsigned short a;
```

```
  short int b= -1;
```

```
  a=b;
```

```
  cout<<"a="<<a<<endl;
```

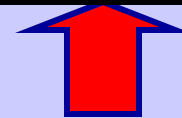
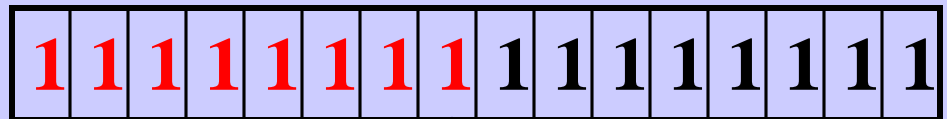
unsigned short a;

```
}
```

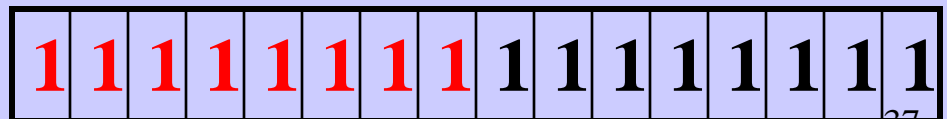
结果: 65535

不同类型的整型数据间的赋值归根到底就是一条：按存储单元中的存储形式直接传送。

a



b



实型数据

实型数又称浮点数，有两种表示方式：

1) 十进制形式： 23.0 24.5 3.56789

2) 指数形式： 23E1 145e-1 356789e1 **e前有数字，后面必须是整数。**

实型变量分单精度 float 和双精度 double 两种形式：

float: 占四个字节，提供7~8位有效数字。

double: 占八个字节，提供15~16位有效数字。

举例说明

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

```
void main(void)
```

```
{    float  a, b;
```

```
    double  c, d;
```

```
    a=0.01;
```

```
    b=3.45678e-2;
```

```
    c=3.45678e-2;
```

```
    d=9.7654e-5;
```

```
    cout<<"a="<<a<<"\t"<<"b="<<b<<endl;
```

```
    cout<<"c="<<c<<"\t"<<"d="<<d<<endl;
```

```
}
```

a=0.01 b=0.0345678

c=0.0345678 d=9.7654e-005

Press any key to continue

实数是既有整数又有小数的数。

实数可以表示成： $N = S \times R^J$

S 称为尾数，尾数决定有效数字，即数字的精度。

J 表示指数（阶码）。

R 是基数，可取2，4，8，16等，对具体机器而言，基数取好后，就不能再变了。

数有正有负，所以设置数符；阶码亦有正负，所以设置阶符

如果为实数，则用浮点数的形式在内存存储，表示如下：

J_t	J	S_f	S
阶符	阶码	数符	尾数

一般用4个字节表示一个浮点数，也有用8个字节表示的。

字长一定，尾数越多，精度越高；阶码越多，范围越大。

当计算机中出现小于机器所能表示的最小数时，机器只能当零来处理，当出现超过机器所能表示的最大数时，出现溢出现象，一旦出现溢出，就会停止运算。定点数，浮点数均会出现溢出现象。

字符型数据 (char)

字符型数据实际上是作为**整型数据**在内存中存储的。

计算机是以字符编码的形式处理字符的，因此，我们在计算机内部是以**ASCII码**的形式表示所有字符的。所以7位二进制数即可表示出一个字符，**我们用一个字节的容量（8位）存储一个字符。**

例如：字符A的ASCII码为0x41或65，在内存中表示为：

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

在程序中表示为：

`char grade ;`//定义一个字符型的变量空间(1个字节)

`grade='A';` //必须用 ‘ ’ 表示，否则易与标识符混同

‘ ’内括起来的字符表示该字符的ASCII码。

进一步，由于在内存中的形式与整型数据相同，所以，可以直接用其整型值给变量赋值。

```
char grade;
```

```
grade=65;
```

以下的赋值形式均是等同的。

```
grade='A';      grade=65;      grade=0x41;      grade=0101;
```

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

```
void main(void)
```

```
{
```

```
    char a,b;
```

```
    a='A'; //输入ASCII码
```

```
    b=65; //输入十进制数
```

```
    cout<<"a="<<a<<endl;
```

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

```
}
```

输出：

a=A

b=A

即在内存中的表示均是相同的

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

非打印字符

有些ASCII的字符代表某些操作，不能打印出来，如回车、退格等，可用两种方式表示这些字符。

1) 用ASCII码的形式 **char re=13;**

2) 用转义字符 **char re='\n'; (p15)**

转义字符	含 义	ASCII代码
\a	响铃	7
\n	换行，将当前位置移到下一行开头	10
\t	水平制表（跳到下一个tab位置）	9
\b	退格，将当前位置移到前一个	8
\r	回车，将当前位置移到本行开头	13
\f	换页，将当前位置移到下页开头	12
\v	竖向跳格	8
\\	反斜杠字符“\”	92
\'	单引号（撇号）字符	39
\"	双引号字符	34
\0	空字符	0
\ddd	1到3位8进制数所代表的字符	
\xhh	1到2位16进制数所代表的字符	

转义字符虽然包含2个或多个字符，但它只代表一个字符。编译系统在见到字符“\”时，会接着找它后面的字符，把它处理成一个字符，在内存中只占一个字节。

典型转义字符：

‘\n’换行 ‘\b’退格 ‘\t’ 下一个输出
区

若输出中包含这些特定格式，则再加一个\
输出 `c:\tc\tc` 表示为 `cout<<"c:\\tc\\tc";`

可以用转义字符表示任一个ASCII字符 ‘\ddd’
(八进制) ‘\xhh’ (十六进制)

‘\101’ ‘\x41’ ‘\x61’ ‘\141’

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

```
void main(void)
```

```
{
```

```
    char c1,c2,c3,c4;
```

```
    char n1,n2;
```

```
    c1='a';    //字符常量
```

```
    c2=97;    //十进制
```

```
    c3='\x61'; //转义字符
```

```
    c4=0141;  //八进制
```

```
    cout<<"c1="<<c1<<"\t"<<"c2="<<c2<<endl;
```

```
    cout<<"c3="<<c3<<"\t"<<"c4="<<c4<<endl;
```

```
    n1='\n';    //转义字符：回车
```

```
    n2='\t';    //转义字符：下一个输出区(Tab)
```

```
    cout<<"使用转义字符\n";
```

```
    cout<<"c1="<<c1<<n2<<"c2="<<c2<<n1;
```

```
    cout<<"c3="<<c3<<n2<<"c4="<<c4<<n1;
```

```
}
```

输出:

c1=a c2=a

c3=a c4=a

使用转义字符

c1=a c2=a

c3=a c4=a

字符串常量:

用" "表示, 在内存中顺序存放, 以'\0'结束。

如: "CHINA"

C	H	I	N	A	\0
---	---	---	---	---	----

实际上内存是对应字符的ASCII码形式

0x43	0x48	0x49	0x55	0x41	\0
------	------	------	------	------	----

01000011	01001000	01001001	01010101	01000001	00000000
----------	----------	----------	----------	----------	----------

'a'在内存中占一个字节

a

"a"占两个字节

a	\0
---	----

01100001

01100001	00000000
----------	----------

标识符常量

在C++中有二种方法定义标识符常量，一种是使用编译预处理指令；另一种是使用C++的常量说明符**const**。

例如：

```
#define PRICE 30
```

//在程序中凡是出现**PRICE**均用**30**替代

```
#define PI 3.1415926
```

```
#define S "China"
```

```
const float pi=3.1415926; //将变量pi定义为常量
```

（举例说明）

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

```
#define PI 3.14156
```

```
#define S "China"
```

```
void main(void)
```

```
{
```

```
    const float pi=3.14156;    //变量作为常量使用
```

```
    cout<<"PI="<<PI<<endl;
```

```
    cout<<"10*PI="<<10*PI<<endl;
```

```
    cout<<S<<endl;
```

```
// PI=PI+3;
```

```
// pi=pi+4;
```

```
    cout<<"PI="<<PI<<endl;
```

```
    cout<<"pi="<<pi<<endl;
```

```
}
```

输出:

PI=3.14156

10*PI=31.4156

China

PI=3.14156

pi=3.14156

下列常量的表示在C++中是否合法？若不合法，指出原因；若合法，则指出常量的数据类型。

32767 35u 1.25e3.43L 0.0086e-32

‘\87’ “Computer System” “a” ‘a’ ‘\96\45’

-0 +0 .5 -.567

变量

- 1) 在程序的执行过程中，其值可以改变的量称为变量。
- 2) 变量名必须用标识符来标识。
- 3) 变量根据其取值的不同值域，分为不同类型的变量：整型变量、实型变量、字符型变量、构造型变量、指针型变量等等。

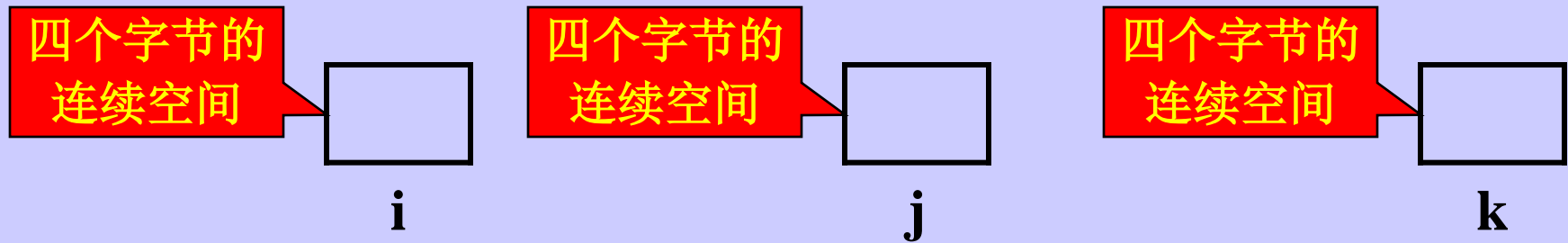
4) 对于任一变量，编译程序要为其分配若干个字节（连续的）的内存单元，以便保存变量的取值。

5) 当要改变一个变量的值时，就是把变量的新的取值存放到为该变量所分配的内存单元中；用到一个变量的值时，就是从该内存单元中取出数据。

6) 不管什么类型的变量，通常均是变量的说明在前，使用变量在后。

int i, j, k;

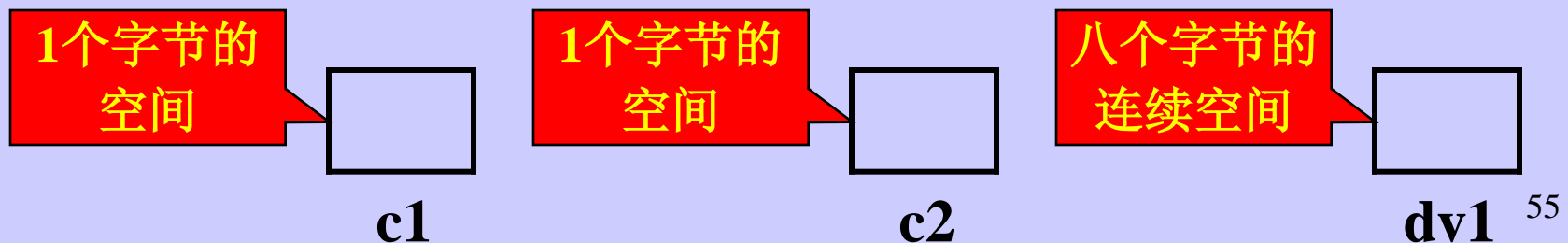
//定义了三个整型变量i,j,k



开辟空间后, 空间中为随机值

char c1,c2; //说明了二个字符型变量c1,c2

double dv1; //说明了一个双精度型变量dv1

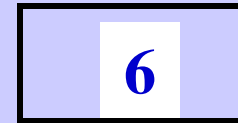


变量赋初值

在定义变量的**同时**给变量赋值，即在内存中开辟出一个空间后马上给此空间赋值。

但这个空间的值并不是固定不变的，**在程序的运行中一样可以改变。**

```
int  a=4; //定义语句，在开辟空间后马上为空间赋值  
a=6;     //重新为该空间赋值
```



a

```
char  a='\x64', b='d';
```

```
int  a1=6, a2=98;
```

```
a='A'; b='\n';
```

```
a1=011; a2=121;
```


算术运算符和算术表达式

一、算术运算符和算术表达式

$+$ $-$ $*$ $/$ $\%$

用算术运算符连接起来的式子是算术表达式

两个整数相除结果为整数 $1/2=0$ $5/2=2$

整数才可求余，余数的符号与左边数的符号相同。

$3\%2=1$ $-3\%2=-1$ $3\%-2=1$ $-3\%-2=-1$ $8\%4=0$

二、优先级与结合性

$()$ $*$ $/$ $\%$ $+$ $-$

三、强制转换类型

(类型名) (表达式)

(double) a

(int) (x+y)

(int) 6.2%4=2

在强制类型运算后原变量不变，但得到一个所需类型的中间变量。

如：int x;

float y=5.8;

x=(int)y;

x=5

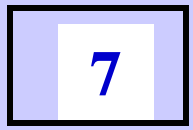
y=5.8

y的值没有改变，仍是单精度浮点型

四、自增、自减运算符 (难点)

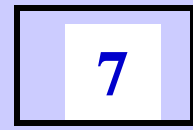
++ --

i=6; i++; i=i+1 i=7



i

++i; i=i+1 i=7



i

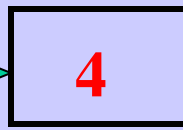
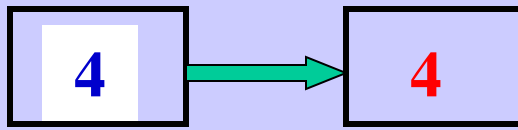
i=6; i--; i=i-1 i=5

--i; i=i-1 i=5

int i, j;

i=3;

j = ++i;

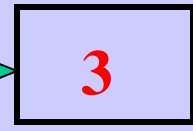
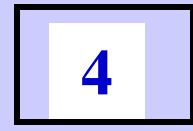


j

int i, j;

i=3;

j = i++;



j

i=4 j=4

++在前, 先运算, 后赋值

i=4 j=3

++在后, 先赋值, 后运算

1) 自增、自减运算符只能用于变量，不可用于常量和表达式

因为表达式在内存内没有具体空间，常量所占的空间不能重新赋值

$3++$ $(x+y)++$ $(-i)++$

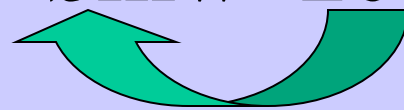
2) 结合方式自右至左，优先级最高，向右取最大

$-i++$ $-(i++)$ $i+++j$ $(i++) + j$

若 $i=3, j=2$ $(i++) + j$ 等于 **5** **$i=4, j=2$**

赋值运算符和赋值表达式

bmw=2002



"=" **左边** 必须是变量名。

若 “=” 两边变量类型不同，在赋值时要进行 **类型转换**。

转换原则：根据左边变量的类型转换。

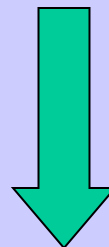
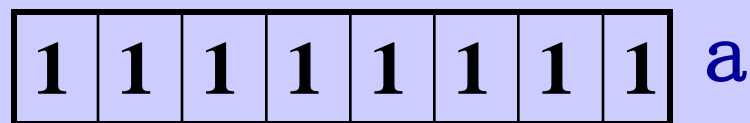
少字节→多字节

1) 若多字节变量为**unsigned**,则转换后多余字节补零。

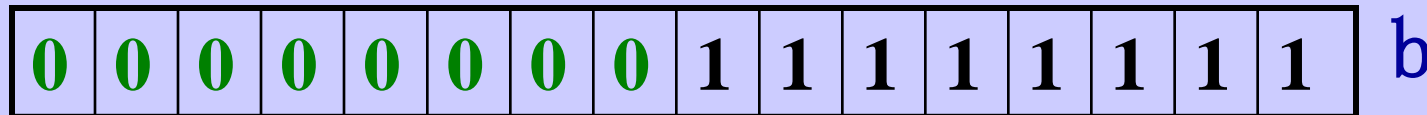
```
short int a=-1;
```

```
unsigned long b;
```

```
b=a;
```



unsigned

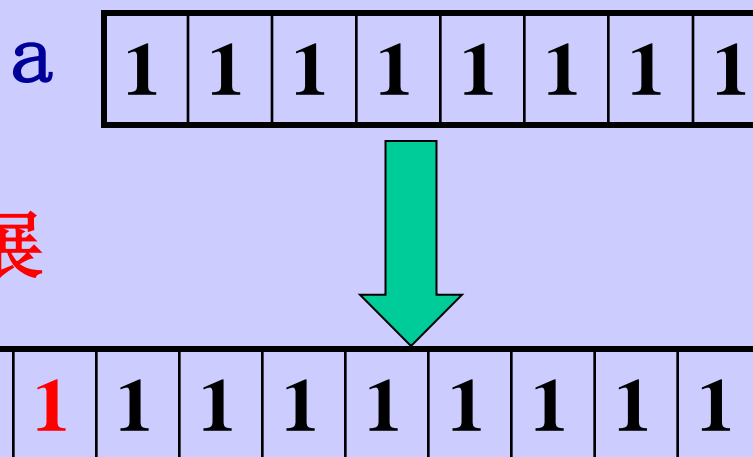


2) 若多字节变量为有符号型，则转换后扩展少字节的最高位。

```
short int a=-1;
```

```
long b;
```

```
b=a; 有符号型，符号扩展
```



转换后，数据的符号不变。

多字节→少字节

低位照搬

int a=-1;

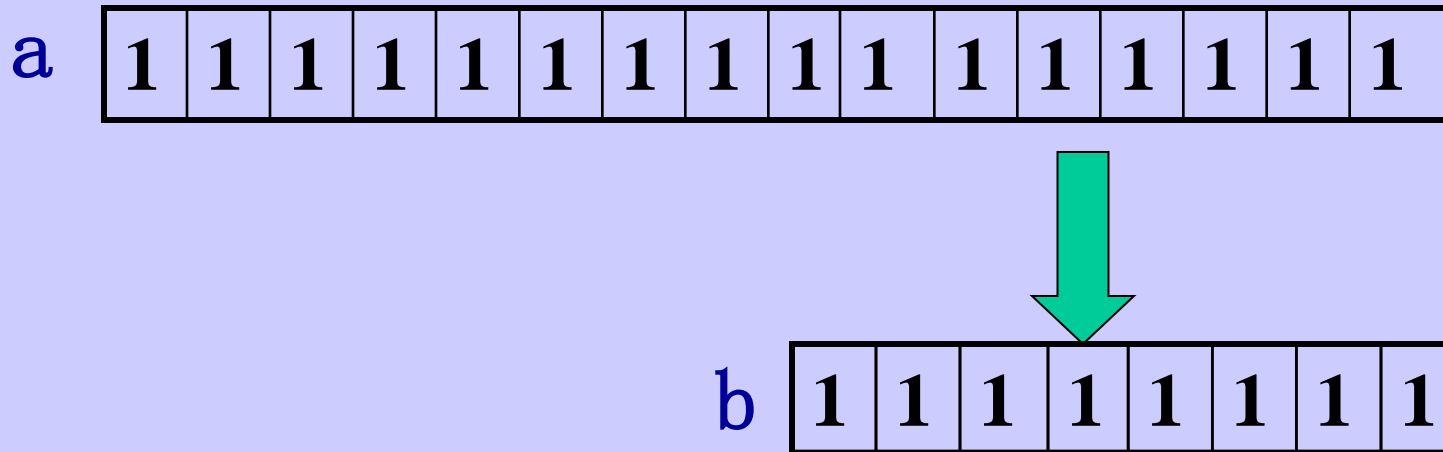
short int b;

b=a; **b=-1**

int a=65535;

short int b;

b=a; **b=-1**



复合的赋值运算符

a+=3 a=a+3

x*=y+3 x=x*(y+3)

x/=x-4 x=x/(x-4)

x+=y x=x+y

i+=j-- i=i+(j--)


赋值表达式

a=b=5 ; b=5 a=5

"="的结合性为**自右至左**

a=12; a+=a-=a*a;

$$\mathbf{a=a-(a * a) =12-(12*12)=-132}$$


$$\mathbf{a=a+(-132)=-132-132=-264}$$


a

-264

关系运算符和关系表达式

关系运算符（比较运算）

< > <= >= == !=

1. == 与 =

a=5; 赋值运算 a==5; 判断是否相等

2. < > <= >= 的优先级大于== !=

3. 算术运算符的优先级大于关系运算符的优先级

关系表达式：用关系运算符将表达式连接起来称为关系表达式。其值非真即假。在C++语言中，用非0代表真，用0表示假。关系表达式的结果只有两个，真为1，假为0。

a=2 b=3 c=4

a>2 0

a>b+c 0

a==2 1

a=='a' 0

a>'a' 0

b==2 1

'a'>'A' 1

b=='a'+1 0

c-a==a 1

逻辑运算符

1. 运算符

与 **&&**

或 **||**

非 **!**

&&

||

!

A	B	结果
0	0	0
0	1	0
1	0	0
1	1	1

有0出0，全1出1

A,B同时成立

A	B	结果
0	0	0
0	1	1
1	0	1
1	1	1

有1出1，全0出0

A或B有一个成立

A	结果
0	1
1	0

有0出1，
有1出0

例如：两个条件：江苏籍 男生

江苏籍的男生

江苏籍&&男生

江苏籍的学生和所有男生

江苏籍||男生

非江苏籍的学生

! 江苏籍

注意：

1.优先级：!→&&→||

! → 算术 → 关系 → 逻辑 → 赋值 → 逗号

2.作为条件，所有非0值均为真；作为结果，只有0或1两种。

$5 > 3 \ \&\& \ 2 \ || \ 8 < 4 - !0$

3.不可写为 $1 < x < 10$ 应为: $1 < x \ \&\& \ x < 10$

4.当前面的表达式可以得出整个表达式的结果时，不必再求后面的表达式。

$a \ \&\& \ b \ \&\& \ c$ 当a为0时，表达式为0，不必求b与c。

$a \ || \ b \ || \ c$ 当a为1时，表达式为1，不必求b与c。

x=4 y=5

i= ++x= =5 || ++y= =6

x=5 y=5 i=1

i= x++= =5&& y++= =6

x=5 y=5 i=0

判断某年是否为闰年

1) 能被400整除 **year%400= =0**

2) 能被4整除，不能被100整除 (2200年不是)

year%4= =0&& year%100!=0

(year%400= =0) || (year%4= =0&&year%100!=0)

当c=4时，以下的值各多少？

(c=1)&&(c=3)&&(c=5) **1**

(c==1)||(c==2) || (c==5) **0**

(c!=2) && (c!=4) &&(c>=1)&&(c<=5) **0**

sizeof（）运算符

sizeof()运算符是一个单目运算符，用于计算某一个操作数类型的字节数。其格式为：

sizeof（<类型>）

sizeof（int） //其值为4

sizeof（float） //其值为4

sizeof（double） //其值为8

sizeof（char） //其值为1

逗号运算符和逗号表达式

表达式1, 表达式2, 表达式3, ..., 表达式n

顺序求解, 结果为最后一个表达式的值, 并且优先级最低。

$a=(3+4, 5*6, 2+1);$ $a=3$

$a=3*3, a+6, a+7;$ 16 $a=9$

$(a=3*5, a*4), a+5$ 20 $a=15$

下列语句中表达式中i, j的值各为多少

1、 int i=0, j=0; **i=3, j=1** 2、 int i=0, j=1;

i=3, (j++)+i ; **i=3, j=3** i+=j*=3;

3、 int i=1, j=0; 4、 int i=1, j=1;

j=i=((i=3)*2); i+=j+=2;

i=6, j=6

i=4, j=3

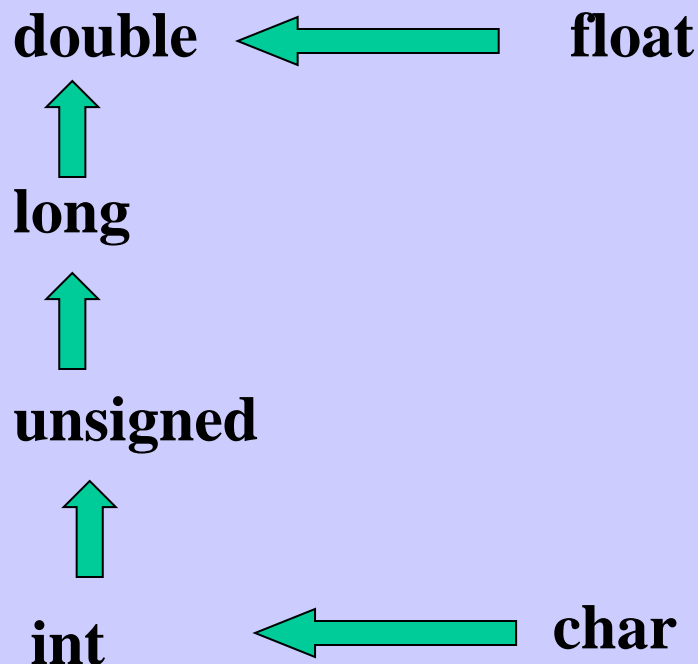
各类数值型数据间的混合运算

整型、实型、字符型数据间可以混合运算。

在进行运算时，不同类型的数据要先转换成同一类型的数据再进行运算。

转换规则如下：

10+'a'+1.5-87.65*'b'



第三章 简单的输入输出

输入语句：cin

程序在执行期间，接收外部信息的操作称为程序的输入；而把程序向外部发送信息的操作称为程序的输出。在C++中没有专门的输入输出语句，所有输入输出是通过**输入输出流**来实现的。

要使用C++提供的输入输出时，必须在程序的开头增加一行：

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

即包含输入输出流的头文件“**iostream.h**”。有关包含文件的作用，在编译预处理部分（第五章）作详细介绍。

输入十进制整数和实数

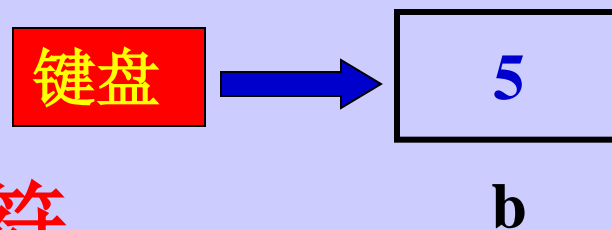
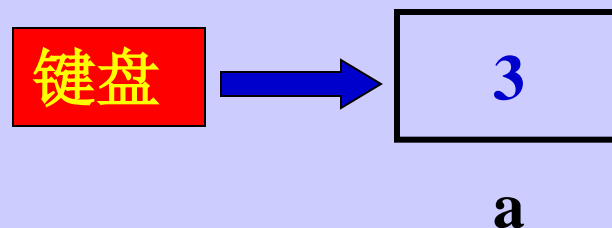
`cin >> <变量名1> 《 >> <变量名2> 》`（举例说明）

`int a,b;`

`cin>>a>>b;` //程序运行至此停下，等待从键盘输入变量值

键盘输入：3 5<CR>

或：3<CR> 5<CR> 均可。



输入语句自动过滤空白字符。

浮点型数据同整型数据一样。

```
float  c,d;
```

```
cin>>c>>d;
```

```
char  ch1,ch2;
```

```
cin>>ch1>>ch2;
```

若输入： ab<CR> 则ch1为a, ch2为b。

若输入： a b<CR> 则ch1为a, ch2为b。

字符型变量过滤空白字符。cin格式过滤空白字符

```
float a;
```

输入: 34 5.678 1a b<CR>

```
int i1,i2;
```

i1:34 a:5.578 i2:1

```
char ch1,ch2;
```

```
cin>>i1>>a>>i2>>ch1>>ch2; ch1:a ch2:b
```

在缺省的情况下，**cin**自动跳过输入的空格，换言之，**cin**不能将输入的空格赋给字符型变量，同样地，回车键也是作为输入字符之间的分隔符，**也不能将输入的回车键字符赋给字符型变量。**

若要把从键盘上输入的每一个字符，包括**空格和回车键**都作为一个输入字符赋给字符型变量时，必须使用函数**cin.get()**。其格式为：

cin.get(<字符型变量>);

cin.get()从输入行中取出一个字符，并将它赋给字符型变量。这个语句一次只能从输入行中提取一个字符。

char c1;

cin.get(c1);

```
char ch1,ch2,ch3;
```

```
cin.get(ch1);
```

```
cin.get(ch2);
```

```
cin.get(ch3);
```

则: **ch1:A**

ch2:空格

ch3:B

输入: **A B<CR>**

并且在输入缓冲区中保留回车键。

空格的ASCII码为32

ch2

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

输入十六进制或八进制数据

在缺省的情况下，系统约定输入的整型数是十进制数据。当要求按八进制或十六进制输入数据时，在cin中必须指明相应的数据类型：**hex**为十六进制；**oct**为八进制；**dec**为十进制。

```
int i,j,k,l;
```

```
cin>>hex>>i;           //指明输入为十六进制数
```

```
cin>>oct>>j;           //指明输入为八进制数
```

```
cin>>k;                 //输入仍为八进制数
```

```
cin>>dec>>l;           //指明输入为十进制数
```

当执行到语句cin时，若输入的数据为：

11 11 12 12<CR>

结果： i:17 j:9 k:10 l:12

使用非十进制数输入时，要注意以下几点：

- 1、八进制或十六进制数的输入，只能适用于**整型变量**，不适用于字符型变量，实型变量。
- 2、当在**cin**中指明使用的数制输入后，**则所指明的数制一直有效，直到在接着的cin中指明输入时所使用的另一数制为止**。如上例中，输入k的值时，仍为八进制。

3、输入数据的**格式、个数和类型**必须与**cin**中所列举的变量类型**一一对应**。一旦输入出错，不仅使当前的输入数据不正确，而且使得后面的提取数据也不正确。

```
int a, b;
```

```
cin>>a,b;
```

```
cin>>a>>b;
```

```
cin>>a b;
```

```
cin>>ab;
```

输出数据 **cout**

与输入**cin**对应的输出是**cout**输出流。

当要输出一个表达式的值时，可使用**cout**来实现，其一般格式为：

cout << <表达式> 《<< <表达式>.....》;

其中运算符“<<”称为插入运算符，它将紧跟其后的表达式的值，输出到显示器**当前光标**的位置。

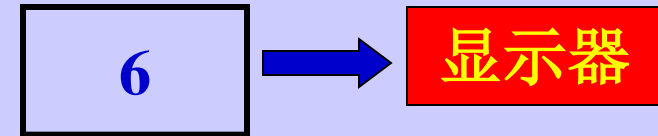
```
int a=6;
```

```
float f1=12.4;
```

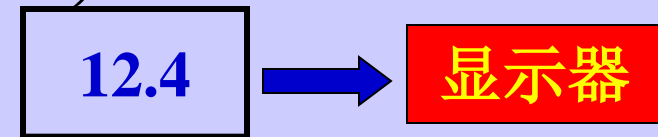
```
char s1[ ]="abcd";
```

```
cout<<a<<'\t'<<f1<<'\t'<<s1<<endl;
```

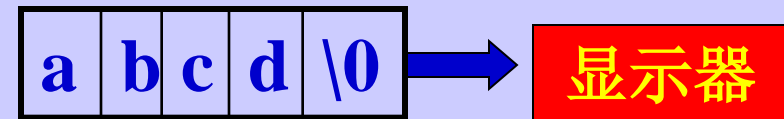
6	12.4	abcd



a



f1



s1

‘\t’为转义字符Tab

endl为回车或 ‘\n’

cout将双引号中的字符串常量按其原样输出

```
char ch1='a',ch2='b';
```

```
cout<<"c1="<<ch1<<"\t"<<"c2="<<ch2<<endl;
```

```
c1=a      c2=b<CR>
```

```
int i1=4,i2=5;
```

```
float a=3.5;
```

```
cout<<"a*i1="<<a*i1<<endl<<"a*i2="<<a*i2<<endl;
```

```
a*i1=14
```

```
a*i2=17.5
```

指定输出项占用的宽度：

在输出的数据项之间进行隔开的另一种办法是**指定输出项的宽度**。如上面的两个输出语句可改写为：

```
cout <<setw(6)<< i<<setw(10)<<j<<endl;
```

```
_____4_____12
```

```
cout << setw(5)<<m<<setw(10)<<j*k<<endl;
```

```
_____7_____24
```

其中**setw(6)**指明其后的**输出项占用的字符宽度为6**，即括号中的值指出紧跟其后的输出项占用的字符位置个数，**并向右对齐**。**setw**是“**set width**”的缩写。

使用**setw()**应注意以下三点：

1、在程序的开始位置必须包含头文件**iomanip.h**，即在程序的开头增加：

```
#include <iomanip.h>
```

2、括号中必须给出一个表达式（值为正整数），它指明紧跟其后输出项的宽度。

3、**该设置仅对其后的一个输出项有效**。一旦按指定的宽度输出其后的输出项后，又回到原来的缺省输出方式。

输出八、十六进制数和科学表示法的实数

对于整型数据可指定以十六进制或八进制输出，而对于实型数据可指定以科学表示法形式输出。例如，设有如下一个程序：

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

```
void main(void)
```

```
{ float x=3.14,y=100;
```

```
    cout.setf(ios::scientific,ios::floatfield);
```

```
    //表明浮点数用科学表示法输出
```

```
    cout << x<<'\t';
```

```
    cout <<y<<endl;
```

```
}
```

执行该程序后的输出为：

3.140000e+000

1.000000e+002

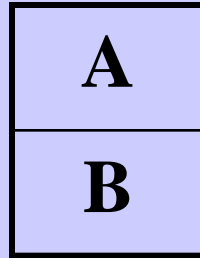
与cin中类同，当在cout中指明以一种进制输出整数时，对其后的输出均有效，直到指明又以另一种进制输出整型数据为止。对实数的输出，也是这样，一旦指明按科学表示法输出实数，则接着的输出均按科学表示法输出，直到指明以定点数输出为止。明确指定按定点数格式输出（缺省的输出方式）的语句为：

```
cout.setf(ios::fixed,ios::floatfield);
```

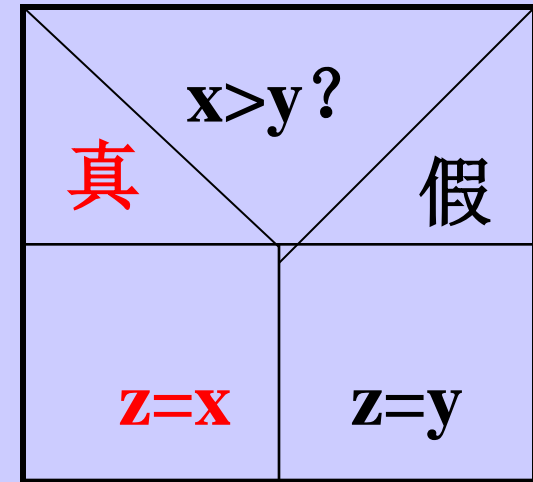
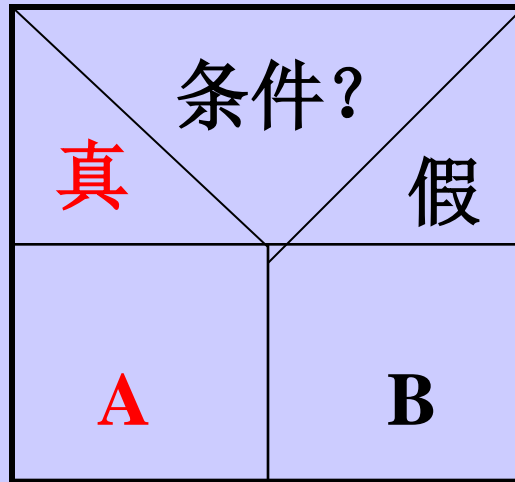

第四章 C++的流程控制语句

程序的三种基本结构

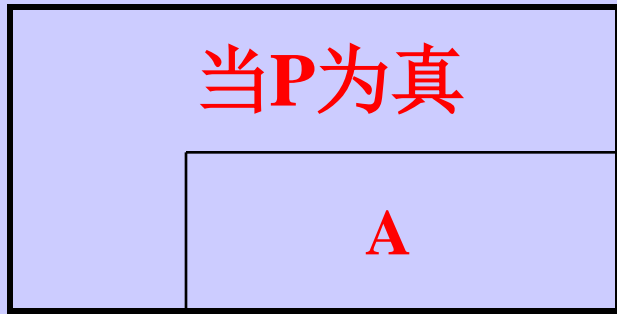
1、顺序



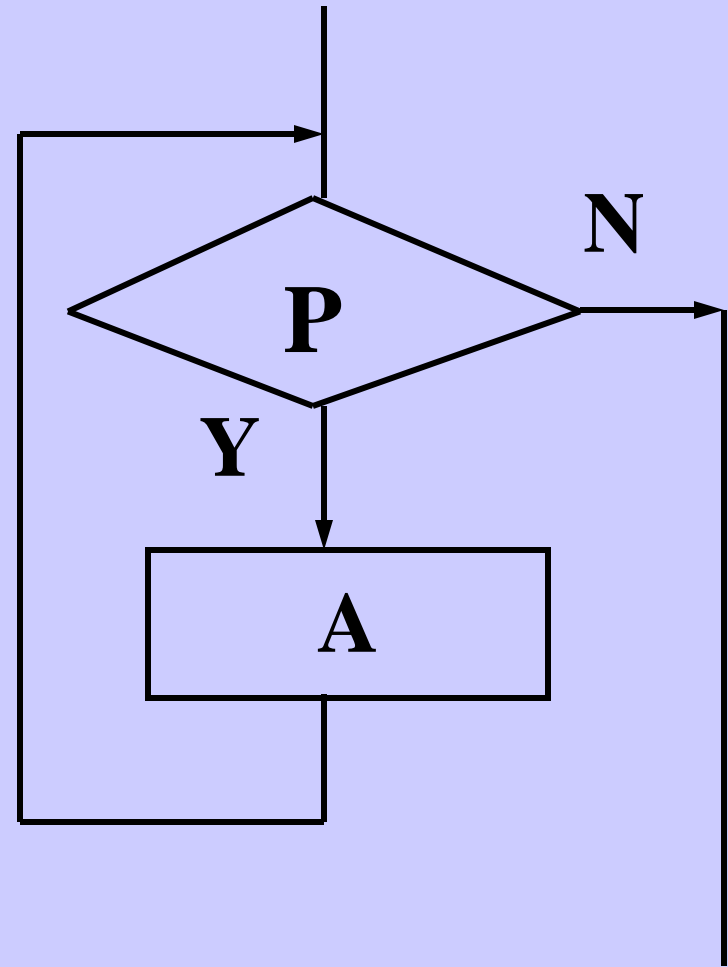
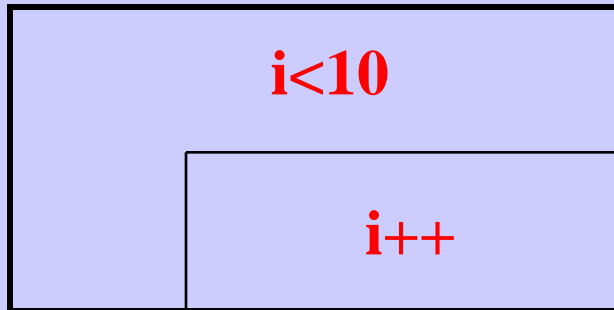
2、选择

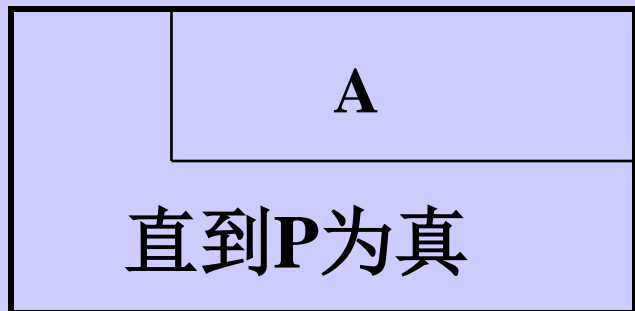


3、循环

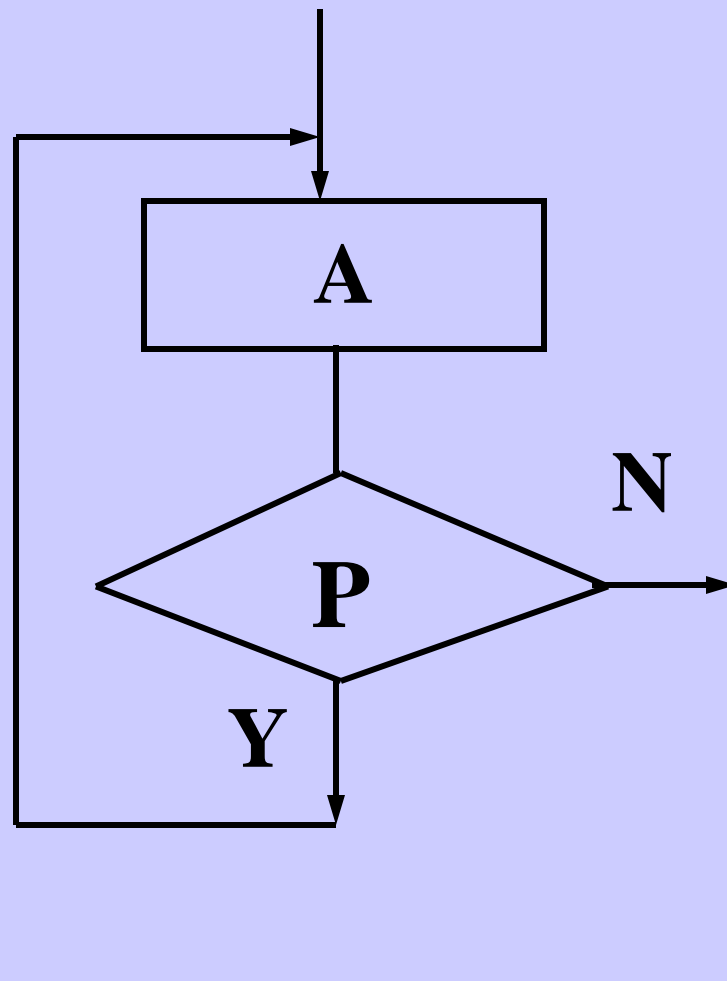
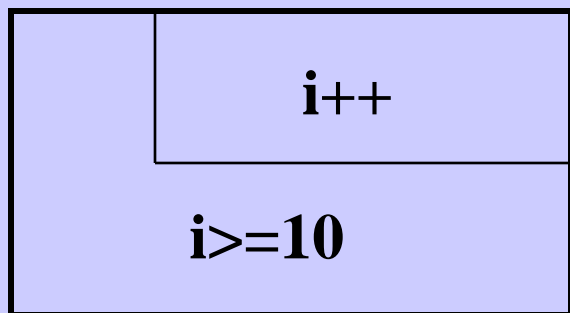


当型





直到型



if语句

判断选择语句，有三种形式：

1) if（表达式） 语句

if (a>b)

cout<<a;

2) if（表达式） 语句1

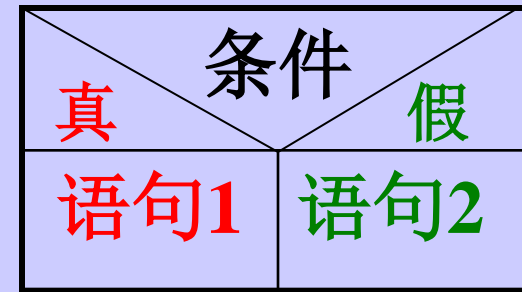
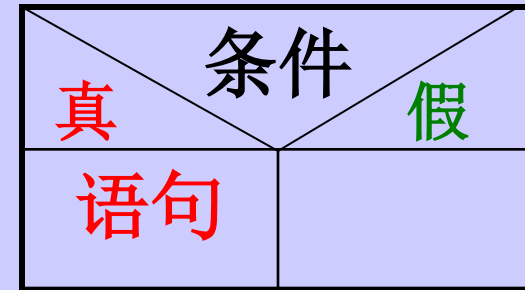
else 语句2

if (a>b)

cout<<a;

else

cout<<b;



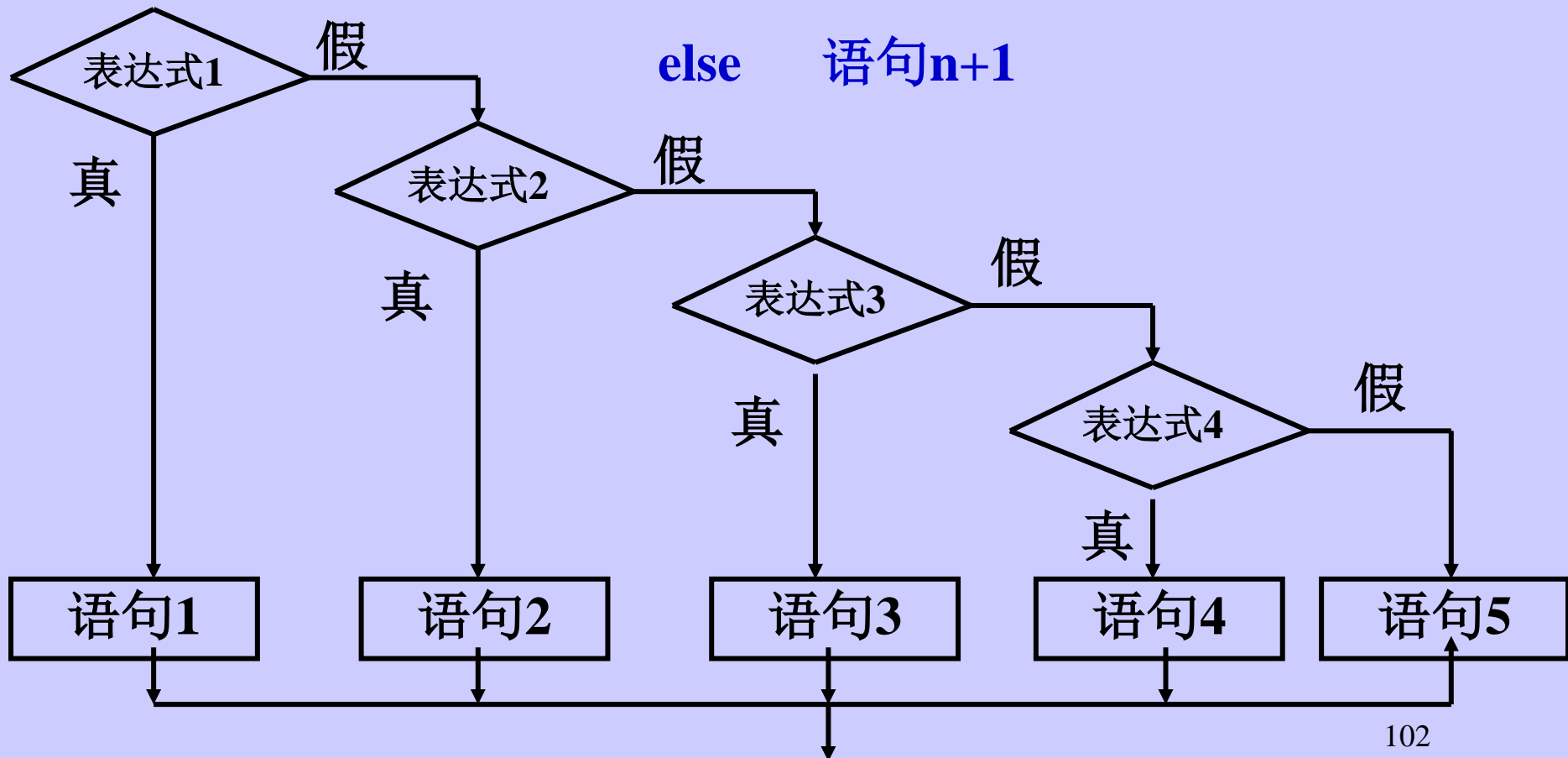
3) if (表达式1) 语句1

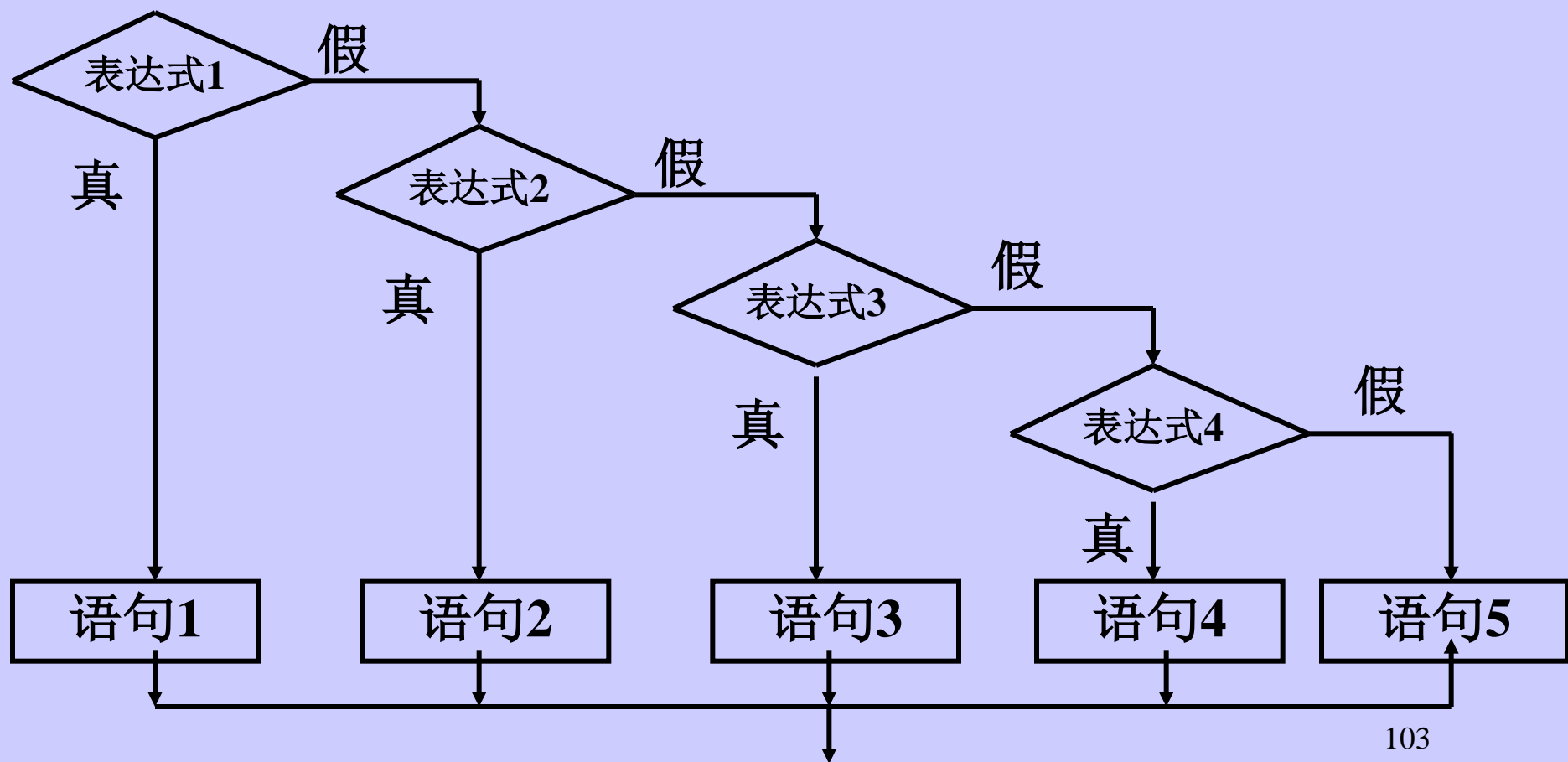
else if (表达式2) 语句2

.....

else if (表达式n) 语句n

else 语句n+1





注意：1) if 后可跟复合语句。

2) 注意 ; 的位置。

3) 注意多重 if else 的搭配。

```
if (a>b)
{
    a=1;
    b=0;
}
else
{
    a=0;
    b=1;
}
```

a>b	
真	假
a=1 b=0	a=0 b=1

if (i > j) i++;

i > j	
真	假
i++	

if (i > j); i++;

i > j	
真	假
i++	

if 总是与它上面最近的 **else** 配对，如要改变，用复合语句 { }。

注意书写格式，相互配对的语句要对齐。

例：输入两个实数，按代数值由小到大次序输出这两个数。

```
void main( void )
```

```
{ float a,b,t; //定义变量
```

```
cout<<" Input 2 Real Number:\n";//在屏幕上的提示信息
```

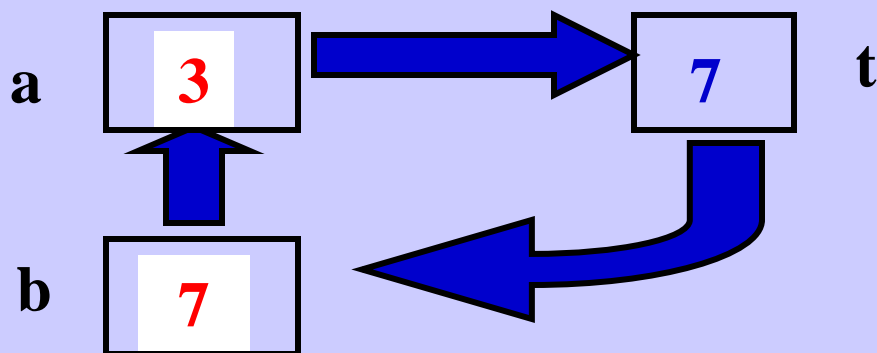
```
cin>>a>>b; //给变量赋值 a:7, b:3
```

```
if(a>b)
```

```
{ t=a; a=b; b=t; }//交换数据，用中间变量
```

```
cout<<a<<"\t"<<b<<endl;//输出变量
```

```
}
```



输出结果:

3 7

嵌套的条件语句（举例说明）

```
x=100; a=10; b=20; ok1=5; ok2=0;
```

```
if(a<b)
```

```
    if(b!=15)
```

```
        if(!ok1) x=1;
```

```
            else if (ok2) x=10; x=-1;
```

```
x=-1
```

条件运算符

是C++中的唯一的三目运算符。

表达式1? 表达式2 : 表达式3

表达式1	
真	假
表达式2	表达式3

max=a>b?a:b ; // 求a, b中的大者

当 a=2 b=1 a>b为真, 表达式的值等于a, max值为2

当 a=1 b=2 a>b为假, 表达式的值等于b, max值为2

注意:

1. 条件运算符的优先级比赋值运算符高

x=(x=3) ? x+2 : x-3 x=5

2. 结合方向自左至右 a>b?a:c>d?c:d

3. 三个表达式的类型可不同 z=a>b?'A':a+b

x=9, y=6, z=5;

x=((x+y)%z>=x%z+y%z)?1:0;

cout<<"x= "<<x<<endl;

x=0

x=1; y=2; z=3;

y=y+z=5

x+=y+=z;

x=x+5=6

cout<< (z+=x>y?x++:y++) <<endl;

9

void main(void)	x	y	z	输出
{ int x=1,y=2,z=3;				
x+=y+=z;	6	5	3	
cout<<x<y?y:x<<endl;	6	5	3	6
cout<<x<y?x++:y++<<endl;	6	6	3	5
cout<<x<<" "<<y<<endl;	6	6	3	6,6
cout<<z+=x>y?x++:y++<<endl;	6	7	9	9
cout<<y<<" "<<z<<endl;	6	7	9	7,9
x=3; y=z=4;	3	4	4	
cout<<(z>=y&&y==x)?1:0<<endl;	3	4	4	0
cout<<z>=y&&y>=x<<endl;	3	4	4	1
}				

执行以下程序段后，变量a,b,c的值分别是：

```
int  x=10, y=9;
```

```
int  a,b,c;
```

```
a=(--x==y++)?--x:++y;  x=8    y=10    a=8
```

```
b=x++;                  b=8    x=9
```

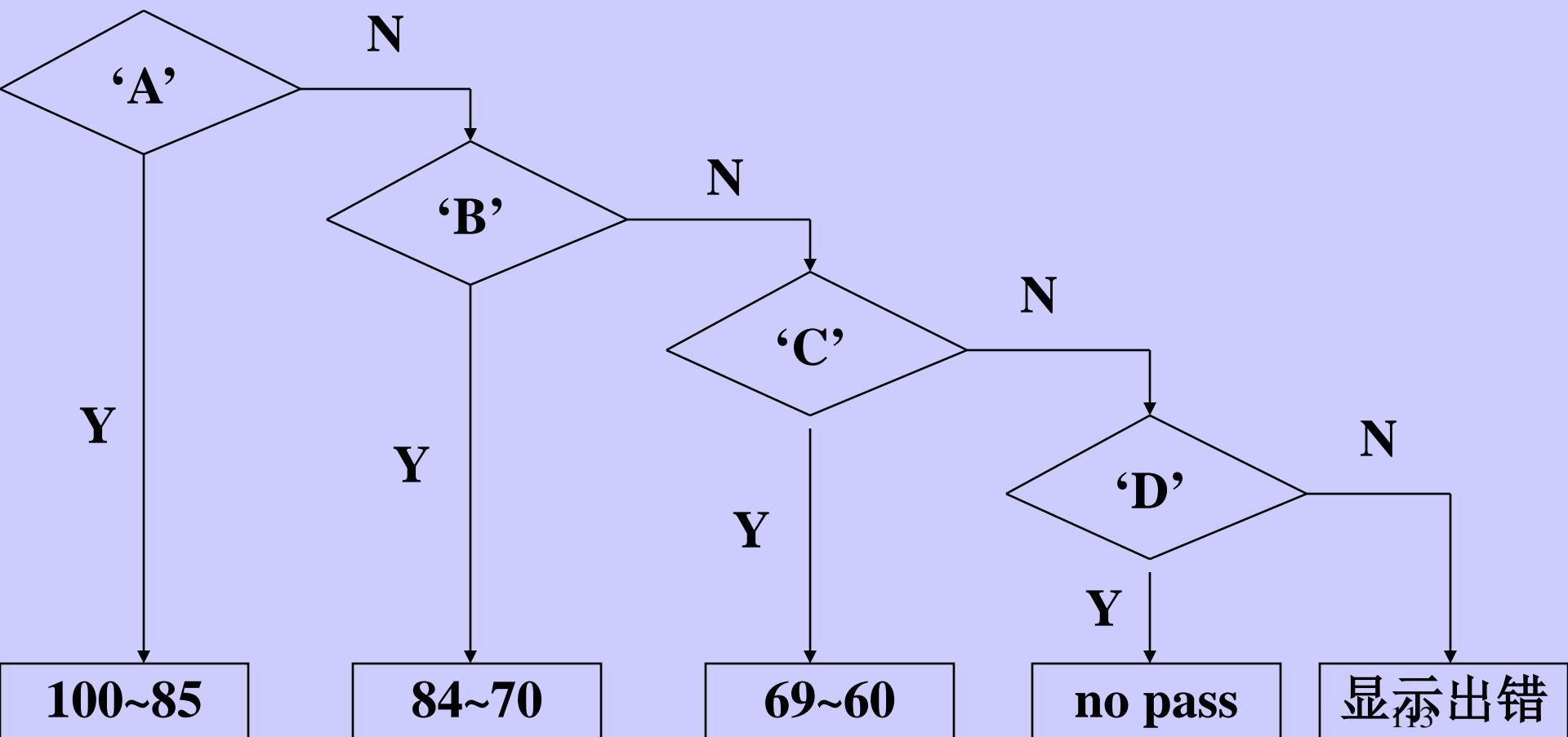
```
c=y;                    c=10
```

```
void main(void )  
  
{ int a=5,b=1,c=0;  
  
    if(a=b+c) cout<<"* * *\n";  
  
    else cout<<"$ $ $\n";  
  
}
```

* * *

switch语句

多分支选择语句。if语句只有两个分支，而实际问题中常常需要用到多分支的选择。如，成绩分为A(100~85)、B(84~70)、C(69~60)、D(60以下)等。



```
cin.get(grade);  
if(grade== 'A')  
    cout<<"100~85\n";  
else if (grade== 'B')  
    cout<<"84~70\n";  
else if (grade== 'C')  
    cout<<"69~60\n";  
else if (grade== 'D')  
    cout<<"no pass\n";  
else  
    cout<<"error\n";
```

switch(表达式)

```
{ case 常量表达式1: 语句1  
  case 常量表达式2: 语句2  
  ... ..  
  case 常量表达式n: 语句n  
  default: 语句n+1  
}
```

switch(grade)

```
{ case 'A': cout<<"100~85\n";  
  case 'B': cout<<"84~70\n";  
  case 'C': cout<<"69~60\n";  
  case 'D': cout<<"no pass\n";  
  default: cout<<"error\n";  
}
```

如果grade为 'A',则结果为

100~85

84~70

69~60

no pass

error

其流程为：先计算表达式的值，然后顺序地与case子句中所列出的各个常量进行比较，若表达式的值与常量中的值相等，就开始进入相应的case语句执行程序，遇到case和default也不再判断，直至switch语句结束。如果要使其在执行完相应的语句后中止执行下一语句，可以在语句后加break。

switch(grade)

```
{ case 'A': cout<<"100~85\n"; break;  
case 'B': cout<<"84~70\n"; break;  
case 'C': cout<<"69~60\n"; break;  
case 'D': cout<<"no pass\n"; break;  
default: cout<<"error\n";  
}
```

注意：

1、**switch**与**if**不同，它仅能判断一种逻辑关系，即表达式是否**等于**指定的常量，而 **if** 可以计算并判断各种表达式。

2、**case**子句后必须为常量，常常是整型和字符型。

3、**default**可以省略，这时，不满足条件什么也不执行。

4、**case**和**default**只起标号的作用，**顺序可以颠倒**，颠倒时注意后面的**break**语句。

5、多个**case**语句可以共用一组程序。

case 'A':

case 'B':

case 'C': cout<<"pass!\n";

```
void main(void )
```

```
{ int i=10;
```

```
    switch(i)
```

```
    { case 9: i++;
```

```
        case 10: i++; i=11
```

```
        case 11: i++; i=12
```

```
        default: i++; i=13
```

```
    }
```

```
    cout<<"i="<<i<<endl;
```

i=13

```
}
```



```

int x=1, y=0, a=0, b=0;
switch( x )
{ case 1:
    switch( y )
    { case 0: a++; break; a=1
      case 1: b++; break;
    }
case 2:
    a++; b++; break; a=2 b=1
case 3:
    a++; b++;
    a=2 b=1
}
cout<<"a="<<a<<"\t"<<"b="<<b<<endl;

```

有3个整数a, b, c，由键盘输入，输出其中最大的数。

while语句

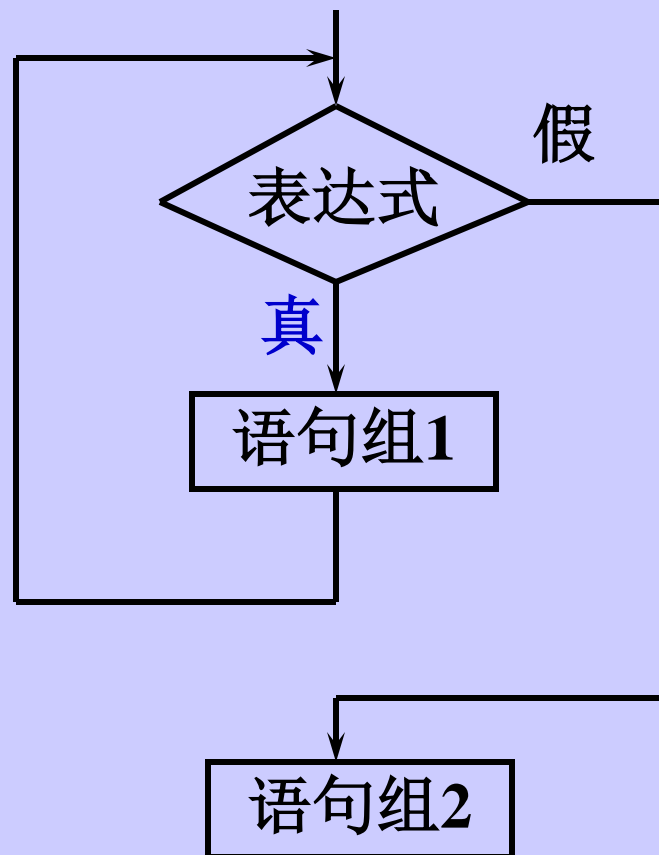
```
while ( 表达式 )  
    { 语句组1 }  
    { 语句组2 }
```

```
a=3;
```

```
while(a<100)
```

```
    a=a+5;
```

```
    cout<<"a="<<a;
```



当循环语句超过一条时，要用{ }将语句组组合在一起。

求 $1+2+3+\dots+100$

```
void main(void)
```

```
{ int i=1,sum=0; //定义变量，初始化
```

```
while(i<=100) //构造循环
```

```
{ sum=sum+i; //循环体，多次执行
```

```
    i=i+1;
```

```
}
```

```
cout<<"sum="<<sum<<endl; //输出结果
```

```
}
```

实际上是将i不停地累加到一起

5050

sum

101

i

循环结束!!

sum=5050

循环条件	初值	真	真	真	真	真	真	真	假
循环次数		1	2	3	4	99	100	101
sum	0	1	3	6	10			5050	
i	1	2	3	4	5		100	101	124

注意：

- 1、循环体如果为一个以上的语句，用{ }括起。
- 2、循环体内或表达式中必须有使循环结束的条件，**即一定有一个循环变量。**
- 3、**while**表达式可以成为语句，要特别小心。

k=2;

while(k!=0) cout<<k, k--;

cout<<endl;

k	2	1	0
循环条件	真	真	假
输出	2	1	回车

输出： 21

```
void main(void)
```

```
{  int  num=0;
```

```
    while(num<=2)
```

1

```
    {  num++;
```

2

```
        cout<<num<<endl;
```

3

```
    }
```

```
}
```

num	0	1	2	3
循环条件	真	真	真	假
输出	1<CR>	2<CR>	3<CR>	无

```
void main(void)
```

```
{  int y=10;
```

```
    while (y--);
```

```
    cout<<"y="<<y<<endl;
```

```
}
```

输出是什么？

循环几次？

y	10	9	1	0
条件	真	真	真	真	假
输出	无	无	无	无	-1

输出： y=-1

循环： 10次

k=10;

while(k=0)

k=k-1;

cout<< k;

k	10
表达式	0

输出： 0

以下语句，循环退出时x为多少？

x=10; while (x!=0) x--; **x=0**

x=10; while (x) x--; **x=0**

x=10; while(x--); **x=-1**

x=10; while(--x); **x=0**

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

```
void main()
```

```
{    char ch;
```

```
    while(cin.get(ch)&&ch!='\n')
```

```
    switch (ch-'2')
```

```
    { case 0:
```

```
        case 1: cout<<(char)(ch+4);
```

```
        case 2: cout<<(char)(ch+4); break;
```

```
        case 3: cout<<(char)(ch+3) ;
```

```
        default : cout<<(char)(ch+2); break;
```

```
    }
```

```
    cout<<endl;
```

```
}
```

从键盘输入2473<CR>, 则程序的输出结果是:

输出: 668977

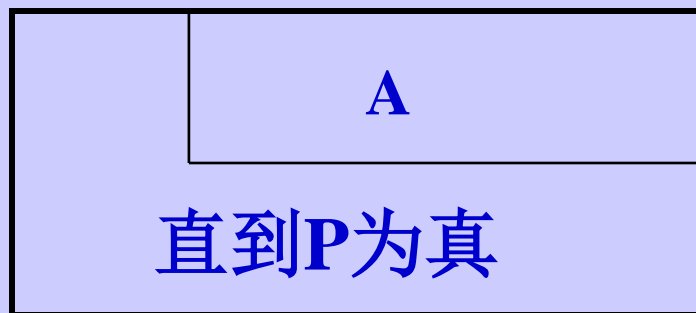
do—while语句

do

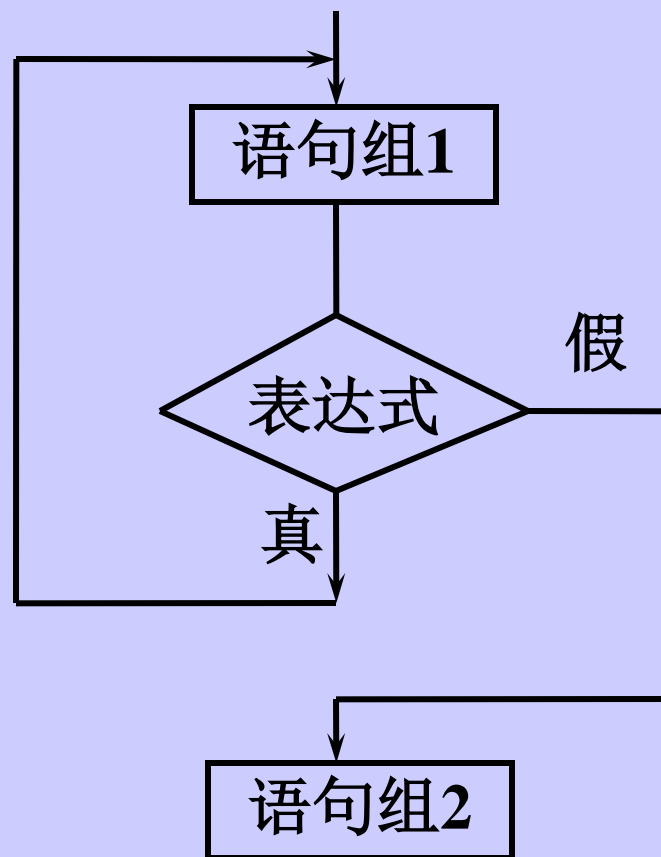
{ 语句组1 }

while (表达式) ;

{ 语句组2 }



直到型



求 $1+2+3+\dots+100$

```
void main(void)
```

```
{  int  i=1,sum=0; //定义变量，初始化
```

```
    do           //构造循环
```

```
    {  sum=sum+i; // 循环体，多次执行
```

```
        i=i+1;
```

```
    }while (i<=100);
```

```
    cout<<"sum="<<sum<<endl; //输出结果
```

```
}
```

注意：

do—while首先执行循环体，然后再判断表达式，至少执行一次循环体。当第一次循环表达式的值为真时，**while**与**do—while**的结果完全一样，否则结果不相同。

x=0,y=0;

do

{ y++; x*=x;

} while ((x>0)&&(y>5));

cout<<"y="<<y<<","<<"x="<<x<<endl;

y	0	1
x	0	0
条件		假

若为while循环，则
一次也不执行循环
体，输出为：

y=0, x=0

输出： y=1,x=0

s=7;

do

s-=2;

while(s!=0);

cout<<"s="<<s<<endl;

		第一次
s	7	5
表达式		N

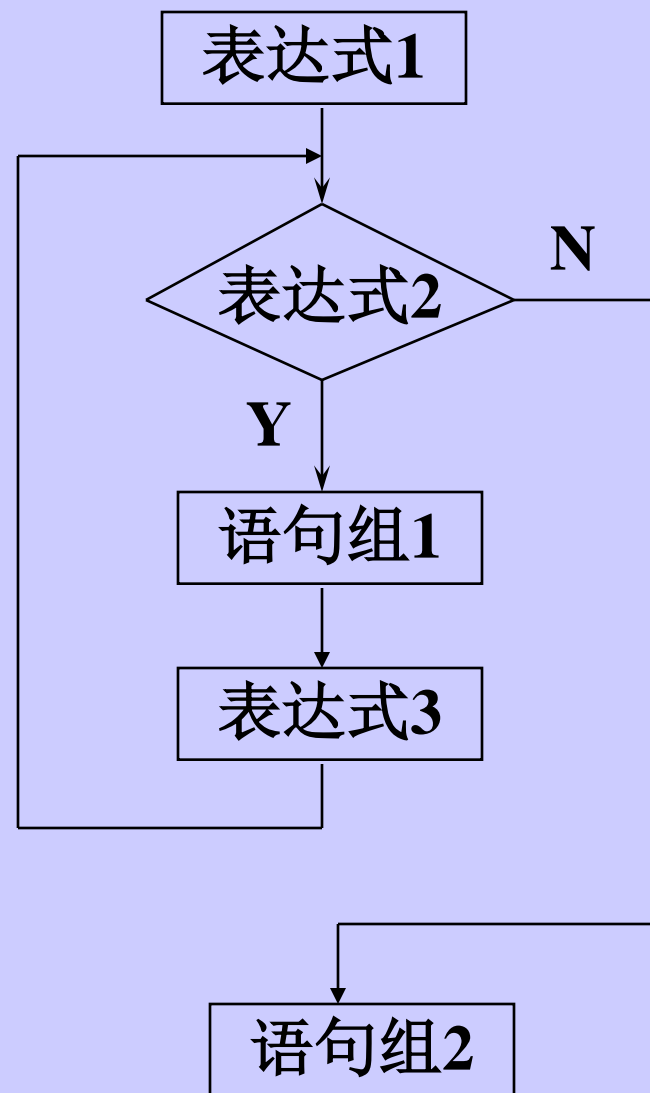
输出： s=5

for语句

for (表达式1; 表达式2; 表达式3)

{ 语句组1(循环体) }

{ 语句组2 }



for (循环变量赋初值; 循环结束条件; 循环变量增值)

求 $1+2+3+\dots+100$

```
void main(void)
```

```
{ int i, sum;
```

```
  for (i=1, sum=0; i<=100; i++)
```

```
    sum=sum+i;
```

```
    cout<<"sum="<<sum<<endl;
```

```
}
```

```
void main(void)
```

```
{ int i, sum;
```

```
  i=1; sum=0;
```

```
  while(i<=100)
```

```
  { sum=sum+i;
```

```
    i=i+1;
```

```
  }
```

```
  cout<<"sum="<<sum<<endl;
```

```
}
```

注意：

1、当型循环，条件测试是在循环开始时进行，有可能一次也进入不了循环体。

2、for语句中的三个表达式可以部分省略或全部省略，但；不能省略，若省略表达式2，则表示循环条件为真。

3、for语句中三个表达式可以是任何有效的C语言表达式。

```
void main(void)
```

```
{  char i, j ;
```

```
    for (i='a',j='z' ; i<j ; i++, j--)
```

```
        cout<<i<<j;
```

```
    cout<<endl;
```

输出: azbycx.....lomn

```
}
```

次数						
i	a	b	c	m	n
j	z	y	x	n	m
i<j	真	真	真	真	真	假
输出	az	by	cx	mn	CR

以下循环结果如何？

```
for ( i=0, k= -1; k=1; i++, k++)
```

```
    cout<<“***\n”;
```

以下循环最多执行 10 次，最少执行 1 次

```
for (i=0, x=0; i<=9&&x!=876 ; i++)
```

```
    cin>>x;
```

循环的嵌套

一个循环体内又包含另一个完整的循环体，称为循环的嵌套。

注意：

1、循环体内有多个语句要用 { } 括起来。

2、书写格式要清晰。

```
for ( ; ; )  
{ .....  
    for ( ; ; )  
    { .....  
    }  
}
```

```
void main(void)
```

```
{  int  i, j, k=0, m=0;
```

```
    for ( i=0; i<2; i++)
```

```
    {    for ( j=0; j<3; j++)
```

```
            k++;
```

```
        k- =j;
```

```
    }
```

```
    m=i+j;
```

```
    cout<<"k="<<k<<"", m="<< m<<endl;
```

```
}
```

输出:

k=0, m=5

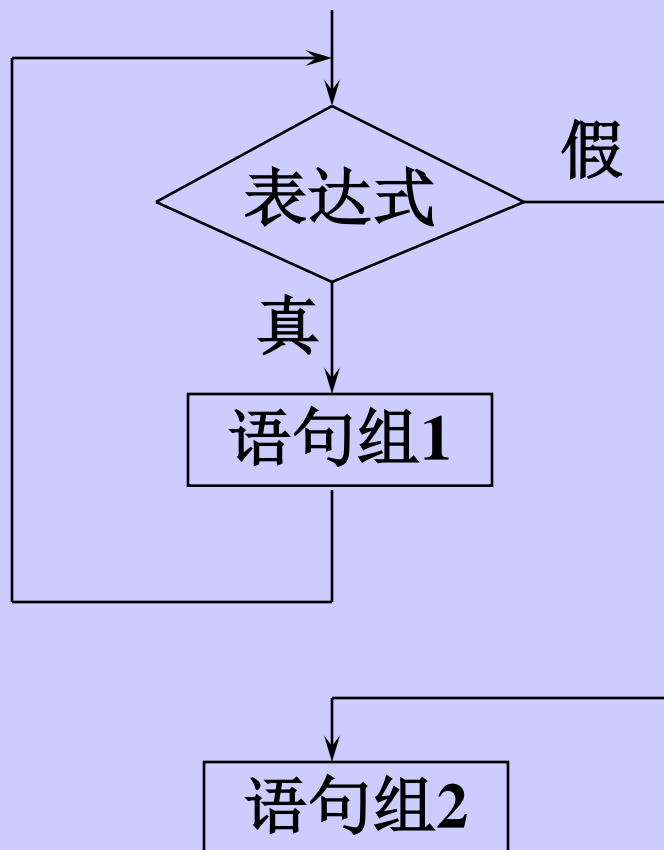
i	0				1				
i<2	真				真				
j	0	1	2	3	0	1	2	3	
j<3	真	真	真	假	真	真	真	假	
k	1	2	3	0	1	2	3	0	

几种循环的比较

while (表达式)

{ 语句组1 }

{ 语句组2 }

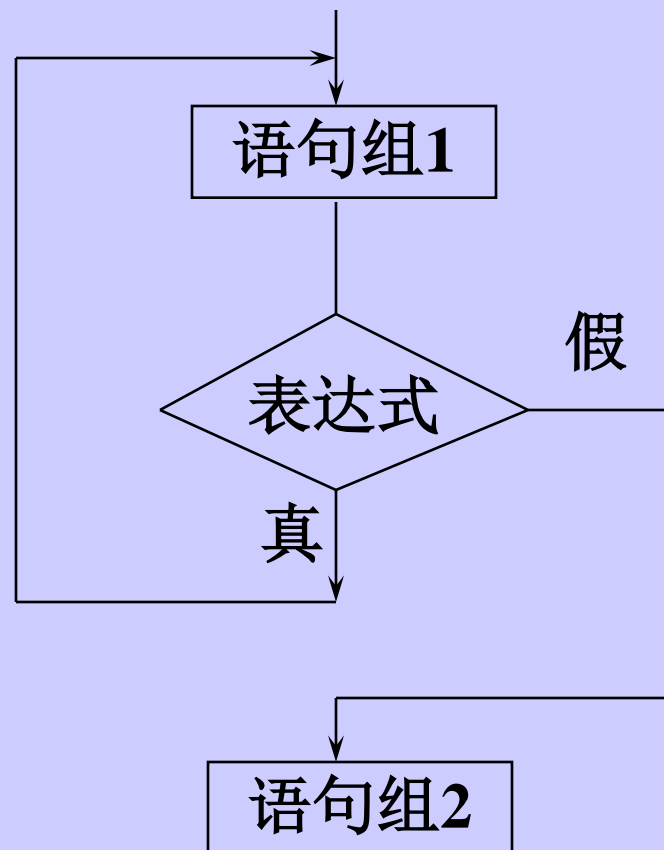


do

{ 语句组1 }

while (表达式) ;

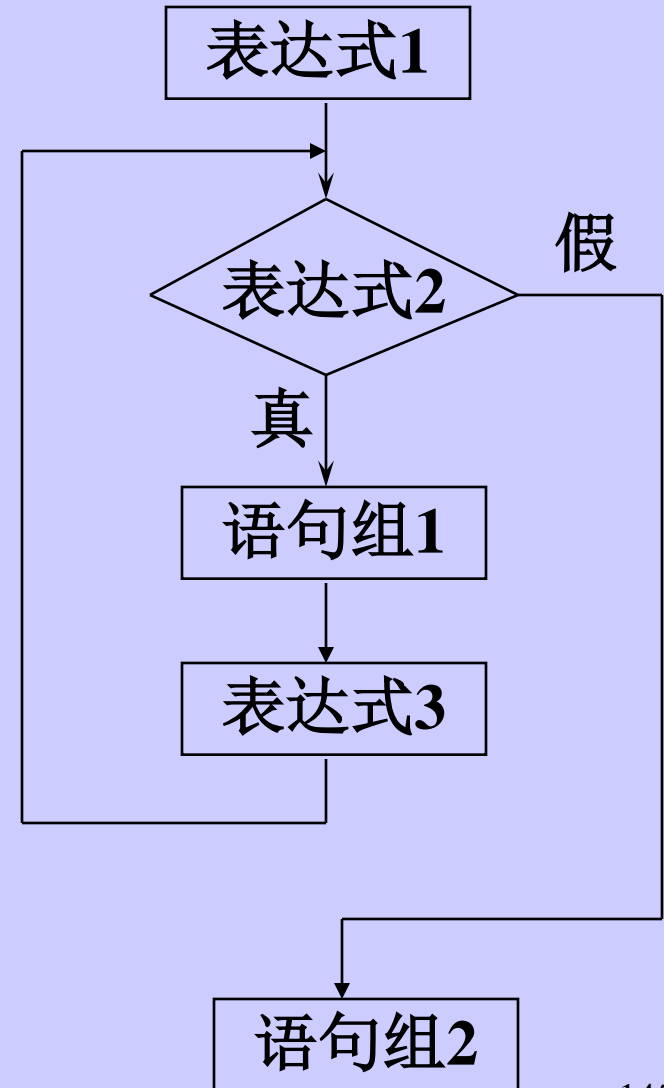
{ 语句组2 }



for（表达式1； 表达式2； 表达式3）

{ 语句组1 }

{ 语句组2 }



最大公约数与最小公倍数

求两自然数 m, n 的最大公约数

欧几里德算法 ($m > n$)

1、 m 被 n 除得到余数 r ($0 \leq r \leq n$) $r = m \% n$

2、若 $r=0$ ，则算法结束， n 为最大公约数，否则做3

3、 $m \leftarrow n, n \leftarrow r$ ，回到1

$m=6$	$n=4$	$r=m \% n=6 \% 4=2$	while ($r=m \% n$)
$m=4$	$n=2$	$r=m \% n=4 \% 2=0$	{ $m=n; \quad n=r;$
所以，公约数=2			}

最小公倍数为两数之积除以最大公约数。 $4 * 6 / 2 = 12$

最大公约数：能同时被m和n整除的最大数。

```
r=m>n?n:m
```

```
for(i=1; i<r; i++)
```

```
if(m%i==0&& n%i==0)
```

```
    a=i;
```

```
cout<<a;
```

将 12345 的每位分别打印出来。

$$12345 \% 10 = 5$$

$$12345 / 10 = 1234$$

$$1234 \% 10 = 4$$

$$1234 / 10 = 123$$

$$123 \% 10 = 3$$

$$123 / 10 = 12$$

$$12 \% 10 = 2$$

$$12 / 10 = 1$$

$$1 \% 10 = 1$$

$$1 / 10 = 0$$

while(n)

{ cout<<n%10<<'\t';

n=n/10;

}

求级数公式:

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots + (-1)^n \frac{x^{2n}}{(2n)!}$$

首先写出通项, 然后再用**当前项** (第**n**项) 除前一项, 得出后一项比前一项大多少倍。

通项: $(-1)^n \frac{x^{2n}}{(2n)!}$

第n项/第n-1项:

$$t = x * x / ((2 * n) * (2 * n - 1))$$

表明前一项比后一项大**t**倍, 即后一项乘**t**等于前一项

$$\text{后一项} = (-1) \times \text{前一项} \times t$$

后一项= $(-1) \times$ 前一项 $\times t$

设通项为term, 则可以写出迭代公式

term $=(-1)*$ **term** $*t;$

当前项

前一项

$t=x*x/((2*n)*(2*n-1))$

$S=0; term=1; n=1;$ //一定要赋初值

$while(fabs(term) \geq 1e-5)$

{ $S=S+term;$

新的 $term=(-1)*$ **旧的** $term*x*x/((2*n)*(2*n-1));$

$n++;$

旧的

}

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots + (-1)^n \frac{x^{2n}}{(2n)!}$$

第n项/第n-1项: $t = x * x / ((2 * n) * (2 * n - 1))$

第一项: $\text{term} = 1;$

第一次循环: $S = S + \text{term}; \text{term} = (-1) * \text{term} * t;$

这时左边的 term 代表第二项, 而右边的 term 为第一项。

第二次循环: $S = S + \text{term}; \text{term} = (-1) * \text{term} * t;$

这时左边的 term 代表第三项, 而右边的 term 为第二项。

当前项 $\text{term} = (-1) * \text{term} * t;$ 前一项

同样是 term , 在循环中不断用旧的数值去推导赋值出新的数值。

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots + (-1)^n \frac{x^{2n}}{(2n)!}$$

S=0;term=1;n=1; //一定要赋初值

while(fabs(term)>=1e-5)

{ S=S+term;

term=(-1)*term*x*x/((2*n)*(2*n-1));

新的

n++;

旧的

}

break语句和continue语句

break在switch语句中，可以使流程跳过判断体，执行下面的程序。在循环体中，也可以从循环体内跳出循环体，提前结束循环。

```
for ( ; ; )
```

```
{ cin>>x;
```

```
    if (x==123) break;
```

```
}
```

当输入123时，结束循环。

break 只能退出一层循环或switch语句。

```
a=10 ; y=0;
```

```
do
```

```
{ a+=2; y+=a;
```

```
cout<<"a="<<a<<" , y="<< y<<endl;
```

```
if (y>50) break;
```

```
} while (a=14);
```

第一次: a=12 y=12

输出: a=12 , y=12

第二次: a=16 y=28

输出: a=16 , y=28

第三次: a=16 y=44

输出: a=16 , y=44

第四次: a=16 y=60

输出: a=16 , y=60

continue: 其作用为结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。

```
void main(void)
```

```
{ int i;
```

```
for (i=1 ; i<=5 ; i++ )
```

```
{ if (i%2) cout<<"*";
```

```
else continue;
```

```
cout<<"#";
```

```
}
```

```
cout<<" $\\n";
```

```
}
```

输出： *##*##\$

i	1	2	3	4	5	6
i<=5	真	真	真	真	真	假
i%2	1	0	1	0	1	
输出	*#	无	*#	无	*#	\$

```
void main(void)
```

```
{  int  i, j, x=0 ;
```

```
    for (i=0 ; i<2; i++)
```

```
    {  x++;
```

```
        for (j=0;j<=3; j++)
```

```
        {  if ( j%2 )  continue;
```

```
            x++;
```

```
        }
```

```
        x++;
```

```
    }
```

```
    cout<<"x="<< x<<endl;
```

```
}
```

输出: x=8

i=0 i<2 第一次

j	0	1	2	3	4
j%2	假	真	假	真	
x	2	2	3	3	4

i=1 i<2 第二次

j	0	1	2	3	4
j%2	假	真	假	真	
x	6	6	7	7	8

i=2 i<2 结束

```

void main(void )
{
    int k=0;
    char c='A';
    do
    {
        switch (c++)
        {
            case 'A': k++; break;
            case 'B': k--;
            case 'C': k+=2; break;
            case 'D': k=k%2; continue;
            case 'E': k=k*10; break;
            default: k=k/3;
        }
        k++;
    }while (c<'G');
    cout<<"k="<< k<<endl;
}

```

c++	A	B	C	D	E	F
c	B	C	D	E	F	G
k	2	4	7	1	11	4
	真	真	真	真	真	假

输出： k=4

总结:

在循环体中，**break**从循环体内跳出循环体，提前结束循环。

```
for(... ; ... ; ... )
```

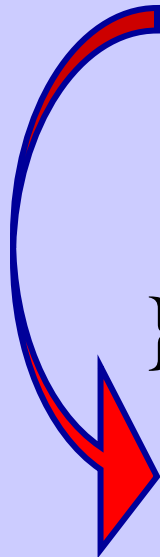
```
{
```

```
.....
```

```
break;
```

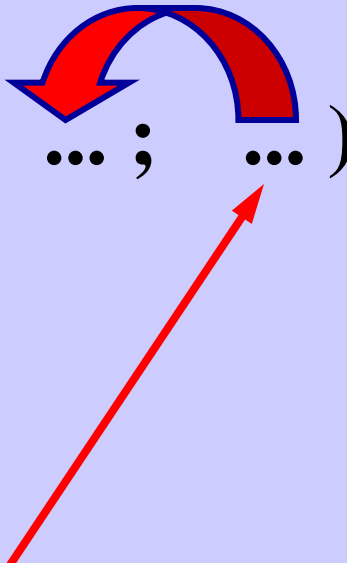
```
.....
```

```
}
```

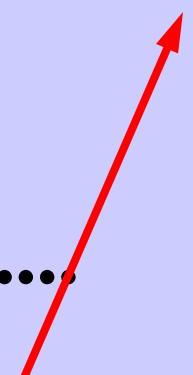


continue: 其作用为结束本次循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。

```
for(... ; ... ; ... )  
{  
    .....  
    continue;  
    .....  
}
```



```
while( ..... )  
{  
    .....  
    continue;  
    .....  
}
```



求素数：只可以被1与自身整除的数。

判断一个数 t 是否为素数，用2到 $t-1$ 循环除。

```
for( i=2; i<t/2 ; i++)
```

```
    if(t%i==0)
```

```
        break;
```

```
    if (i>=t/2)    cout<<“是素数。 \n”;
```

```
else cout<<“不是素数\n”;
```

进一步，由于 t 不可能被大于 $t/2$ 的数整除，所以可将循环次数降低。

求范围内的素数(50~100):

```
for(t=50, k=0 ; t<=100; t++)  
{   for( i=2; i<t    ; i++)  
    if(t%i==0)  
        break;  
    if (i==t)  
    { cout<<t<<" ";  
      k++;  
      if(k%5==0) cout<<endl;  
    }  
}
```

判断t是否
为素数

保证每行输
出5个数据

鸡兔共有30只，脚共有90只，问鸡兔各有多少？

```
void main(void)
```

```
{ int i; //i代表鸡，则兔为30-i只
```

```
for(i=0; i<=15; i++)
```

```
if(2*i + 4*(30-i) == 90)
```

```
{ cout<<“鸡” <<i<<endl;
```

```
cout<<“兔” <<30-i<<endl;
```

```
}
```

```
}
```

一百万富翁 遇到一陌生人，陌生人找他谈一个换钱的计划，该计划如下：我每天给你十万元，而你第一天只需给我一分钱，第二天我仍给你十万元，你给我两分钱，第三天我仍给你十万元，你给我四分钱，....，你每天给我的钱是前一天的两倍，直到满一个月（30天），百万富翁很高兴，欣然接受了这个契约。请编写程序计算陌生人给百万富翁多少钱，百万富翁给陌生人多少钱？

利用循环语句编程，打印下列图形：

```

      *
    * *
  * * *
* * * *
* * *
* *
*
```

行号	空格	星号
1	3	1
2	2	2
3	1	3
4	0	4

行号: i
空格: $4-i$
星号: i

找规律:

上面四行

```
for(i=0;i<4;i++)
{
    for(j=4-i-1;j>0;j--)
        cout<<" ";
    for(k=1;k<=i+1;k++)
        cout<<" * ";
    cout<<endl<<endl;
}
```

```

void main(void)
{
    int i,j,k;
    for(i=0;i<4;i++)
    {
        for(j=4-i-1;j>0;j--)
            cout<<"  ";
        for(k=1;k<=i+1;k++)
            cout<<" * ";
        cout<<endl<<endl;
    }
    for(i=0;i<4-1;i++)
    {
        for(j=4-1-i;j>0;j--)
            cout<<" * ";
        cout<<endl<<endl;
    }
}

```

打印图形：

```

      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

行号	空格	星号
0	3	1
1	2	3
2	1	5
3	0	7

行号： i

空格： $3-i$

星号： $2*i+1$

如果打印 n 行 行号： $0 \sim n-1$ 空格： $0 \sim n-1-i$

计算： $2+22+222+....+2222222=?$

累加和 $s=0$ 设通项为 t

所以，通项的循环表示为：

$$22=2*10+2;$$

$$222=22*10+2;$$

$$2222=222*10+2;$$

前一项

$$t=t*10+2;$$

当前项

t 的初值为2

循环体为：

```
{ s = s + t  
  t = t * 10 + 2;  
}
```

```
void main(void)  
{    int t,s;  
      s=0;  
      t=2;  
      for(int i=0;i<7;i++)  
      {  
          s=s+t;  
          t=t*10+2;  
      }  
      cout<<s<<endl;  
}
```


满足以下条件三位数 n ，它除以11所得到的商等于 n 的各位数字的平方和，且其中至少有两位数字相同。**131** $131/11=11$ $1^2+3^2+1^2=11$

分析：

数字应该从100循环到999

将每位数字剥出，判断其是否满足条件

满足以下条件三位数 n ，它除以11所得到的商等于 n 的各位数字的平方和，且其中至少有两位数字相同。**131** $131/11=11$ $1^2+3^2+1^2=11$

分析：

用 a, b, c 分别代表三位数， a, b, c 分别从0循环到9，组成所有可能的三位数，然后找出满足条件的数来。

$$\frac{\pi}{2} = \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \cdots \times \frac{2n}{2n-1} \times \frac{2n}{2n+1} \cdots$$

求n=1000时 π 的近似值

分析: 通项: $t = \frac{2n}{2n-1} \times \frac{2n}{2n+1}$

迭代算法:

$$s = s \times t$$

本次计算
的结果

上次迭代
的结果

迭代结束条件:

迭代 1000次

注意s与t的初始值

n从1开始迭代 s=1

下面程序的功能是用公式求 π 的近似值，直到最后一项的值小于 10^{-6} 为止。请填空。

```
void main(void )  
{  int  i=1;  
    double    pi=0;  
    while (i*i<=10e+6)  
    {  pi=pi+1.0/(i*i)  ;  
        i++;  
    }  
    pi=sqrt (6.0 *pi) ;  
    cout<<“ pi=”<<pi<<endl;  
}
```

有1020个西瓜，第一天卖一半多两个，以后每天卖剩下的一半多两个，问几天以后能卖完？请填空。

```
#include"stdio.h"
```

```
void main(void )
```

```
{ int day, x1, x2;
```

```
    day=0; x1=1020;
```

```
    while (  x1  )
```

```
    { x2=  x1/2-2 ;
```

```
        x1=x2;    day++;
```

```
    }
```

```
    cout<<"day="<<day<<endl;
```

```
}
```

第五章 函数与编译预处理

概述

函数是程序代码的一个自包含单元，用于完成某一特定的任务。

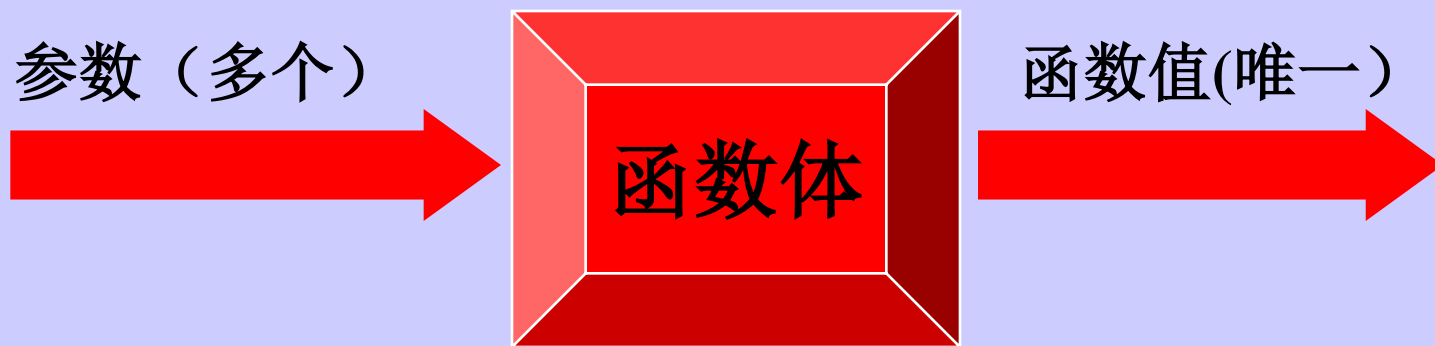
C++是由函数构成的，函数是C++的基本模块。

有的函数完成某一操作；有的函数计算出一个值。通常，一个函数即能完成某一特定操作，又能计算数值。

为什么要使用函数？

- 1、避免重复的编程。
- 2、使程序更加模块化，便于阅读、修改。

所编写的函数应尽量少与主调函数发生联系，这样便于移植。



说明：

- 1、一个源程序文件由一个或多个函数组成，编译程序以文件而不是以函数为单位进行编译的。
- 2、一个程序可以由多个源文件组成，可以分别编译，统一执行。
- 3、一个程序必须有且只有一个`main()`函数，C++从`main()`函数开始执行。
- 4、C++语言中，所有函数都是平行独立的，无主次、相互包含之分。函数可以嵌套调用，不可嵌套定义。
- 5、从使用角度来说，分标准函数和用户自定义函数；从形式来说，分无参函数和有参函数。

库函数是**C++编译系统已预定义的函数**，用户根据需要可以直接使用这类函数。库函数也称为标准函数。

为了方便用户进行程序设计，C++把一些常用数学计算函数（如`sqrt()`、`exp()`等）、字符串处理函数、标准输入输出函数等，都作为库函数提供给用户，用户可以直接使用系统提供的库函数。

库函数有很多个，当用户使用任一库函数时，在程序中必须包含相应的头文件。如**`#include<iostream.h>`**等。

用户在设计程序时，可以将完成某一相对独立功能的程序定义为一个函数。用户在程序中，根据应用的需要，由**用户自己定义函数**，这类函数称为用户自定义的函数。

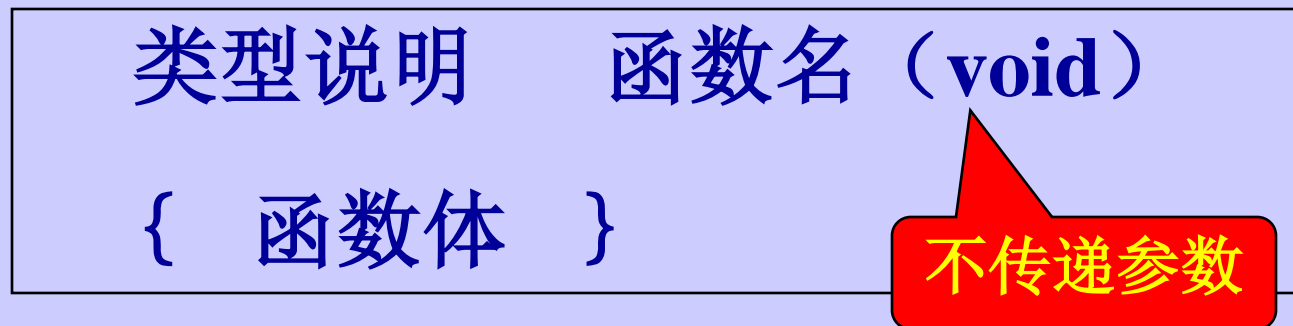
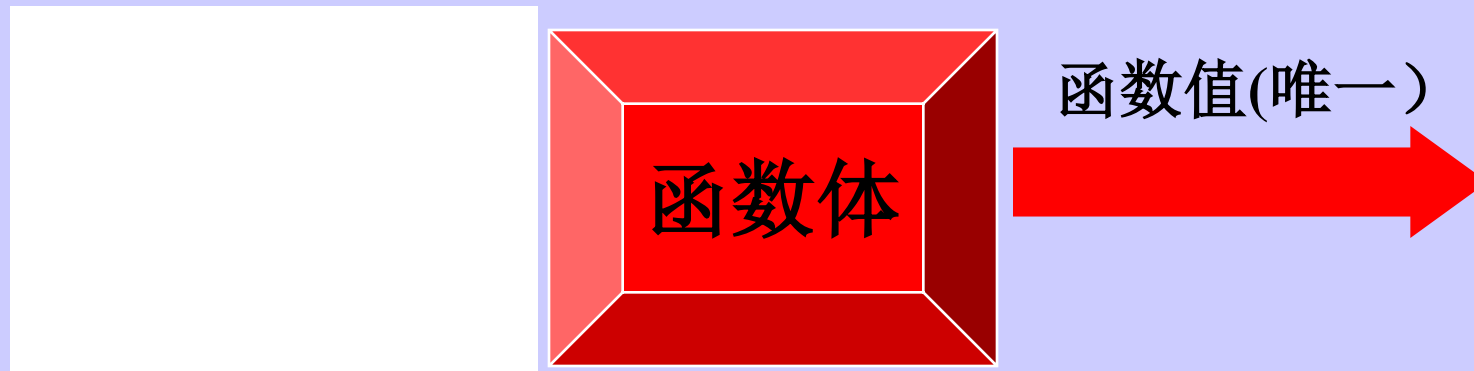
根据定义函数或调用时是否要给出参数，又可将函数分为：无参函数和有参函数。

函数定义的一般形式

一、无参函数

主调函数并不将数据传给被调函数。

无参函数主要用于完成某一操作。



```
void main(void )
```

```
{ printstar ( );
```

调用函数

```
    print_message ( );
```

调用函数

```
    printstar( );
```

调用函数

```
} 函数类型
```

函数名

```
void    printstar (void )
```

函数体

```
{    cout<<“* * * * * * * * * * \n”;
```

```
}
```

```
void    print_message (void)
```

```
{    cout<<“ How do you do! \n”;
```

```
}
```

两个被调函数
主要用于完成
打印操作。

输出: * * * * * * * * * *

How do you do!

* * * * * * * * * *

二、有参函数

主调函数和被调函数之间有数据传递。主调函数可以将参数传递给被调函数，被调函数中的结果也可以带回主调函数。

类型说明	函数名（形式参数列表说明）
{ 函数体 }	

形参列表说明

函数类型

`int max (int x,int y)`

`{ int z;`

函数名

函数体

`z=(x>y)? x : y ;`

`return z;`

函数值

`void main {`
`void)`

主调函数

`{ int a,b,c;`

`cin>>a>>b;`

调用函数

`c=max (a , b) ;`

实际参数

`cout<<"The max is"<< c<<endl;`

将实际值**a**,**b**传给
被调函数的参数
x,y，计算后得到
函数值**z**返回

`}`


```
int max (int x,int y)
```

```
{ int z;
```

```
  z=(x>y)? x : y ;
```

```
  return z;
```

```
}
```

```
void main (void )
```

```
{ int a,b,c;
```

```
  cin>>a>>b;
```

```
  c=max (a , b) ;
```

```
  cout<<"The max is"<< c<<endl;
```

```
}
```



a



b



c

函数参数和函数的值

形参是被调函数中的变量；实参是主调函数赋给被调函数的特定值。实参可以是常量、变量或复杂的表达式，不管是哪种情况，在调用时实参必须是一个确定的值。

形参与实参类型相同，一一对应。

形参必须要定义类型，因为在定义被调函数时，不知道具体要操作什么数，而定义的是要操作什么类型的数。

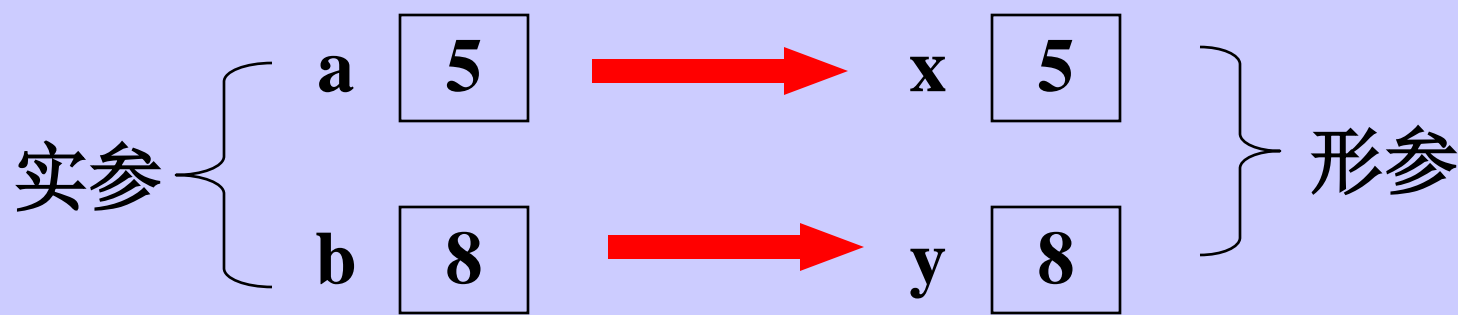
```
int max (int x,int y)
{ int z;
  z=(x>y)? x : y ;
  return z;
}
void main (void )
{ int a,b,c;
  cin>>a>>b;
  c=max (a+b , a*b) ;
  cout<<"The max is"<<c<<endl;
}
```

若a为3， b为5， 则实参为8, 15， 分别送给形参x, y。

先计算， 后赋值

说明：

- 1、在未出现函数调用时，形参并不占内存的存储单元，只有在函数开始调用时，形参才被分配内存单元。调用结束后，形参所占用的内存单元被释放。
- 2、实参对形参变量的传递是“**值传递**”，即单向传递。在**内存中实参、形参分占不同的单元**。
- 3、形参只作用于被调函数，可以在别的函数中使用相同的变量名。



```
void fun(int a, int b)
```

```
{  a=a*10;
```

```
    b=b+a;
```

```
    cout<<a<<'\t'<<b<<endl;
```

```
}
```

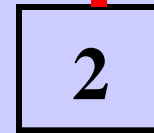
```
void main(void)
```

```
{  int a=2, b=3;
```

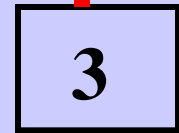
```
    fun(a,b);
```

```
    cout<<a<<'\t'<<b<<endl;
```

```
}
```



a



b

20

2

23

3

```
void fun(int x, int y)
```

```
{  x=x*10;
```

```
    y=y+x;
```

```
    cout<<x<<'\t'<<y<<endl;
```

```
}
```

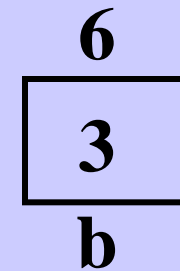
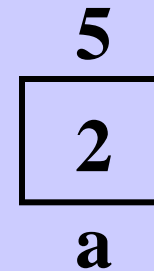
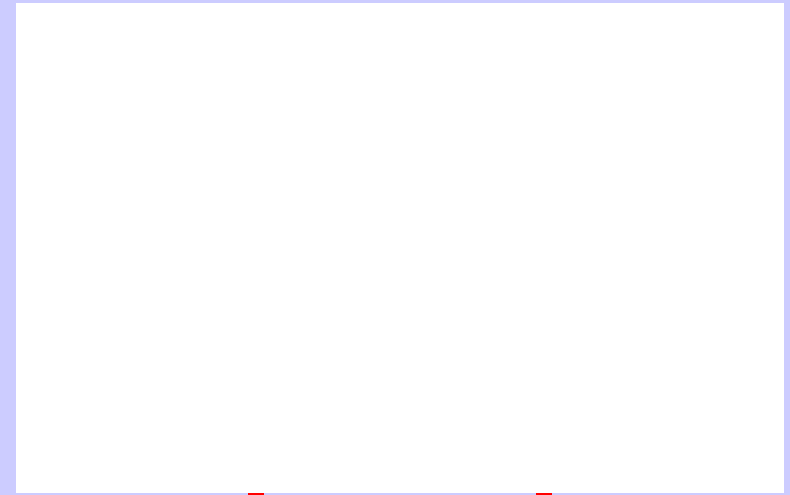
```
void main(void)
```

```
{  int a=2, b=3;
```

```
    fun(a+b,a*b);
```

```
    cout<<a<<'\t'<<b<<endl;
```

```
}
```



50	56
2	3

形参实参类型相等，一一对应

形参必须要定义类型，因为在定义被调函数时，不知道具体要操作什么数，而定义的是要操作什么类型的数。

形参是被调函数中的变量；实参是主调函数赋给被调函数的特定值。在函数调用语句中，实参不必定义数据类型，因为实参传递的是一个具体的值(常量)，程序可依据这个数值的表面形式来判断其类型，并将其赋值到对应的形参变量中。

函数的返回值

函数的返回值通过**return**语句获得。函数只能有唯一的返回值。

函数返回值的类型就是函数的类型。

return语句可以是一个表达式，函数先计算表达式后再返回值。

return语句还可以终止函数，并将控制返回到主调函数。

一个函数中可以有一个以上的**return**语句，执行到哪一个**return**语句，哪一个语句起作用。


```
int add ( int a, int b)
```

```
{
```

先计算，后返回

```
    return (a+b);
```

```
}
```

```
int max ( int a, int b)
```

```
{
```

```
    if (x>y)
```

```
        return x ;
```

```
    else return y;
```

```
}
```

若函数体内没有return语句，就一直执行到函数体的末尾，然后返回到主调函数的调用处。

可以有多个return语句

既然函数有返回值，这个值当然应属于某一个确定的类型，应当在定义函数时指定函数值的类型。

int max (float a, float b) // 函数值为整型

函数返回值的类型，也是函数的类型

不带返回值的函数可说明为void型。

函数的类型与函数参数的类型没有关系。

double blink (int a, int b)

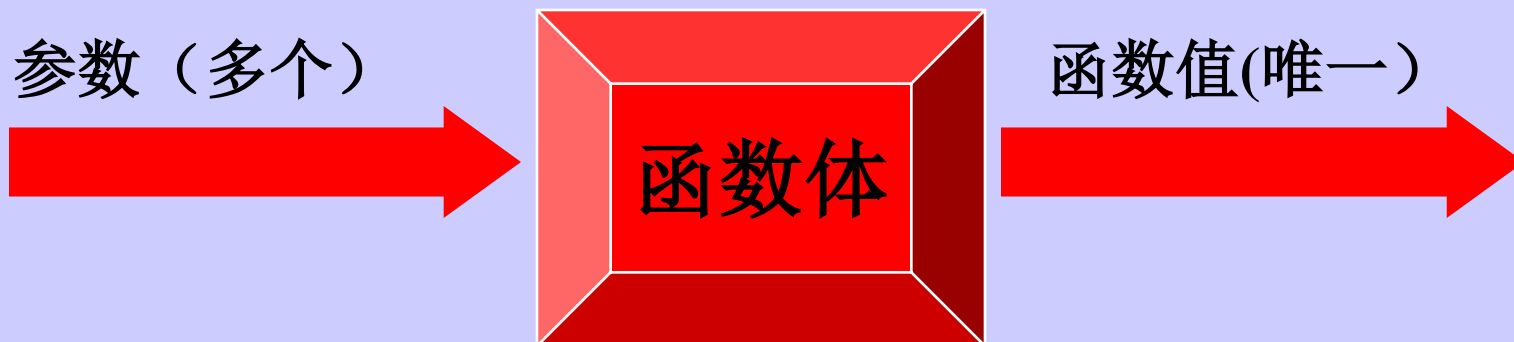
如果函数的类型和return表达式中的类型不一致，则以函数的类型为准。函数的类型决定返回值的类型。对数值型数据，可以自动进行类型转换。

如果有函数返回值, 函数的类型就是返回值的类型

```
int max ( int a, int b)
{   int z;
    z=x>y?x:y;
    return z;
}
```

如果有函数返回值, 返回值就是函数值, 必须惟一。

函数体的类型、形式参数的类型必须在函数的定义中体现出来。



函数的调用

函数调用的一般形式

函数名（实参列表）；

```
i=2; f (i, ++i);
```

实际调用： **f (3, 3);**

形参与实参类型相同，一一对应。

函数调用的方式

作为语句

```
printstar( );
```

作为表达式

```
c=max (a,b);
```

作为另一个函数的参数

```
cout<<max (a,b);
```

在一个函数中调用另一函数（即被调用函数）需要具备哪些条件呢？

- 1) 被调用的函数必须是已存在的函数
- 2) 如果使用库函数，必须用 `#include <math.h>`
- 3) 函数调用遵循先定义、后调用的原则，即被调函数应出现在主调函数之前。

```
float max(float x, float y)
```

```
{ float z;
```

```
  z=(x>y)? x : y ;
```

```
  return z;
```

```
}
```

形参必须说明参数类型

被调函数先定义

```
void main (void)
```

```
{ float  a,b, c;
```

```
  cin>>a>>b;
```

```
  c=max (a+b , a*b) ;
```

```
  cout<<“The max is”<<c<<endl;
```

```
}
```

定义之后再调用

实参传递的是一个具体的值，不必说明参数类型

4) 如果使用用户自己定义的函数，而该函数与调用它的函数（即主调函数）在同一个程序单位中且位置在主调函数之后，则必须在调用此函数之前对被调用的函数作声明。

```
void main (void)
```

```
{ float a,b, c;
```

```
float max (float,float);
```

函数原型说明

```
cin>>a>>b;
```

```
c=max (a,b) ;
```

```
cout<<“The max is”<<c<<endl;
```

```
}
```

```
float max (float x, float y)
```

函数定义

```
{ float z;
```

```
z=(x>y)? x : y ;
```

```
return z;
```

```
}
```

定义是一个完整的函数单位，
而原型说明仅仅是说明函数的
返回值及形参的类型。


```
void main(void)
```

```
{ int i=2, x=5, j=7; void fun(int,int);
```

```
    fun ( j, 6);
```

```
    cout<<i<<'\t'<< j<<'\t'<< x<<endl;
```

```
}
```

```
void fun ( int i, int j)
```

```
{ int x=7;
```

```
    cout<<i<<'\t'<< j<<'\t'<<x<<endl;
```

输出: 7 6 7

```
}
```

i

x

j

2

5

7

6



2 7 5

```
void main(void )
```

```
{  int  x=2,y=3, z=0; void add(int,int,int);
```

```
  cout<<"(1) x="<<x<<" y="<<y<<" z="<<z<<endl;
```

```
  add (x, y, z);
```

```
  cout<< "(3) x="<<x<<" y="<<y<<" z="<<z<<endl;
```

```
}
```

```
void add ( int x, int y, int z)
```

```
{  z=x+y; x=x*x; y=y*y;
```

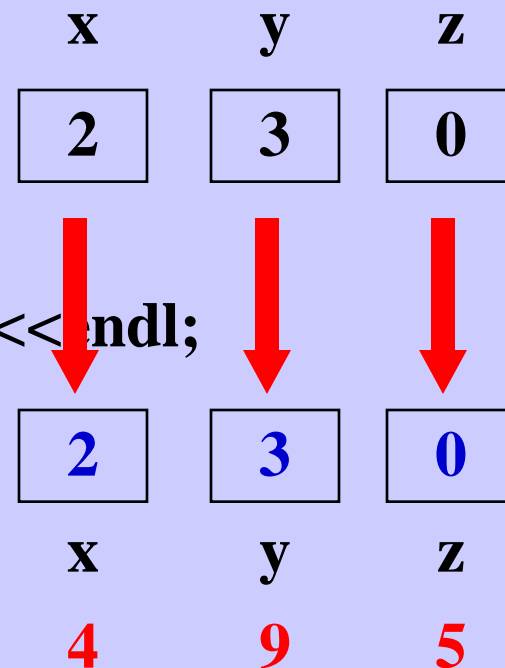
```
  cout<<"(2) x="<<x<<" y="<<y<<" z="<<z<<endl;
```

```
}
```

(1) x=2 y=3 z=0

(2) x=4 y=9 z=5

(3) x=2 y=3 z=0



编写程序，分别计算下列函数的值(**x**从键盘输入)

$$f_1(x) = 3x^3 + 2x^2 - 1$$

$$f_2(x) = x^2 + 2x + 1$$

```
float f1(float x)  
{ float y;  
    y=3*x*x*x+2*x*x-1;  
    return y;  
}  
  
void main(void)  
{  
    float x, y;  
    cin>>x;  
    y=f1(x);  
    cout<<"x="<<x<<" , y="<<y<<endl;  
}
```

编写程序，分别计算下列函数的值(x从键盘输入)

$$s = 1 + \frac{1}{x} + \frac{1}{x^2} + \frac{1}{x^3} + \cdots (x > 1)$$

当最后一项小于**0.00001**时，累加结束。

```
float fun(float x)
```

```
{ float s=1, t=1;
```

```
do
```

```
{ t=t/x;
```

```
  s+=t;
```

```
  }while (t>0.00001); }
```

```
return s;
```

```
}
```

```
void main(void)
```

```
{ float x;
```

```
  cin>>x;
```

```
  cout<<"s="<<fun(x)<<endl;
```

```
}
```

计算100~200之间的素数，用函数prime()判断一个数是否是素数，若是该函数返回1，否则返回0。

```
int prime(int x)
{
    for(int i=2;i<x/2;i++)
        if(x%i==0)
            return 0;

    return 1;
}

void main(void)
{
    for(int i=100;i<=200; i++)
        if(prime(i)==1)
            cout<<i<<'\t';

}
```

计算输入两个数的最大公约数

```
int gys(int a, int b)
```

```
{ int r;
```

```
    if(a<b){r=a; a=b; b=r;}
```

```
    while(r=a%b)
```

```
    { a=b; b=r;}
```

```
    return b;
```

```
}
```

```
void main(void)
```

```
{ int x, y;
```

```
    cin>>x>>y;
```

```
    cout<<gys(x,y)<<endl;
```

```
}
```


计算输入三个数的最大公约数

```
int gys(int a, int b, int c)
```

```
{ int r;
```

```
  if(a<b){r=a; a=b; b=r;}
```

```
  r=r>c?r:c;
```

```
  for(int i=r-1; i>=1; i--)
```

```
  { if(a%i==0 && b%i==0 && c%i==0)
```

```
      break;
```

```
  }
```

```
  return i;
```

```
}
```

```
void main(void)
```

```
{ int x, y, z;
```

```
  cin>>x>>y>>z;
```

```
  cout<<gys(x,y,z)<<endl;
```

```
}
```

写一个函数验证哥德巴赫猜想：一个不小于6的偶数可以表示为两个素数之和，如 $6=3+3$ ， $8=3+5$ ， $10=3+7$。在主函数中输入一个不小于6的偶数n，函数中输出以下形式的结果：

$$34=3+31$$

函数的嵌套调用

C语言中，所有函数都是平行独立的，无主次、相互包含之分。函数可以嵌套调用，不可嵌套定义。

```
int max ( int a, int b)
{
    int c;
    int min ( int a, int b)
    {
        return ( a<b? a: b);
    }
    c=min(a,b);
    return ( a>b? a : b);
}
```

嵌套定义

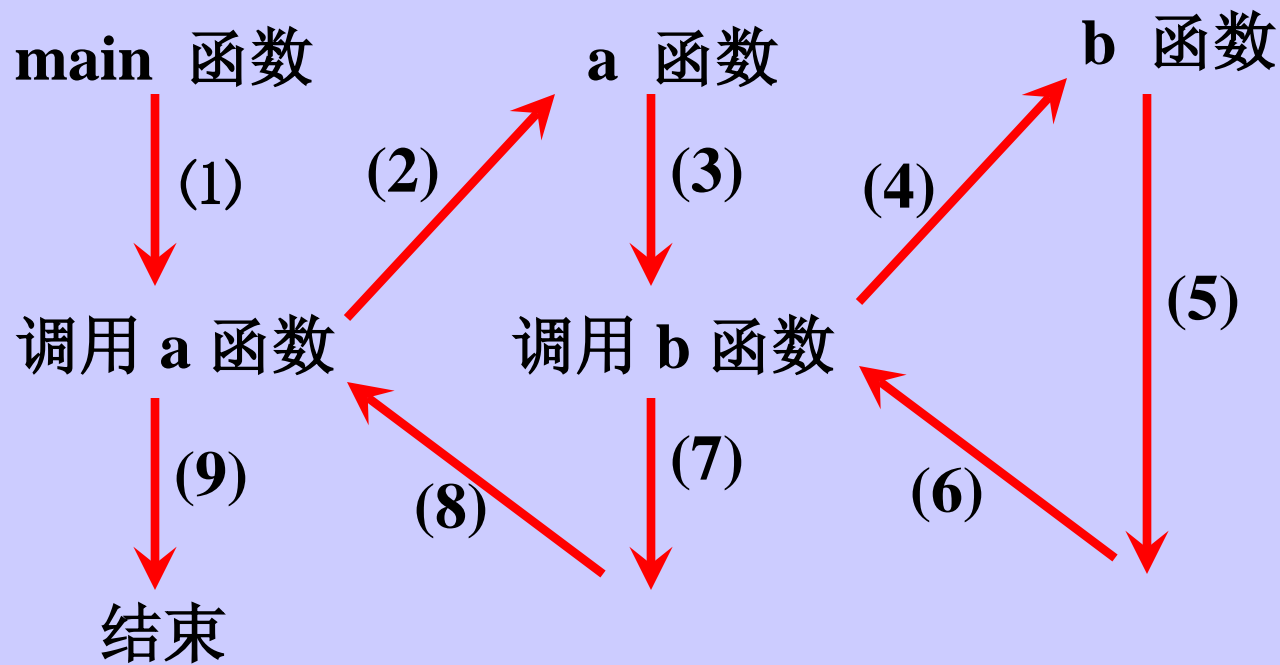
```
int min ( int a, int b)
{
    return ( a<b? a: b);
}

int max ( int a, int b)
{
    int c;
    c=min(a,b);
    return ( a>b? a : b);
}
```

平行定义

嵌套调用

在main函数中调用a函数，在a函数中又调用b函数。



$$f(k, n) = 1^k + 2^k + 3^k + \dots + n^k$$

```
int power(int m,int n) //m^n
```

```
{    int i,product=m;  
    for(i=1;i<n;i++)
```

平行定义

```
        product=product*m;
```

```
    return product;
```

平行定义

```
}
```

```
int sum_of_power(int k,int n) //n^k的累加和
```

```
{    int i,sum=0;
```

```
    for(i=1;i<=n;i++)
```

嵌套调用

```
        sum+=power(i,k);
```

```
    return sum;
```

```
}
```

```
void main(void)
```

```
{    int k,m;
```

```
    cin>>k>>m;
```

嵌套调用

```
    cout<<"f("<<k<<","<<m<<")="<<sum_of_power(k,m)<<endl;
```

```
    //m^k的累加和
```

```
}
```

函数的递归调用

在调用一个函数的过程中直接或间接地调用函数本身，称为函数的递归调用。

```
int f(int x)
```

```
{ int y,z ;
```

```
....
```

```
z=f(y);
```

```
....
```

```
return (2*z);
```

```
}
```

```
int f1(int x)
```

```
{ int y,z ;
```

```
....
```

```
z=f2(y);
```

```
....
```

```
return (2*z);
```

```
}
```

```
int f2(int t)
```

```
{ int a, c ;
```

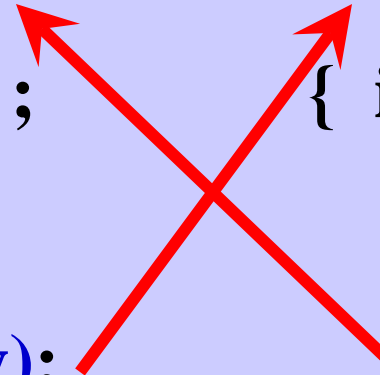
```
....
```

```
c=f1(a);
```

```
....
```

```
return (3+c);
```

```
}
```



有5个人坐在一起，问第5个人多少岁，他说比第4个人大2岁。问第4个人多少岁，他说比第3个人大2岁。问第3个人多少岁，他说比第2个人大2岁。问第2个人多少岁，他说比第1个人大2岁。问第1个人多少岁， he 说是10岁。请问第5个人多大？

$\text{age}(5) = \text{age}(4) + 2$

$\text{age}(4) = \text{age}(3) + 2$

$\text{age}(3) = \text{age}(2) + 2$

$\text{age}(2) = \text{age}(1) + 2$

$\text{age}(1) = 10$

`void main(void)`

`{ int age(int);`

`cout << age(5) << endl;`

`}`

$$\text{age}(n) = \begin{cases} 10 & n=1 \\ \text{age}(n-1) + 2 & n > 1 \end{cases}$$

必须有递归结束条件

`int age (int n)`

`{ int c;`

`c = age(n-1) + 2;`

`return c;`

`}`

`int age (int n)`

`{ int c;`

`if (n == 1) c = 10;`

`else`

`c = age(n-1) + 2;`

`return c;`

`}`

```
void main(void)
```

```
{ int age(int);
```

```
  cout<<age(5)<<endl;
```

```
}
```

```
int age ( int n )
```

```
{ int c;
```

```
  if (n==1) c=10;
```

```
  else c=age(n-1)+2;
```

```
  return c;
```

```
}
```

age (5)

n=5

c=age (4)+2

return c

c=18

age (4)

n=4

c=age (3)+2

return c

c=16

age (3)

n=3

c=age (2)+2

return c

c=14

age (2)

n=2

c=age (1)+2

return c

c=12

age (1)

n=1

c=10

return c

虽然算法一致，但n不同，c不同，在内存中每一层函数变量所在的内存单元均不相同。必须有递归终止条件。

用递归方法求n!

$$n! = \begin{cases} 1 & n=0,1 \\ n*(n-1)! & n>1 \end{cases}$$

```
float fac (int n)
```

```
{ float y;
```

```
  if ((n==0)|| (n==1) y=1;
```

```
  else y=n*fac(n-1);
```

```
  return y;
```

```
}
```

```
void main (void)
```

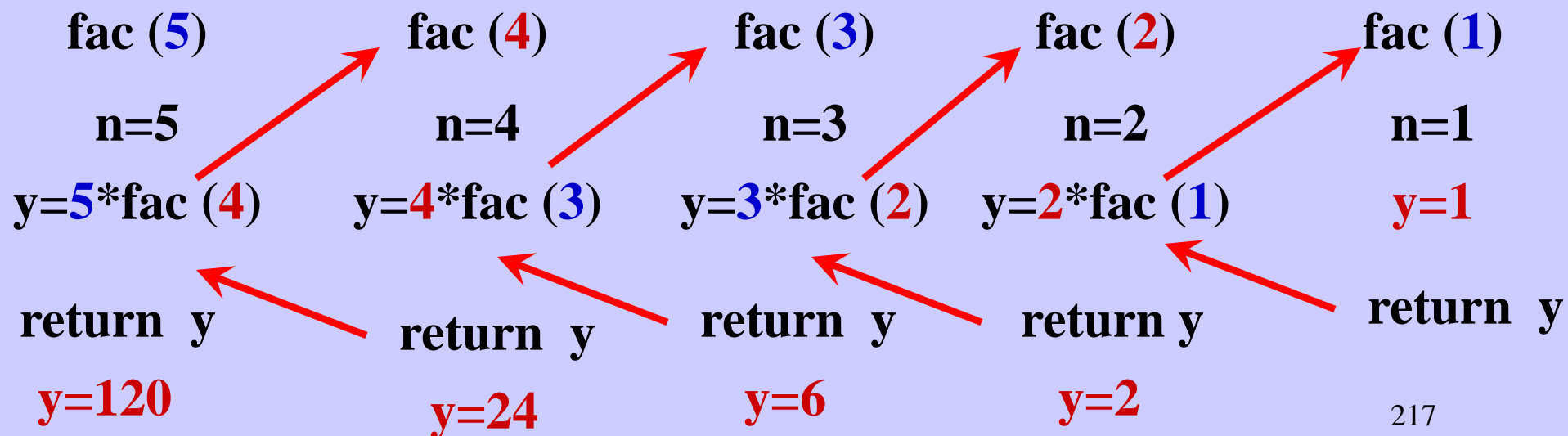
```
{ float y; int n;
```

```
  cout<<"Input n:\n";
```

```
  cin>>n ;
```

```
  cout<<n<<"!="<<fac(n)<<endl;
```

```
}
```



```

int sub(int);
void main (void)
{  int i=5;
   cout<<sub(i)<<endl;
}

```

```

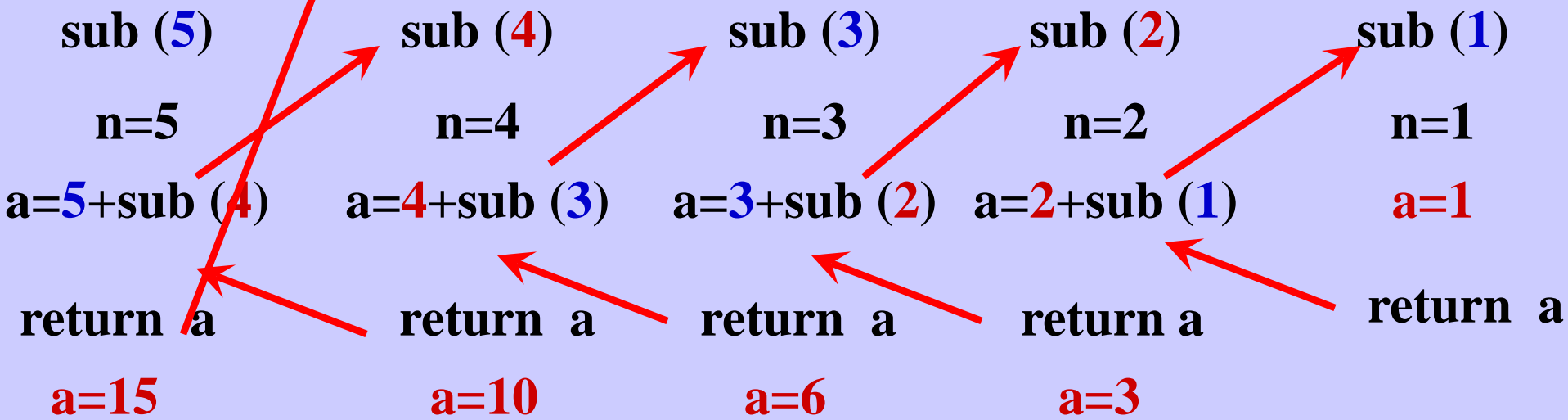
int sub (int n )

```

```

{  int a;
   if (n==1) return 1;
   a=n+sub(n-1);
   return a;
}

```



算法相同,层层调用,每层函数的变量所占内存单元不同。

```
void main (void)
```

```
{ int i=5;
```

```
cin>>i;
```

```
f(i);
```

```
}
```

43211234

输入: 1234

```
void f(int n )
```

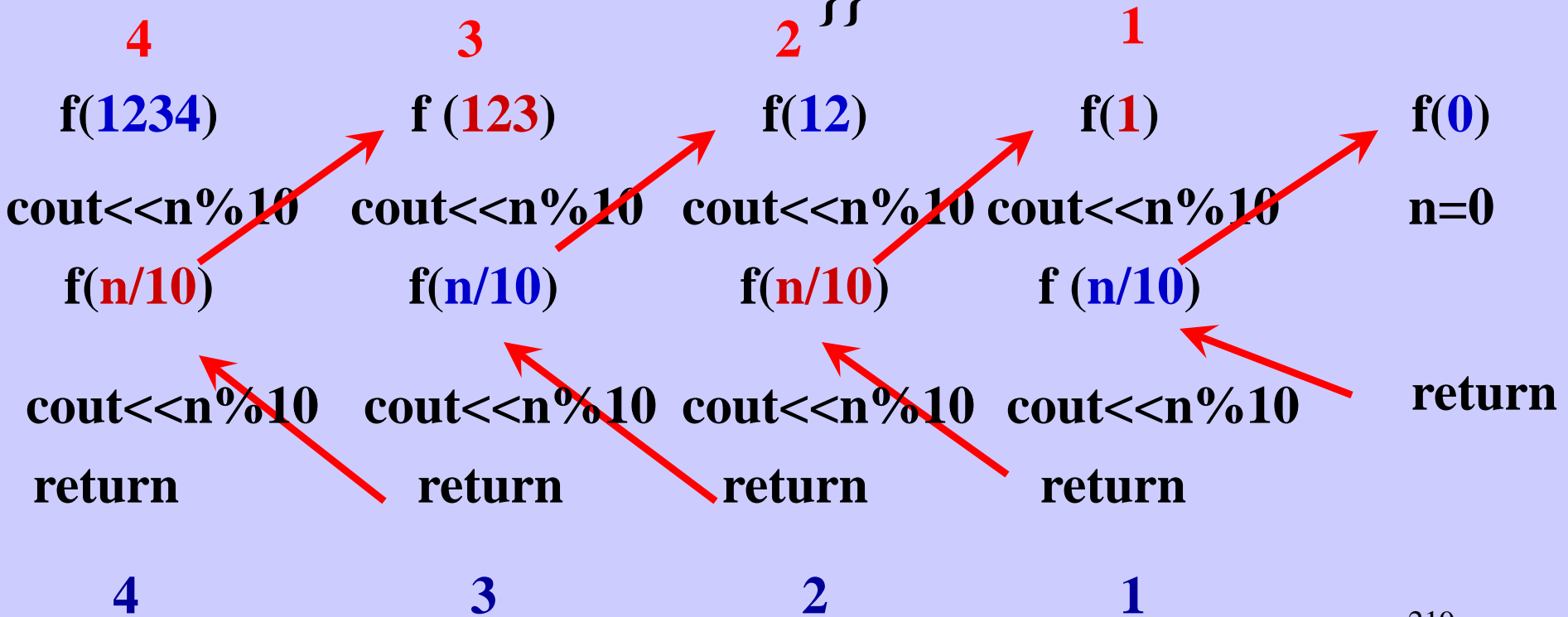
```
{if(n==0) return;
```

```
else {cout<<n%10;
```

```
f(n/10);
```

```
cout<<n%10; return;
```

```
}}
```



```
void recur(char c)  
{ cout<<c;  
    if(c<'5') recur(c+1);  
    cout<<c;  
}
```

```
void main(void)  
{ recur('0');  
}
```

输出： 012345543210

```
void f(int n)
```

```
{ if(n>=10)
```

1

```
    f(n/10);
```

12

```
    cout<<n<<endl;
```

123

```
}
```

1234

```
void main(void)
```

12345

```
{ f(12345);
```

```
}
```

作用域和存储类

作用域是指程序中所说明的标识符在哪一个区间内有效，即在哪一个区间内可以使用或引用该标识符。在C++中，作用域共分为五类：块作用域、文件作用域、函数原型作用域、函数作用域和类的作用域。

块作用域

我们把用花括号括起来的一部分程序称为一个块。在块内说明的标识符，只能在该块内引用，即其作用域在该块内，开始于标识符的说明处，结束于块的结尾处。

在一个函数内部定义的变量或在一个块中定义的变量称为局部变量。

在函数内或复合语句内部定义的变量，其作用域是从定义的位置起到函数体或复合语句的结束。**形参也是局部变量。**

```
float f1( int a)
{ int b,c;
  ....
}
```

} a,b,c有效

```
void main(void )
{ int m, n;
  ....
}
```

} m,n有效

```
float f2( int x, int y)
{ int i, j;
  ....
}
```

} x,y,i,j 有效

主函数main中定义的变量，也只在主函数中有效，同样属于局部变量。
不同的函数可以使用相同名字的局部变量，它们在内存中分属不同的存储区间，互不干扰。

```
void main(void)
{  int x=10;
    {  int x=20;
        cout<<x<<endl;
    }
    cout<<x<<endl;
}
```

定义变量既是在
内存中开辟区间

20

10

注意：

具有块作用域的标识符在其作用域内，将屏蔽其作用块包含本块的同名标识符，即变量名相同，局部更优先。

```
void main(void)
```

```
{  int a=2, b=3, c=5;
```

```
    cout<<a<<'\t'<<b<<'\t'<<c<<endl;
```

```
    {    int  a, b=2;
```

```
        a=b+c;
```

```
        cout<<a<<'\t'<<b<<'\t'<<c<<endl;
```

```
    }
```

```
    c=a-b;
```

```
    cout<<a<<'\t'<<b<<'\t'<<c<<endl;
```

```
}
```

2

a

3

b

-1

c

2

3

5

7

2

5

2

3

-1

```
void main(void)
```

```
{ int a=1,b=2,c=3;
```

```
    ++a;
```

a=2

```
    c+=++b;
```

b=3, c=6

```
    { int b=4, c;
```

b=4

```
        c=b*3;
```

c=12

```
        a+=c;
```

a=14

```
        cout<<"first:"<<a<<"\t"<<b<<"\t"<<c<<endl;
```

a=14,b=4,c=12

```
        a+=c;
```

a=26

```
        cout<<"second:"<<a<<"\t"<<b<<"\t"<<c<<endl;
```

a=26,b=4,c=12

```
    }
```

```
    cout<<"third:"<<a<<"\t"<<b<<"\t"<<c<<endl;
```

a=26,b=3,c=6

```
}
```

文件作用域

在函数外定义的变量称为全局变量。

全局变量的作用域称为文件作用域，即在整个文件中都是可以访问的。

其缺省的作用范围是：从定义全局变量的位置开始到该源程序文件结束。

当在块作用域内的变量与全局变量同名时，局部变量优先。

```
int p=1, q=5;
```

全局变量

```
float f1( int a)
```

```
{ int b,c;
```

```
.....
```

局部变量

```
}
```

```
char c1,c2;
```

```
main( )
```

```
{ int m, n;
```

```
.....
```

```
}
```

a,b,c有效

p,q有效

m,n有效

c1,c2有效

全局变量增加了函数间数据联系的渠道，在函数调用时可以得到多于一个的返回值。

```
int min;
```

全局变量

```
int max (int x, int y)
```

```
{ int z;
```

```
min=(x<y)?x : y;
```

```
z=(x>y)? x : y ;
```

```
return z;
```

函数值为4

```
}
```

```
void main (void)
```

```
{ int a,b,c;
```

```
cin>>a>>b;
```

```
c=max (a , b) ;
```

```
cout<<"The max is"<<c<<endl;
```

```
cout<<" The min is"<<min<<endl;
```

```
}
```

1

min

min 在main()和max()中均有效，
在内存中有唯一的存储空间。

1

a

4

b

4

c

The max is 4

The min is 1

在同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量不起作用。

```
int a=3, b=5;
```

```
int max(int a, int b)
```

```
{ int c;
```

```
    c=a>b? a:b;
```

```
    return c;
```

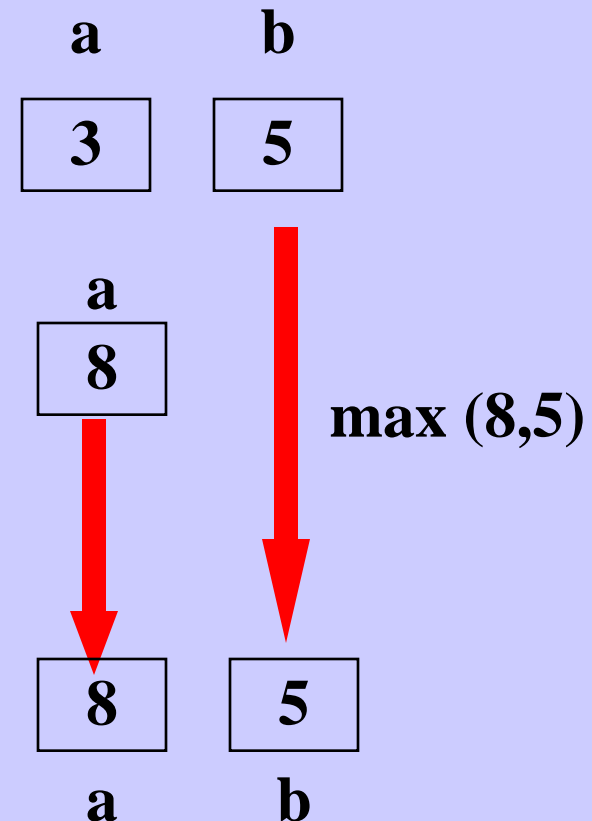
```
}
```

```
void main(void)
```

```
{ int a=8;
```

```
    cout<<max(a,b)<<endl;
```

```
}
```



输出： 8


```
int x;
```

```
void cude(void)
```

```
{ x=x*x*x ;
```

x为0

```
}
```

```
void main (void)
```

```
{ int x=5;
```

```
cude ( );
```

```
cout<<x<<endl;
```

```
}
```

输出: 125

输出: 5

在块作用域内可通过作用域运算符“::”来引用与局部变量同名的全局变量。

```
#include <iostream.h>

int i= 100;                                ::i=104

void main(void)                             i=18
{
    int i , j=50;                          j=108
    i=18;      //访问局部变量i
    ::i= ::i+4; //访问全部变量i
    j= ::i+i;  //访问全部变量i和局部变量j
    cout<<"::i="<<::i<<"\n";
    cout<<"i="<<i<<"\n";
    cout<<"j="<<j<<"\n";
}
```

函数原型作用域

在函数原型的参数表中说明的标识符所具有的作用域称为函数原型作用域，它从其说明处开始，到函数原型说明的结束处结束。

float tt(int x , float y); //函数tt的原型说明

由于所说明的标识符与该函数的定义及调用无关，所以，可以在函数原型说明中只作参数的类型说明，而省略参量名。

float tt (int , float);

```

int i=0;

int workover(int i)
{
    i=(i%i)*((i*i)/(2*i)+4);
    cout<<"i="<<i<<endl;
    return i;
}

int rest (int i)
{
    i=i<2?5:0;
    return i;
}

```

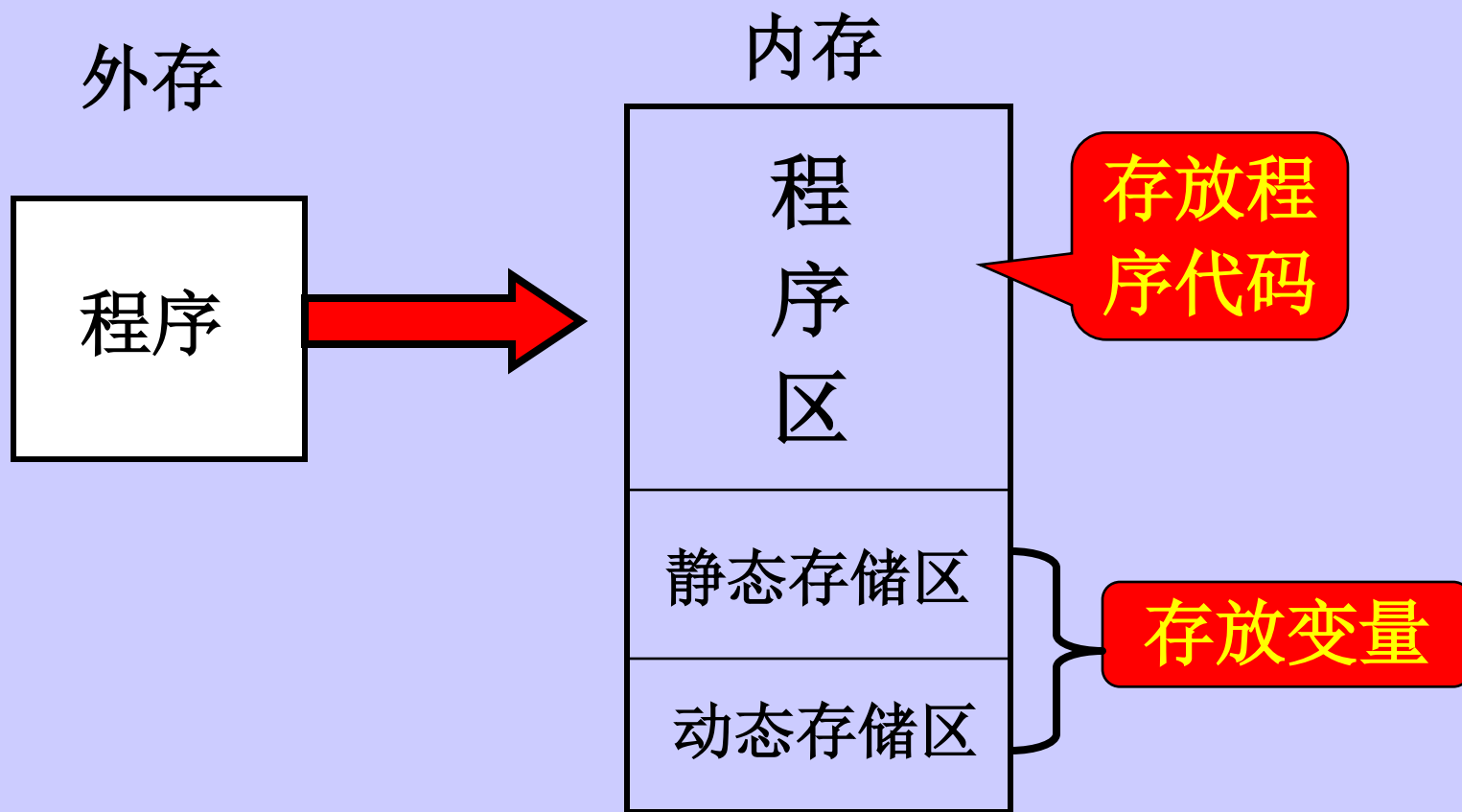
i=5
 i=2
 i=5
 i=0
 i=5

```

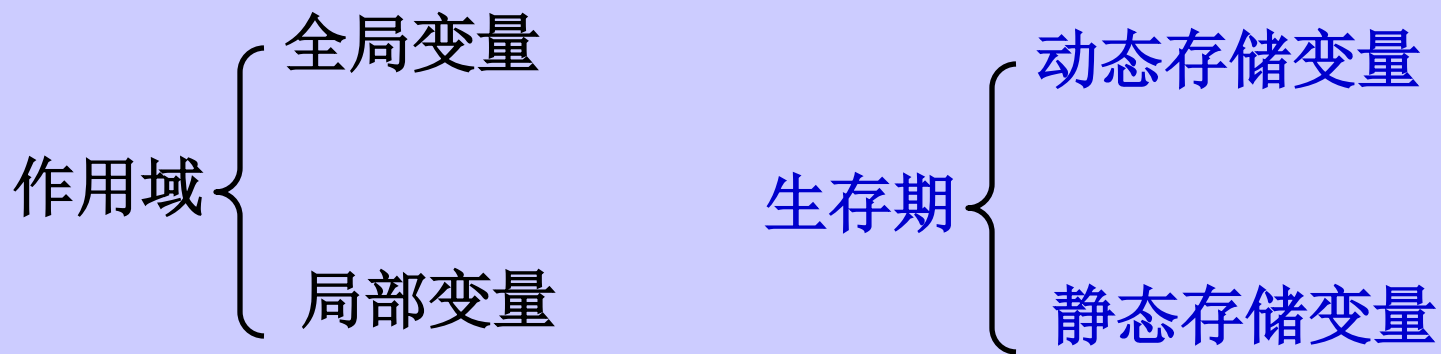
void main(void)
{
    int i=5;
    rest(i/2);
    cout<<"i="<<i<<endl;
    rest(i=i/2);
    cout<<"i="<<i<<endl;
    i= rest(i/2);
    cout<<"i="<<i<<endl;
    workover(i)
    cout<<"i="<<i<<endl;
}

```

存储类

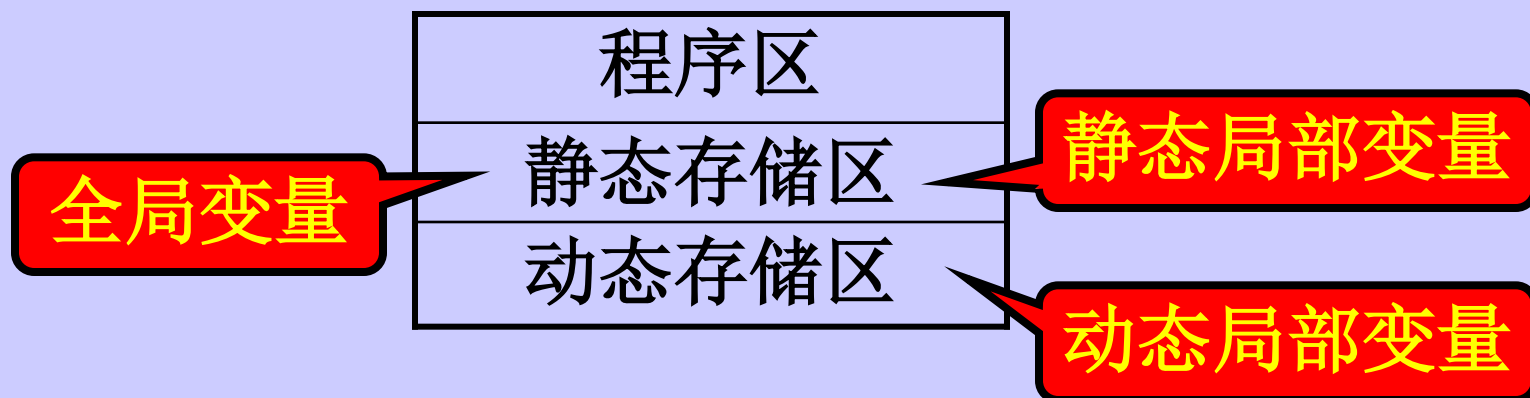


需要区分变量的存储类型



静态存储: 在文件运行期间有固定的存储空间，直到文件运行结束。

动态存储: 在程序运行期间根据需要分配存储空间，**函数结束后立即释放空间**。若一个函数在程序中被调用两次，则每次分配的单元有可能不同。



局部变量的分类

动态变量（**auto**）：默认，存储在动态区

寄存器变量（**register**）：在cpu内部存储

静态局部变量（**static**）：存储在静态区

动态局部变量未被赋值时，其值为随机值。其作用域的函数或复合语句结束时，空间被程序收回。

程序执行到静态局部变量时，为其在静态区开辟存储空间，该空间一直被保留，直到程序运行结束。

由于存储在静态区，静态局部变量或全局变量未赋初值时，系统自动使之**为0**。

```
int fun(int a)
```

```
{ int c;
```

```
  static int b=3;
```

```
  c=a+ b++;
```

```
  return c;
```

```
}
```

```
void main(void)
```

```
{ int x=2, y;
```

```
  y=fun(x);
```

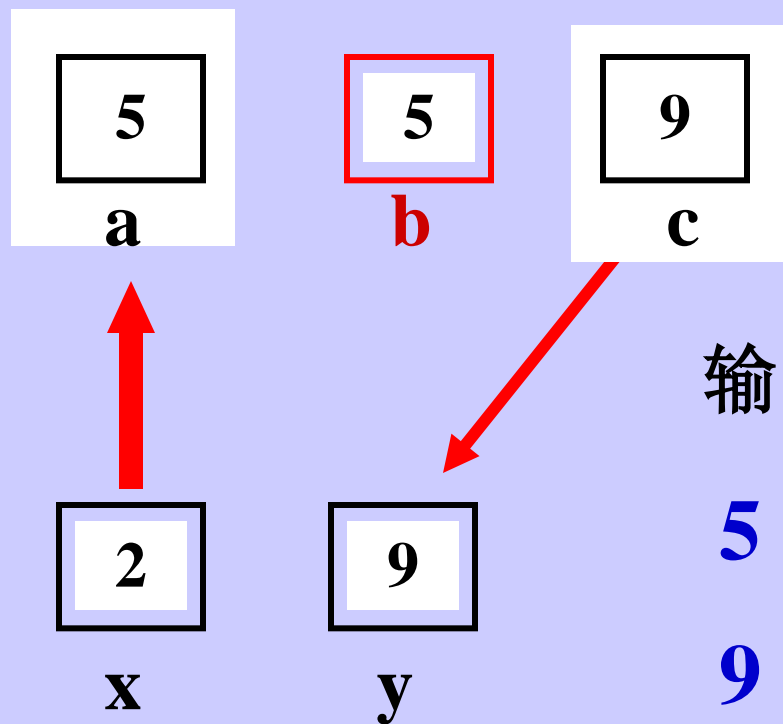
```
  cout<<y<<endl;
```

```
  y=fun(x+3);
```

```
  cout<<y<<endl;
```

```
}
```

只赋一次初值



输出:

5

9

变量b是静态局部变量，在内存一旦开辟空间，就不会释放，空间值一直保留


```

int f (int a)
{ int b=0;
  static int c=3;
  b=b+1;
  c=c+1;
  return a+b+c;
}

```

只赋一次初值

```

void main(void)
{ int a=2,i;
  for (i=0;i<3;i++)
    cout<<f(a)<<endl;
}

```

i=0	a=2	b=0, b=1	输出: 7
		c=3, c=4	
i=1	a=2	b=0, b=1	输出: 8
		c=4, c=5	
i=2	a=2	b=0, b=1	输出: 9
		c=5, c=6	

7

8

9

```
int func (int a, int b)
```

```
{ static int m=0, i=2;
```

```
    i+=m+1;
```

```
    m=i+a+b;
```

```
    return m;
```

```
}
```

```
void main(void)
```

输出: 8,17

```
{ int k=4, m=1, p;
```

```
    p=func(k, m);    cout<<p<<endl;
```

```
    p=func(k, m);    cout<<p<<endl;
```

```
}
```

func(4, 1)

a=4, b=1

m=0, i=2

i=3

m=3+4+1=8

func(4, 1)

a=4, b=1

m=8, i=3

i=3+8+1=12

m=12+4+1=17

```
int q(int x)
```

```
{  int y=1;
```

```
    static int z=1;
```

```
    z+=z+y++;
```

```
    return x+z;
```

```
}
```

4

9

18

```
void main(void)
```

```
{  cout<<q(1)<<“\t”;
```

```
    cout<<q(2)<<“\t”;
```

```
    cout<<q(3)<<“\t”;
```

```
}
```

全局变量的存储方式（extern static）

全局变量是在函数的外部定义的，编译时分配在静态存储区，如果未赋初值，其值为0。

1、extern 存储类别

全局变量的默认方式，当在一个文件中要引用另一个文件中的全局变量或在全局变量定义之前要引用它时，可用extern作说明，相当于扩大全局变量的作用域。

2、静态（static）存储类别

它仅能在本文件中引用，即使在其它文件中用extern说明也不能使用。相当于限制了全局变量的作用域范围。

程序的作用是：给定b的值，输入a和m，求 $a \times b$ 和 a^m 的值。

文件file1.c中的内容为：

```
int a;
```

外部全局
变量定义

```
void main(void)
```

```
{ extern int power (int);
```

```
int b=3, c, d, m;
```

```
cin>>a>>m;
```

```
c=a*b;
```

```
cout<<a<<"*"<<b<<"="<<c<<endl;
```

```
d= power(m);
```

```
cout<<a<<"**"<<m<<"="<<d<<endl;
```

```
}
```

外部全局
变量说明

文件file2.c中的内容为：

```
extern int a;
```

```
int power (int n )
```

```
{ int i, y=1;
```

```
for (i=1; i<=n; i++)
```

```
y*=a;
```

```
return y;
```

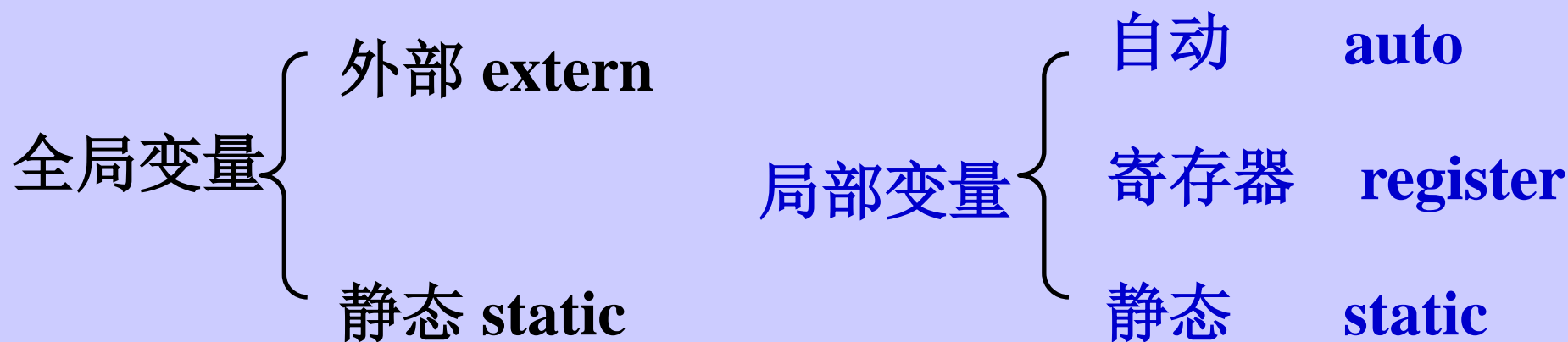
```
}
```

引用文件外定
义的全局变量

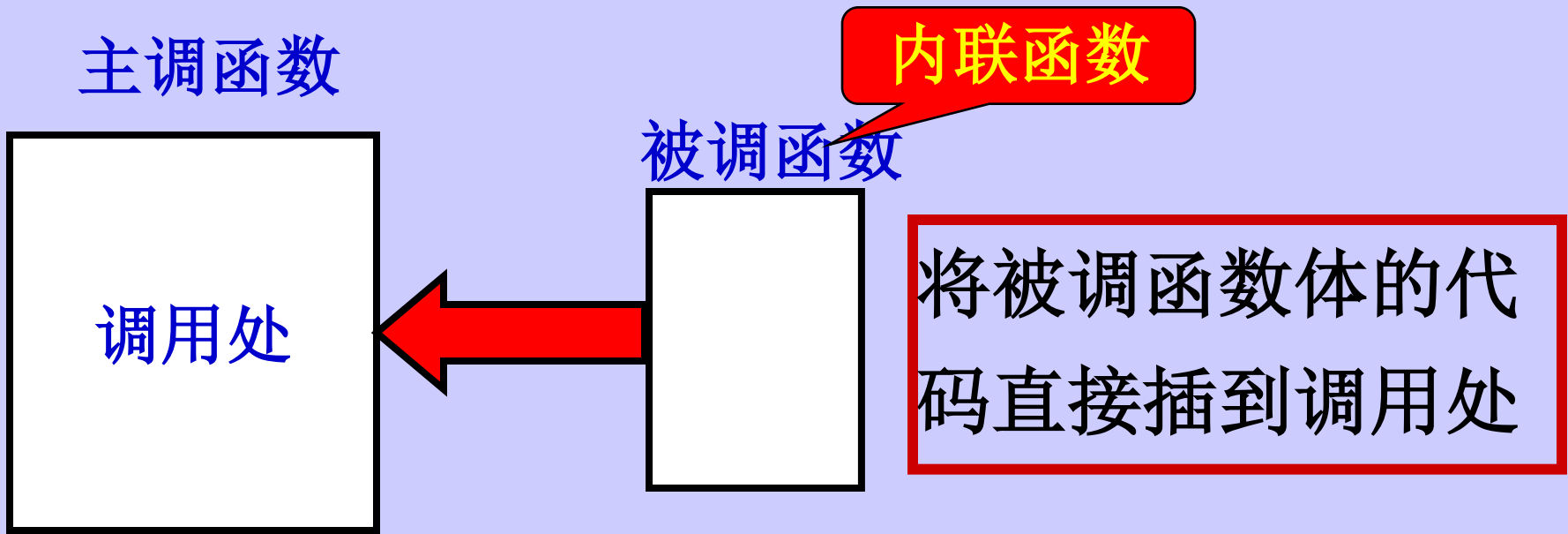
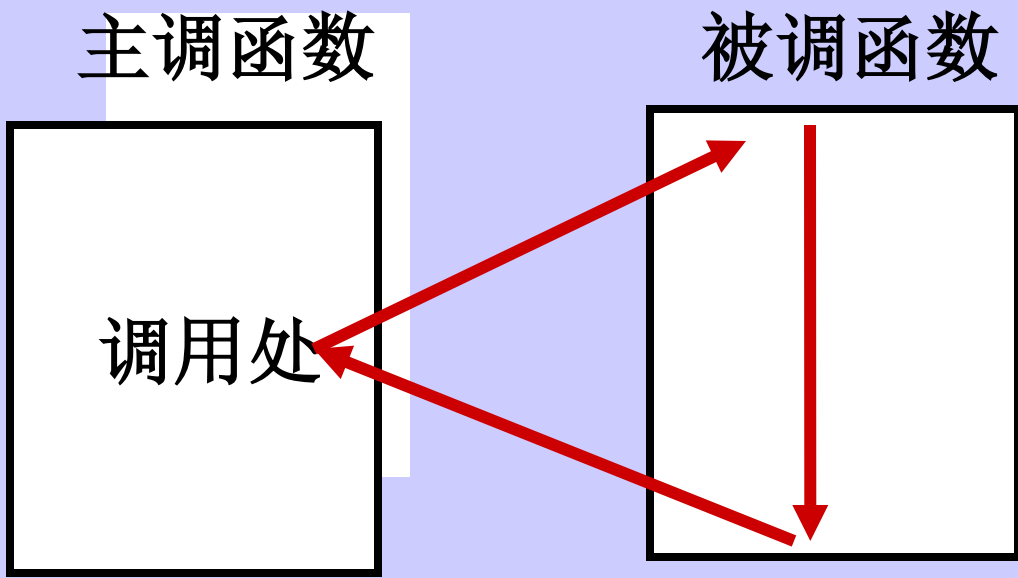
静态局部变量：static 在函数内部定义，存储在静态存储区，与auto对应，在别的函数中不能引用。

全局静态变量：static 在函数外部定义，只限在本文件中使用，与extern对应。

当变量名相同致使作用域相重时，起作用的是最近说明的那个变量。



内联函数



内联函数的实质是用存储空间（使用更多的存储空间）来换取时间（减少执行时间）。

内联函数的定义方法是，在函数定义时，在函数的类型前增加修饰词**inline**。


```
inline int max (int x, int y)
```

```
{ int z;
```

```
    z=(x>y)? x : y ;
```

```
    return z;
```

```
}
```

```
void main (void )
```

```
{ int a,b,c;
```

```
    cin>>a>>b;
```

```
    c=max (a+b , a*b) ;
```

```
    cout<<“The max is”<<c<<endl;
```

```
}
```

使用内联函数时应注意以下几点：

- 1、C++中，除在函数体内含有循环，**switch**分支和复杂嵌套的**if**语句外，所有的函数均可定义为内联函数。
- 2、内联函数也要定义在前，调用在后。形参与实参之间的关系与一般的函数相同。
- 3、对于用户指定的内联函数，编译器是否作为内联函数来处理由编译器自行决定。说明内联函数时，只是请求编译器当出现这种函数调用时，作为内联函数的扩展来实现，而不是命令编译器要这样做。
- 4、正如前面所述，内联函数的实质是采用空间换取时间，即可加速程序的执行，当出现多次调用同一内联函数时，程序本身占用的空间将有所增加。如上例中，内联函数仅调用一次时，并不增加程序占用的存储间。

具有缺省参数值和参数个数可变的函数

在C++中定义函数时，允许给参数指定一个缺省的值。在调用函数时，若明确给出了这种实参的值，则使用相应实参的值；若没有给出相应的实参，则使用缺省的值。（举例说明）

```
int fac(int n=2)
```

```
{ int t=1;
```

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

```
t=t*i;
```

```
return t;
```

```
}
```

```
void main(void)
```

```
{
```

```
cout<<fac( ) <<endl;
```

```
}
```

输出： 720

输出： 2

```
int  area(int long=4 , int width=2)
```

```
{  return long* width;
```

```
}
```

48

```
void  main(void )
```

16

```
{  int  a=8, b=6;
```

8

```
    cout<< area(a,b) <<endl;
```

```
    cout<< area(a) <<endl;
```

```
    cout<< area( ) <<endl;
```

```
}
```

使用具有缺省参数的函数时，应注意以下几点：

1.不可以靠左边缺省

```
int area(int long , int width=2)
```

```
int area(int long =4, int width)
```

错误！

2.函数原型说明时可以不加变量名

```
float v(float,float=10,float=20);
```

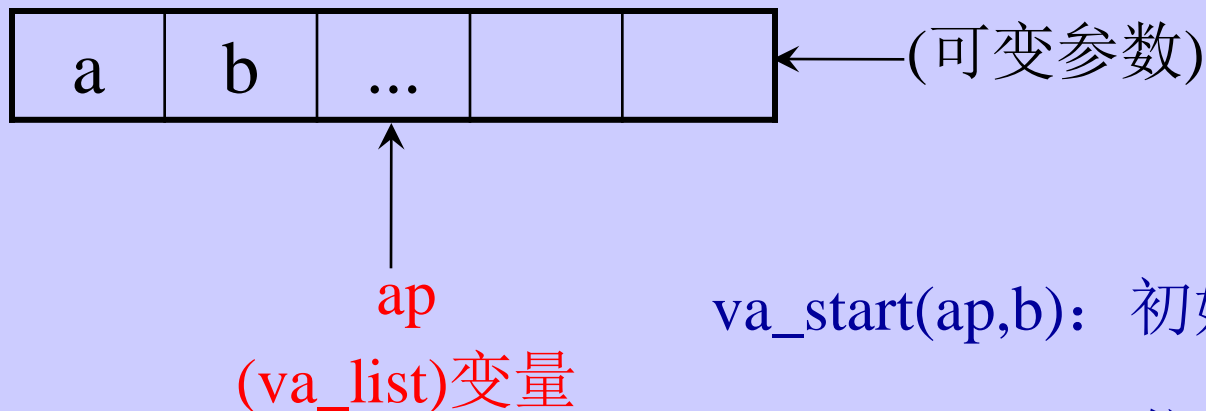
3.只能在前面定义一次缺省值，即原型说明时定义了缺省值，后面函数的定义不可有缺省值。

参数个数可变的函数

到目前为止，在定义函数时，都明确规定了函数的参数个数及类型。在调用函数时，实参的个数必须与形参相同。在调用具有缺省参数值的函数时，本质上，实参的个数与形参的个数仍是相同的，由于参数具有缺省值，因此，在调用时可省略。在某些应用中，在定义函数时，并不能确定函数的参数个数，参数的个数在调时才能确定。在C++中允许定义参数个数可变的函数。

首先，必须包含头文件“stdarg.h”，因为要用到里面的三个库函数va_start()、va_arg()和va_end()。

其次，要说明一个va_list类型的变量。va_list与int，float类同，它是C++系统预定义的一个数据类型(非float)，只有通过这种类型的变量才能从实际参数表中取出可变有参数。如：va_list ap;



va_start(ap,b): 初始化

va_arg(ap,int): 依次取参数

va_end(ap): 正确结束

`va_start()`:有两个参数, `va_start(ap,b)`; `b`即为可变参数前的最后一个确定的参数。

`va_arg()`:有两个参数, `va_arg(ap,int)` `int`即为可变参数的数据类型名。

```
int temp;
```

```
temp=va_arg(ap,int);
```

`va_end()`:完成收尾工作。`va_end(ap)`;

在调用参数个数可变的函数时, 必定有一个参数指明可变参数的个数或总的实参个数。如第一个参数值为总的实际参数的个数。

使用参数数目可变的函数时要注意以下几点：

1、在定义函数时，固定参数部分必须放在参数表的前面，可变参数在后面，并用省略号“...”表示可变参数。在函数调用时，可以没有可变的参数。

2、必须使用函数va_start()来初始化可变参数，为取第一个可变的参数作好准备工作；使用函数va_arg()依次取各个可变的参数值；最后用函数va_end()做好结束工作，以便能正确地返回。

3、在调用参数个数可变的函数时，必定有一个参数指明可变参数的个数或总的实参个数。

函数的重载

所谓函数的重载是指完成不同功能的函数可以具有**相同的函数名**。

C++的编译器是根据**函数的实参**来确定应该调用哪一个函数的。

```
int fun(int a, int b)
```

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

```
int fun (int a)
```

```
{ return a*a; }
```

```
void main(void)
```

```
{ cout<<fun(3,5)<<endl;
```

```
cout<<fun(5)<<endl;
```

```
}
```

8

25

1、定义的重载函数必须具有不同的参数个数，或不同的参数类型。只有这样编译系统才有可能根据不同的参数去调用不同的重载函数。

2、仅返回值不同时，不能定义为重载函数。

即仅函数的类型不同，不能定义为重载函数

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

float fun (int a,int b)
{ return (float) a*a; }
```

```
void main(void)
{ cout<<fun(3,5)<<endl;
  cout<<fun(3,5)<<endl;
}
```

```
double sin(double x1,double x2)
```

```
{    return x1*x2;}
```

sin(x,x)

```
double sin(double x,int a)
```

```
{    return a+x;}
```

sin(x,10)

```
void main(void)
```

```
{    double x;
```

```
    cin>>x;
```

```
    cout<<sin(x)<<endl;;
```

```
    cout<<sin(x,x)<<endl;
```

```
    cout<<sin(x,10)<<endl;
```

```
}
```

**不同的参
数类型**

```
int add(int a,int b,int c)
```

```
{    return a+b+c; }
```

```
int add(int a,int b)
```

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

不同的参
数个数

```
void main(void)
```

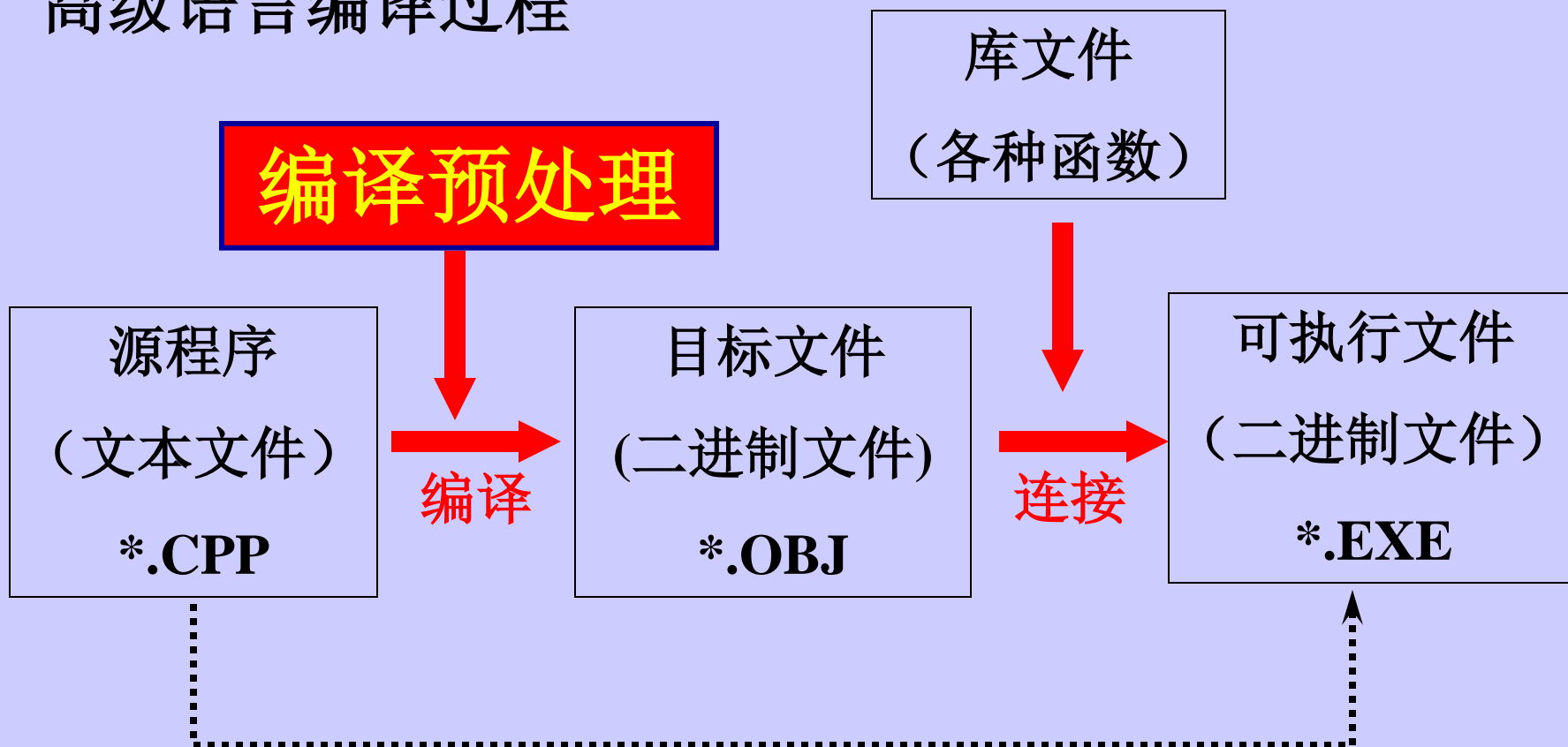
```
{    cout<<"3+5="<<add(3,5)<<endl;
```

```
    cout<<"3+5+8="<<add(3,5,8)<<endl;
```

```
}
```

编译预处理

高级语言编译过程



C语言提供的编译预处理的功能有以下三种：

宏定义

文件包含

条件编译

宏定义

不带参数的宏定义

用一个指定的标识符（即名字）来代表一个字符串，以后凡在程序中碰到这个标识符的地方都用**字符串**来代替。

这个标识符称为**宏名**，**编译前**的替代过程称为“**宏展开**”。

define 标识符 字符串


```
#define PRICE 30
```

```
void main(void)
```

```
{ int num, total; /*定义变量*/
```

```
    num=10;    /*变量赋值*/
```

```
    total=num*PRICE;
```

编译前用30替代

```
    cout<<"total="<<total<<endl;
```

```
}
```

编译程序将宏定义的内容认为是字符串，没有任何实际的物理意义。

注意：

1、宏展开只是一个简单的“**物理**”替换，不做**语法检查**，不是一个语句，**其后不加分号“；”**

2、**#define**命令出现在函数的外面，其有效范围为定义处至本源文件结束。可以用**# undef**命令终止宏定义的作用域。

```
#define G 9.8
```

```
void main(void )
```

```
{.....}
```

```
# undef G
```

```
int max(int a,int b)
```

```
{..... }
```

3、对程序中用双引号括起来的字符串内容，即使与宏名相同，也不进行置换。

4、在进行宏定义中，可以用已定义的宏名，进行层层置换。

```
# define R 3.0
```

```
# define PI 3.1415926
```

```
# define L 2*PI*R
```

层层置换

```
# define S PI*R*R
```

层层置换

```
void main(void)
```

```
{    cout<<"L="<<L<<" S="<<S<<endl;
```

不置换

不置换

```
}
```

带参数的宏定义

#define 宏名(参数表) 字符串

#define

S(a, b)

a*b

... **宏定义**

形式参数

定义的宏

float x, y, area;

cin >> x >> y;

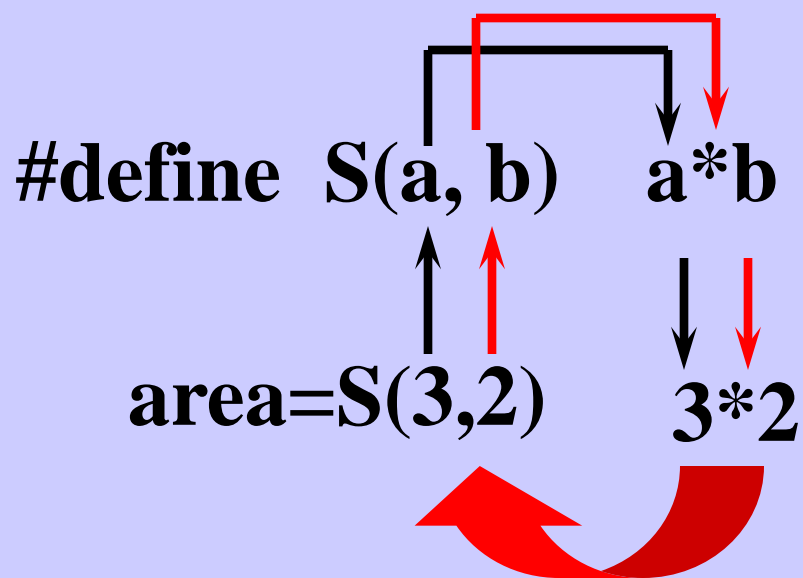
实参代入后还原

area = S(x, y); /* area = x*y; */

宏调用

实际参数

按**#define**命令行中指定的字符串从左至右进行置换宏名，字符串中的**形参以相应的实参代替**，字符串中的非形参字符保持不变。



机械地将实参代入宏定义的形参形式

S(a,b)等同于 a*b

S(3,2)等同于 3*2

```
#define PI 3.1415926
```

```
#define S(r) PI*r*r
```

$S(r) \longrightarrow PI*r*r$

```
void main(void)
```

$S(a) \longrightarrow PI*a*a$

```
{ float a, area, b;
```

```
    a=3.6; b=4.0;
```

```
    area=PI*a*a
```

编译前机械替换，
实参形参一一对应

```
    cout<<"r="<<a<<"\narea="<<area<<endl;
```

```
}
```

```
#define PI 3.1415926
```

$S(r) \longrightarrow PI * r * r$

```
#define S(r) PI*r*r
```

$S(a+b) \longrightarrow PI * a + b * a + b$

```
void main(void)
```

错误

```
{ float a, area, b;
```

```
    a=1; b=2;
```

```
    area=S(a+b);
```

```
    cout<<"r="<<a<<"\narea="<<area<<endl;
```

```
}
```

```
#define S(r) PI*(r)*(r)
```

编译前机械替换，
实参形参一一对应

宏展开时实参不运
算，不作语法检查

定义宏时在宏名与带参数的括弧间不能有空格。

```
#define S_(r) P*r*r
```

带参数的宏与函数调用的区别
相同：有实参、形参，代入调用。

不同之处：

1、函数调用先求表达式的值，然后代入形参，而宏只是机械替换。

2、函数调用时形参、实参进行类型定义，而宏不需要，只是作为字符串替代。

3、函数调用是在运行程序时进行的，其目标代码短，但程序执行时间长。而宏调用是在编译之前完成的，运行时已将代码替换进程序中，目标代码长，执行时间稍快。

一般用宏表示实时的、短小的表达式。

```
#define A 3
```

```
#define B(a) ((A+1)*a)
```

93

执行 $x=3*(A+B(7))$; 后, x 的值为:

```
#define neg(x) ((-x)+1)
```

```
int neg( int x)
```

```
{return x+1; }
```

```
void main(void)
```

```
{ int y;
```

```
  y= $((-1)+1)$ 
```

```
  cout<<"y="<<y<<endl;
```

```
}
```

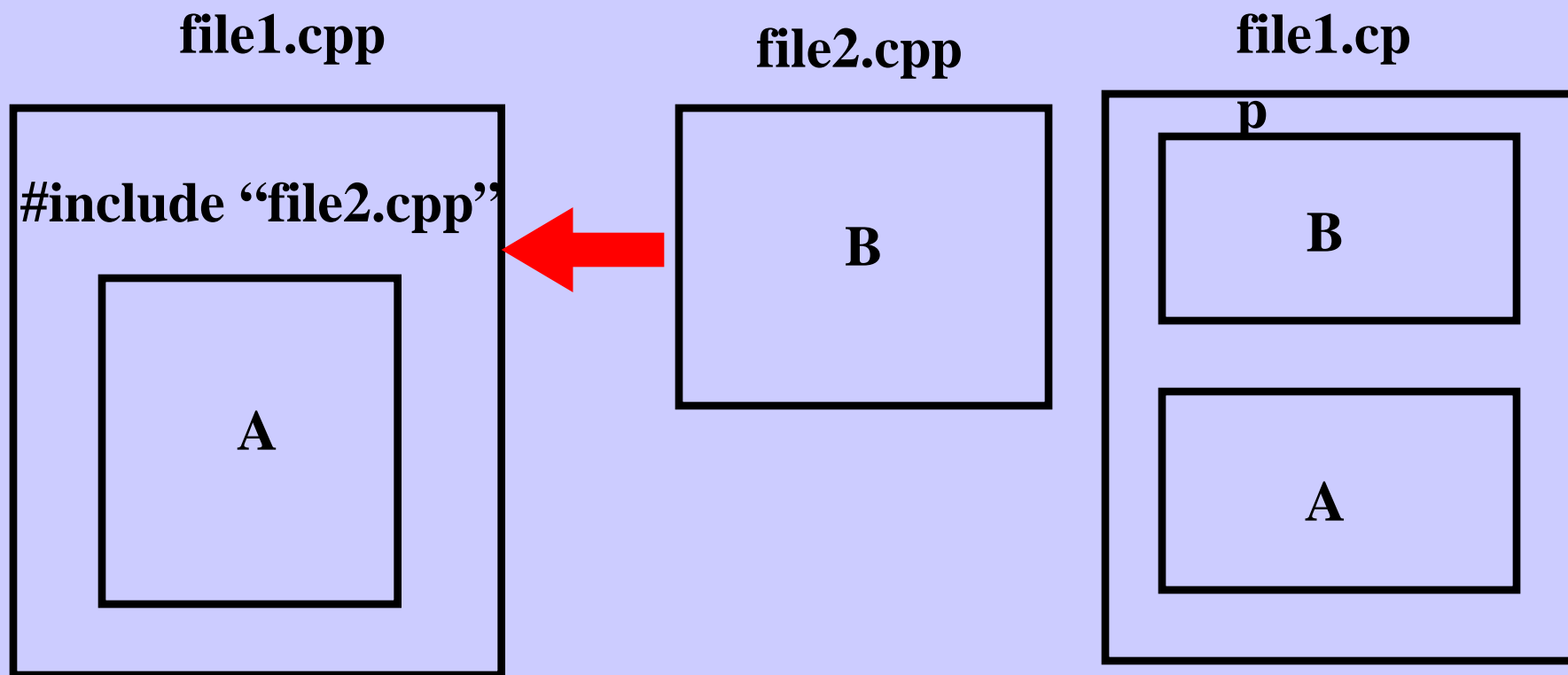
编译前机械替换,
实参形参一一对应

$y=0$

文件包含

一个源文件可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。

include “文件名”



注意：

- 1、文件名是C的源文件名，是文本文件，后缀名可以任选。`*.cpp *.h`
- 2、一个`#include`语句只能指定一个被包含文件。
- 3、文件名用双引号或尖括号括起来。
- 4、包含后所有源文件编译为一个可执行文件。

条件编译

C语言允许有选择地对程序的某一部分进行编译。
也就是对一部分源程序指定编译条件。

源程序



可以将部分源程序
不转换为机器码

条件编译有以下几种形式:

标识符

1、 **# ifdef** 标识符

define **DEBUG**

程序段1

.....

else

ifdef **DEBUG**

程序段2

cout<<x<<'\\t'<<y<<endl;

end if

endif

当标识符已被定义过（用**#define**定义），则对程序段1进行编译，否则编译程序段2.

2、 **# ifndef** 标识符

程序段1

else

程序段2

endif

define DEBUG

.....

ifndef DEBUG

cout<<x<<'\t'<<y<<endl;

endif

与形式1相反，当标识符没有被定义过（用**#define**定义），则对程序段1进行编译，否则编译程序段2。

调试完后加**#define DEBUG**，则不输出调试信息。

3、 **# if** 表达式

程序段1

else

程序段2

endif

define DEBUG 1

.....

if DEBUG

cout<<x<<'\t'<<y<<endl;

endif

当表达式为真(非零),
编译程序段1, 表达式
为零, 编译程序段2。

调试完后改为 **#define DEBUG**
0, 则不输出调试信息。

采用条件编译后, 可以使机器代码程序缩短。

以下程序的运行结果是：

```
#define DEBUG
```

```
void main(void)
```

```
{ int a=14, b=15, c;
```

```
c=a/b;
```

```
# ifdef DEBUG
```

```
cout<<"a="<<oct<<a<<" b="<<b<<endl;
```

```
# endif
```

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

```
}
```

输出： a=16, b=17c=0

程序的多文件组织

而在设计一个功能复杂的大程序时，为了便于程序的设计和调试，通常将程序分成若干个模块，把实现一个模块的程序或数据放在一个文件中。当一个完整的程序被存放在多于一个文件中时，称为程序的多文件组织。

内部函数和外部函数

内部函数：函数只限于在本文件中调用，其它文件不能调用，用**static** 定义该函数。

```
static float fac( int n)
{ ..... }
```

外部函数：函数的默认形式，可以被其它文件调用，用**extern** 定义该函数。调用时，在文件中用**extern** 说明。

```
void main(void)
{ extern enter_string( );
  char str[80];
  enter_string(str);
  .....
}
```

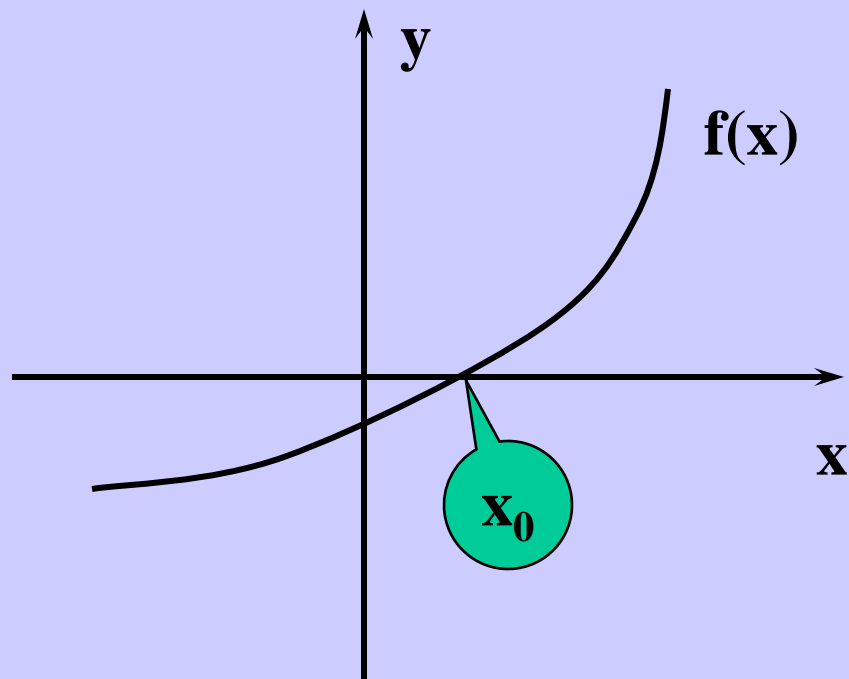
说明外部函数

补充算法

方程求解

1、牛顿切线法

只有为数不多的方程有精确解，一般都是用迭代方法近似求方程的解。方程 $f(x)=0$ 的实数解实际上是曲线 $f(x)$ 在 x 轴上交点的值。

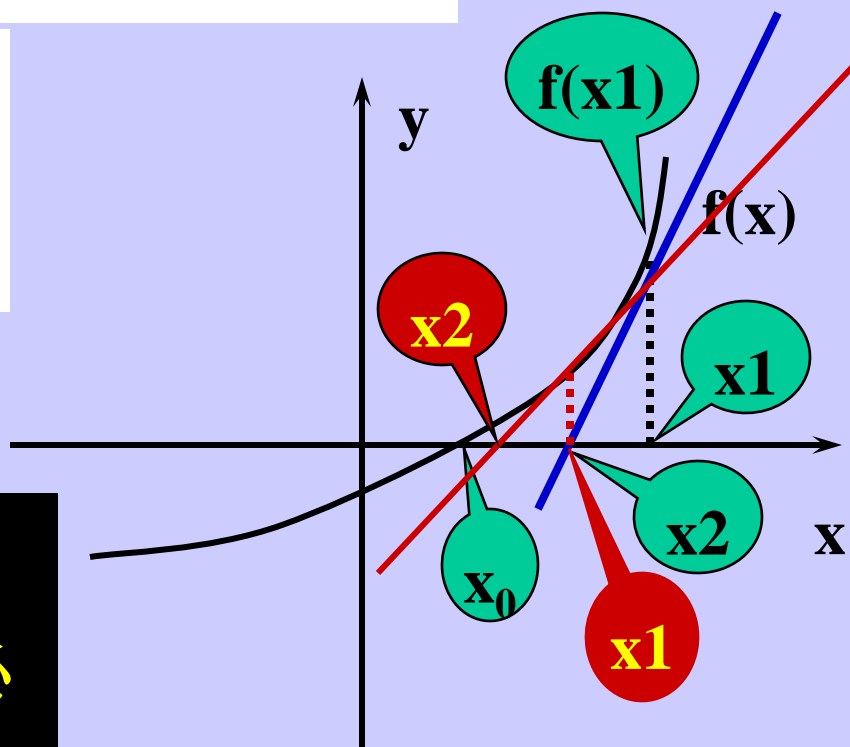


- 1、任选一 x 值 x_1 ,在 $y_1=f(x_1)$ 处做切线与 x 轴相交于 x_2 处。
- 2、若 $|x_2-x_1|$ 或 $|f(x_2)|$ 小于指定的精度,则令 $x_1=x_2$,继续做1。当其满足所需的精度时, x_2 就是方程的近似解。

根据已知点求其切线的公式为:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

这就是牛顿切线法。



牛顿切线法收敛快,
适用性强, 缺陷是必须
求出方程的导数。

已知方程为 $f(x)=x*x-a$ 时，用牛顿切线法求方程的解。给定初值 x_0 ，精度 10^{-6} ，算法编程如下。

```
cin>>x1; //从键盘输入x0
```

```
do
```

上一循环的新值成为本次循环的旧值

```
{ x0=x1;
```

旧值算本次循环的新值

```
x1=x0-(x0*x0-a)/(2*x0); //
```

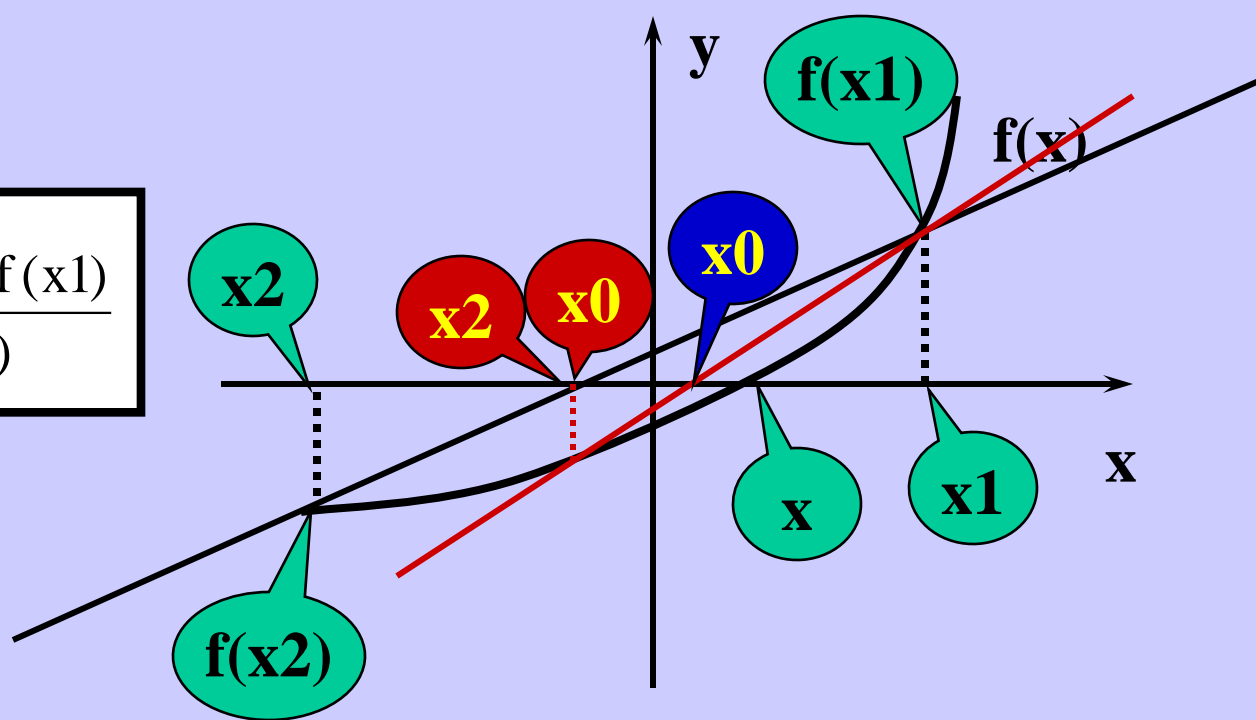
$$x1 = x0 - \frac{f(x0)}{f'(x0)}$$

```
} while (fabs(x1-x0)>=1e-6);
```

```
cout>>"x=">>x1>>endl;
```

2、弦截法

$$x_0 = \frac{x_1 * f(x_2) - x_2 * f(x_1)}{f(x_2) - f(x_1)}$$



- 1、在x轴上取两点x1和x2, 要确保x1与x2之间有且只有方程唯一的解。
- 2、x1与x2分别与f(x)相交于y1=f(x1)、y2=f(x2)。
- 3、做直线通过y1、y2与x轴交于x0点。
- 4、若|f(x0)|满足给定的精度, 则x0即是方程的解, 否则, 若f(x0)*f(x1)<0, 则方程的解应在x1与x0之间, 令x2=x0, 继续做2。同理, 若f(x0)*f(x1)>0, 则方程的解应在x2与x0之间, 令x1=x0, 继续做2, 直至满足精度为止。

用两分法求方程的根。

$$x^3 - 5x^2 + 16x - 80 = 0$$

$$x0 = \frac{x1 * f(x2) - x2 * f(x1)}{f(x2) - f(x1)}$$

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

```
float f (float x)
```

输入x, 输出f(x)

```
{return x*x*x-5*x*x+16*x-80;
```

```
}
```

判断输入是否合法

输入x1,x2, 输出x0

```
float xpoint(float x1,float x2)
```

```
{ float x0;
```

```
x0=(x1*f(x2)-x2*f(x1))/(f(x2)-f(x1));
```

```
return x0;
```

```
}
```

```
void main(void )
```

```
{ float x1,x2, x0, f0, f1, f2;
```

```
do
```

```
{ cout<<"Input x1, x2\n";
```

```
cin>>x1>>x2;
```

```
f1=f(x1); f2=f(x2);
```

```
} while (f1*f2>0);
```

```
do
```

```
{ x0=xpoint(x1,x2);
```

```
f0=f(x0);
```

```
if ((f0*f1) >0) { x1=x0;f1=f0;}
```

```
else { x2=x0; f2=f0;}
```

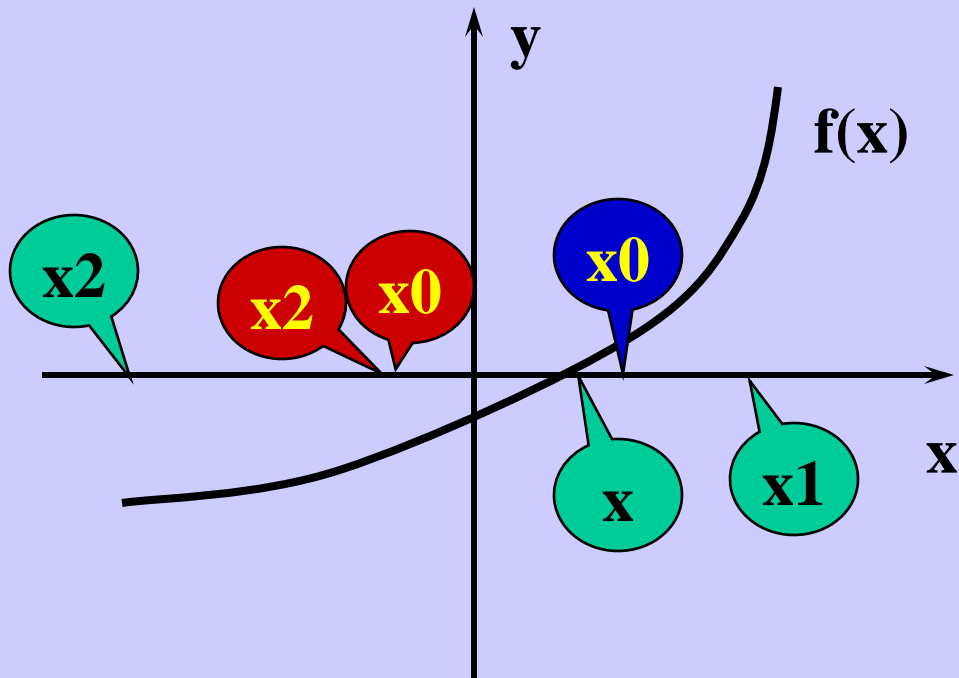
```
}while (fabs(f0)>=0.0001);
```

```
cout<<"x="<< x0<<endl;
```

```
}
```


3、两分法

$$x_0 = (x_1 + x_2) / 2$$



1、在x轴上取两点 x_1 和 x_2 , 要确保 x_1 与 x_2 之间有且只有方程唯一的解。

2、求出 x_1, x_2 的中点 x_0 。

3、若 $|f(x_0)|$ 满足给定的精度, 则 x_0 即是方程的解, 否则, 若 $f(x_0) \cdot f(x_1) < 0$, 则方程的解应在 x_1 与 x_0 之间, 令 $x_2 = x_0$, 继续做2。同理, 若 $f(x_0) \cdot f(x_1) > 0$, 则方程的解应在 x_2 与 x_0 之间, 令 $x_1 = x_0$, 继续做2, 直至满足精度为止。

用两分法求方程的根。

$$x^3 - 5x^2 + 16x - 80 = 0$$

$$x0 = (x1 + x2) / 2$$

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

```
float f (float x)
```

输入x, 输出f(x)

```
{return x*x*x-5*x*x+16*x-80;
```

```
}
```

判断输入是否合法

求x1与x2的中点

```
void main(void )
```

```
{ float x1,x2, x0, f0, f1, f2;
```

```
do
```

```
{ cout<<"Input x1, x2\n";
```

```
cin>>x1>>x2;
```

```
f1=f(x1); f2=f(x2);
```

```
} while (f1*f2>0);
```

```
do
```

```
{ x0=(x1+x2)/2;
```

```
f0=f(x0);
```

```
if ((f0*f1) >0){ x1=x0;f1=f0;}
```

```
else { x2=x0;f2=f0;}
```

```
}while (fabs(f0)>=0.0001);
```

```
cout<<"x="<< x0<<endl;
```

```
}
```

```
int q(int x)
```

```
{  int y=1;
```

```
    static int z=1;
```

```
    z+=z+y++;
```

```
    return x+z;
```

```
}
```

4

9

18

```
void main(void)
```

```
{  cout<<q(1)<<"\t";
```

```
    cout<<q(2)<<"\t";
```

```
    cout<<q(3)<<"\t";
```

```
}
```

下面函数pi的功能是：根据以下公式，返回满足精度（0.0005）要求的 π 的值，请填空。

$$\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1}{3} \frac{2}{5} + \frac{1}{3} \frac{2}{5} \frac{3}{7} + \frac{1}{3} \frac{2}{5} \frac{3}{7} \frac{4}{9} + \dots$$

将后一项除
以前一项，
找规律

输入精度
输出 π

```
#include "math.h"
double pi(double eps)
{ double s, t; int n;
  for ( t=1,s=0,n=1; t>eps; n++)
  { s+=t;
    t=n*t/(2*n+1);
  }
  return 2*s;
}
```

```
main( )
{ double x;
  cout<<"\nInput a precision:";
  cin>>x;
  cout<< "π="<<pi(x);
}
```

```
void f(int n)
```

```
{ if(n>=10)
```

1

```
    f(n/10);
```

12

```
    cout<<n<<endl;
```

123

```
}
```

1234

```
void main(void)
```

12345

```
{ f(12345);
```

```
}
```

```
void main(void)
```

```
{ char s; cin.get(s);
```

输入: 2347<CR>

```
while(s!='\n')
```

```
{ switch(s-'2')
```

```
{ case 0:
```

5454555555657

```
case 1: cout<<s+4;
```

```
case 2: cout<<s+4;break;
```

```
case 3: cout<<s+3;
```

```
default: cout<<s+2; break;
```

```
}
```

```
cin.get(s);
```

```
}cout<<endl;
```

```
}
```

第六章 数组

一维数组的定义和引用

数组是同一类型的一组值（10个 char 或15个 int），在内存中顺序存放。

整个数组共用一个名字，而其中的每一项又称为一个元素。

一、定义方式：

类型说明符 数组名[常量表达式];

定义类型

元素个数

int a[4]; // 表明a数组由4个int型元素组成

数组名称

必须是常数

int a[4]; // 表明a数组由4个int型元素组成

其元素分别为: a[0], a[1], a[2], a[3]

其序号从0开始。若存放首地址为2000H, 则在内存中为:

2000H	2004H	2008H	200CH	2010H
a[0]	a[1]	a[2]	a[3]	

C++不允许对数组的大小作动态的定义, 即数组的大小不能是变量, 必须是常量。

如果要根据不同的数值改变数组的大小，可用常量表达式。如：

```
#define SIZE 50
```

```
void main(void)
```

```
{ int art[SIZE];
```

```
.....
```

```
}
```

二、一维数组元素的引用

数组必须先定义，具体引用时（赋值、运算、输出）其元素等同于变量。

```
void main(void )
```

```
{  int i, a[10];
```

定义

```
    for ( i=0; i<10; i++)
```

赋值

```
        a[i]=i;
```

```
    for ( i=9; i>=0 ; i--)
```

```
        cout<<a[i]<<'\t';
```

```
    cout<<"\n";
```

输出

```
}
```

i=0, a[0]=0

i=1, a[1]=1

i=2, a[2]=2

i=9, a[9]=9

a

0	a[0]
1	a[1]
2	a[2]
3	a[3]
4	a[4]
5	a[5]
6	a[6]
7	a[7]
8	a[8]
9	a[9]

输出： 9 _ 8 _ 7 _ 6 _ 5 _ 4 _ 3 _ 2 _ 1 _ 0

三、一维数组的初始化

在定义数组的同时给数组元素赋值。

注意：

1、对数组中的一部分元素列举初值，未赋值的部分是0。

```
int a[10]= {0,1, 2, 3, 4, 5};
```

```
int a[10]= {0,1, 2, 3, 4, 5, 0, 0, 0, 0};
```

2、不能给数组整体赋值，只能一个一个地赋值。

```
int a[10]= {0,1,2,.....,9};
```

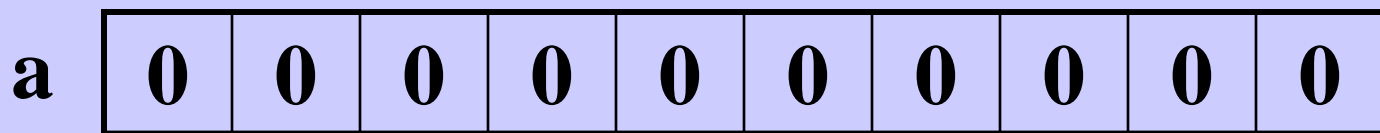
非法

```
int a[10]= {0,1, 2, 3, 4, 5,6,7,8,9};
```

3、可以用 `int a[] = {0,1,2,3,4,5,6,7,8,9};` 给数组赋值，编译器会自动计算出内的元素项数，并将数组定义为该长度。

4、用局部static 或全局定义的数组不赋初值，系统均默认其为 `'\0'`。

`static int a[10];`（即存储在静态数据区中的数组其元素默认为0）



数组在内存中顺序存放，第一个元素位于地址的最低端。

求Fibonacci数列：1,1,2,3,5,8,.....的前20个数，即

$$F1=1 \quad (n=1)$$

$$F2=1 \quad (n=2)$$

$$Fn=Fn-1+Fn-2 \quad (n \geq 3)$$

f[0] f[1] f[2] f[3] f[4] f[5] f[5] f[6] f[7]

1	1	2	3	5	8	13	21		
----------	----------	----------	----------	----------	----------	-----------	-----------	--	--

$$f[i]=f[i-1]+f[i-2]$$

```
void main (void)
```

```
{ int i;
```

```
int f [20]={1,1};
```

```
for (i=2 ; i<20 ; i++ )
```

```
    f [i]=f [i-1]+f [i-2];
```

```
for ( i=0; i<20; i++)
```

```
{ if (i%5==0) cout<<"\n";
```

```
    cout<<f [i]<<"\t";
```

```
}
```

```
}
```

下面程序的运行结果是：

```
void main(void)
```

```
{ int a[6], i;
```

```
  for (i=1; i<6; i++)
```

```
  { a[i]=9*(i-2+4*(i>3))%5 ;
```

```
    cout<<a[i]<<'\t';
```

```
  }
```

```
}
```

a[0] a[1] a[2] a[3] a[4] a[5]

随机	-4	0	4	4	3
----	----	---	---	---	---

i	1	2	3	4	5
a[i]	-4	0	4	4	3

输出： -4 0 4 4 3

排序算法

用起泡法对6个数排序（由小到大）

将相邻的两个数两两比较，将小的调到前头。

9	8	8	8	8	8	8	5	5	5	5	5	4	4	4
8	9	5	5	5	5	5	8	4	4	4	4	5	3	3
5	5	9	4	4	4	4	4	8	2	3	3	3	5	0
4	4	4	9	2	2	2	2	2	8	0	0	0	0	5
2	2	2	2	9	0	0	0	0	0	8	8	8	8	8
0	0	0	0	0	9	9	9	9	9	9	9	9	9	9

第一趟

循环5次

第二趟

循环4次

第三趟

循环3次

4 3 3

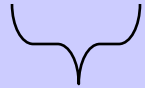
3 4 0

0 0 4

5 5 5

8 8 8

9 9 9



第四趟

循环2次

3 0

0 3

4 4

5 5

8 8

9 9

第五趟

循环1次

总结:

	共有6个数					n
趟数	1	2	3	4	5	$j(1 \sim n-1)$
次数	5	4	3	2	1	$n-j$

```
for (j=1; j<= $n-1$ ; j++)  
    for (i=1; i<= $n-j$  ; i++)  
    { if (a[i]>a[i+1])  
        {  $t=a[i]$ ;  
           $a[i]=a[i+1]$ ;  
           $a[i+1]=t$ ;  
        }  
    }  
}
```

一般，元素的序号从0开始，因此，程序可以变动如下：

```
for (j=0; j<n-1; j++)
```

```
    for (i=0; i<n-1-j; i++)
```

```
        { if (a[i]>a[i+1])
```

```
            { t=a[i];
```

```
              a[i]=a[i+1];
```

```
              a[i+1]=t;
```

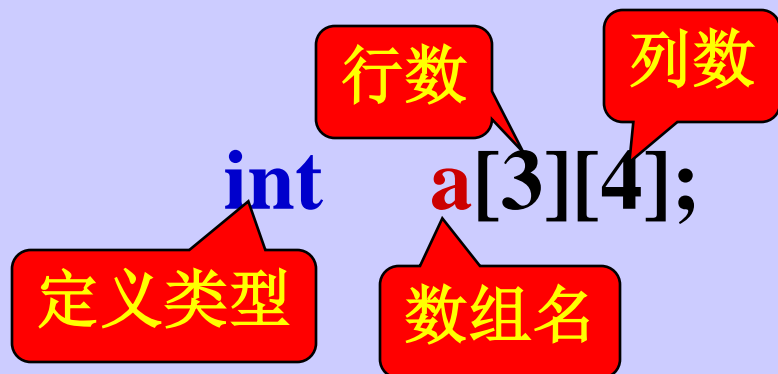
```
            }
```

```
        }
```

二维数组的定义和引用

一、定义方式:

类型说明符 数组名[常量表达式][常量表达式];


int a[3][4];

表明a数组由 3×4 个int型元素组成

其元素分别为: a[0][0], a[0][1], a[0][2], a[0][3],
a[1][0], a[1][1], a[1][2], a[1][3],
a[2][0], a[2][1], a[2][2], a[2][3]

其元素分别为：**a[0][0], a[0][1], a[0][2], a[0][3],**
a[1][0], a[1][1], a[1][2], a[1][3],
a[2][0], a[2][1], a[2][2], a[2][3]

其行列的序号均**从0开始**。若存放首地址为2000H，则在内存中为：

2000H		2008H		2010H		2014H		201cH		2020H		2028H		202cH	
a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]				

即在内存中，多维数组依然是**直线顺序**排列的，**第**
一个元素位于最低地址处。

二、二维数组的引用

与一维数组一样，二维数组必须先定义，其维数必须是常量。具体引用时（赋值、运算、输出）其元素等同于变量。

输入：1 2 3 4 5 6<CR>

输出： 1 2 3
 4 5 6

```
void main(void)
```

定义

```
{ int a[2][3], i, j;
```

```
    cout<<"Input 2*3 numbers\n";
```

```
    for (i=0; i<2; i++) /* 输入 */
```

```
        for(j=0; j<3; j++)
```

```
            cin>>a[i][j];
```

赋值

```
    for (i=0; i<2; i++) /* 输出 */
```

```
        { for(j=0; j<3; j++)
```

```
            cout<<a[i][j]<<"\t";
```

```
            cout<<"\n";
```

输出

```
        }
```

```
    }
```

三、二维数组的初始化

在定义数组的同时给数组元素赋值。即在编译阶段给数组所在的内存赋值。

1、分行赋值

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

2、顺序赋值

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}; //依次赋值
```

3、部分赋值

```
int a[3][4]={1,5,9};
```

/ a[0][0]=1, a[1][0]=5, a[2][0]=9 其余元素为0 */*

1	0	0	0
5	0	0	0
9	0	0	0

0	1	0	0
5	0	0	0
0	0	0	0

```
int a[3][4]={0,1,5};
```

/ a[0][0]=0, a[0][1]=1, a[1][0]=5 */*

4、分行或全部赋值时，可以省略第一维，第二维不可省。

```
int a[ ][4]={1,2},{5,6,7,8},{9,10,11,12};
```

5、不能给数组整体赋值，只能一个一个地赋值。

```
int a[2][3]={1,2,3,.....,12};
```

6、用static 定义的数组不赋初值，系统均默认其为‘\0’。

```
static int a[2][3];
```



```
void main(void)
```

```
{  int  a[3][3], i, j;  
    for (i=0; i<3; i++)  
    { for (j=0; j<3; j++)
```

```
        if (i==2)
```

```
            a[i][j]=a[i-1][a[i-1][j]]+1;    i=2
```

```
        else
```

```
            a[i][j]=j;
```

```
        cout<<a[i][j]<<'\t';
```

```
    }
```

```
    cout<<"\n";
```

```
}
```

i=0 a[0][0]=0 a[0][1]=1 a[0][2]=2

i=1 a[1][0]=0 a[1][1]=1 a[1][2]=2

a[2][0]=a[1][a[1][0]]+1=a[1][0]+1=1

a[2][1]=a[1][a[1][1]]+1=a[1][1]+1=2

a[2][2]=a[1][a[1][2]]+1=a[1][2]+1=3

输出: __0__1__2

 __0__1__2

 __1__2__3

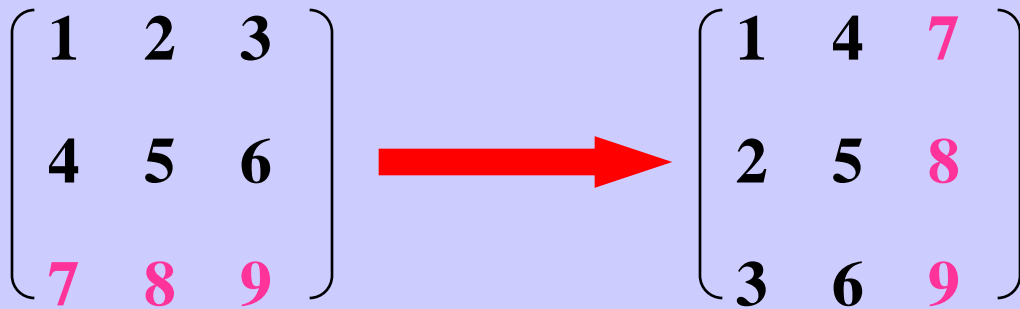
有一个 3×4 的矩阵，要求编程序求出其中值最大的那个元素的值，以及其所在的行号和列号。

先考虑解此问题的思路。从若干个数中求最大者的方法很多，我们现在采用“打擂台”算法。如果有若干人比武，先有一人站在台上，再上去一人与其交手，败者下台，胜者留台上。第三个人再上台与在台上者比，同样是败者下台，胜者留台上。如此比下去直到所有人都上台比过为止。最后留在台上的就是胜者。

程序模拟这个方法，开始时把a[0][0]的值赋给变量max，**max**就是开始时的擂主，然后让下一个元素与它比较，将二者中值大者保存在max中，然后再让下一个元素与新的max比，直到最后一个元素比完为止。max最后的值就是数组所有元素中的最大值。

```
max=a[0][0]; //使max开始时取a[0][0]的值  
for (i=0;i<=2;i++) //从第0行到第2行  
    for (j=0;j<=3;j++) //从第0列到第3列  
        if (a[i][j]>max)//如果某元素大于max  
        {  
            max=a[i][j]; //max将取该元素的值  
            row=i; //记下该元素的行号i  
            colum=j; //记下该元素的列号j  
        }  
cout<<row<<'\t'<<colum<<'\t'<<max<<endl;
```

将数组行列式互换。



```
for (i=0; i<3; i++)
```

```
    for (j=0; j<3; j++)
```

```
        { t=a[i][j];
```

```
            a[i][j]=a[j][i];
```

```
            a[j][i]=t;
```

```
        }
```

```
for (i=0; i<3; i++)
```

```
    for (j=0; j<i; j++)
```

```
        { t=a[i][j];
```

```
            a[i][j]=a[j][i];
```

```
            a[j][i]=t;
```

```
        }
```

打印杨辉三角形

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

$$a[i][j]=a[i-1][j-1]+a[i-1][j]$$

```
void main(void)
```

```
{    static int n[2],i,j,k;
```

```
    for(i=0;i<2;i++)
```

```
        n[j++]=n[i]+i+1;
```

```
    cout<<n[k]<<'\\t'<<n[k++]<<endl;
```

```
}
```

2 1

以下程序用于从键盘上输入若干个学生的成绩，统计出平均成绩，并输出低于平均成绩的学生成绩。输入负数结束

```
void main()
```

```
{ float x[100],sum=0, ave,a;
```

```
int n=0,i;
```

```
cout<<"Input score\n";
```

```
  cin>>a;
```

```
while(a>=0)
```

```
{ x[n]=a;
```

```
  sum+=a;
```

```
  n++
```

```
  cin>>a;
```

```
}
```

```
ave=sum/n;
```

```
cout<<"ave="<<ave<<endl;
```

```
for( i=0; i<n;i++)
```

```
  if(x[i]<ave)
```

```
    cout<<"x["<<i<<"]"<<x[i]<<endl;
```


输入一个十进制数，输出它对应的八进制数。

不断地除8，求其
余数，直到被除
数为0，最后余数
倒序排列。

$$725/2=362$$

$$362/2=181$$

$$181/2=90$$

$$90/2=45$$

$$45/2=22$$

$$22/2=11$$

$$11/2=5$$

$$5/2=2$$

$$2/2=1$$

$$1/2=0$$

$$\text{余数}=1=K_0$$

$$\text{余数}=0=K_1$$

$$\text{余数}=1=K_2$$

$$\text{余数}=0=K_3$$

$$\text{余数}=1=K_4$$

$$\text{余数}=0=K_5$$

$$\text{余数}=1=K_6$$

$$\text{余数}=1=K_7$$

$$\text{余数}=0=K_8$$

$$\text{余数}=1=K_9$$



```
void main(void)
```

```
{   int x , i, n ;
```

```
    int a[100];
```

```
    cin>>x;
```

```
    i=0;
```

```
    while(x)
```

```
    {
```

```
        a[i]=x%8;
```

```
        x=x/8;
```

```
        i++;
```

```
    }
```

```
    n=i;
```

```
    for(i=n-1;i>=0;i--)
```

```
        cout<<a[i];
```

```
    cout<<endl;
```

```
}
```



将余数依次
存入数组中

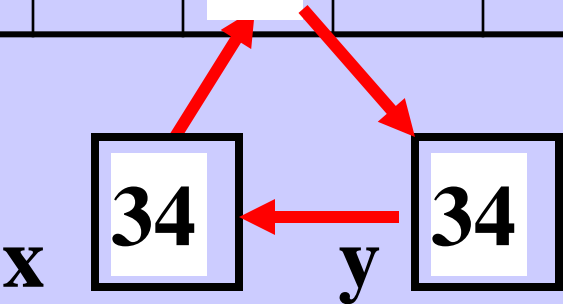
已有一个已排好序的数组, 今输入一个数, 要求按原来排序的规律将它插入数组中。

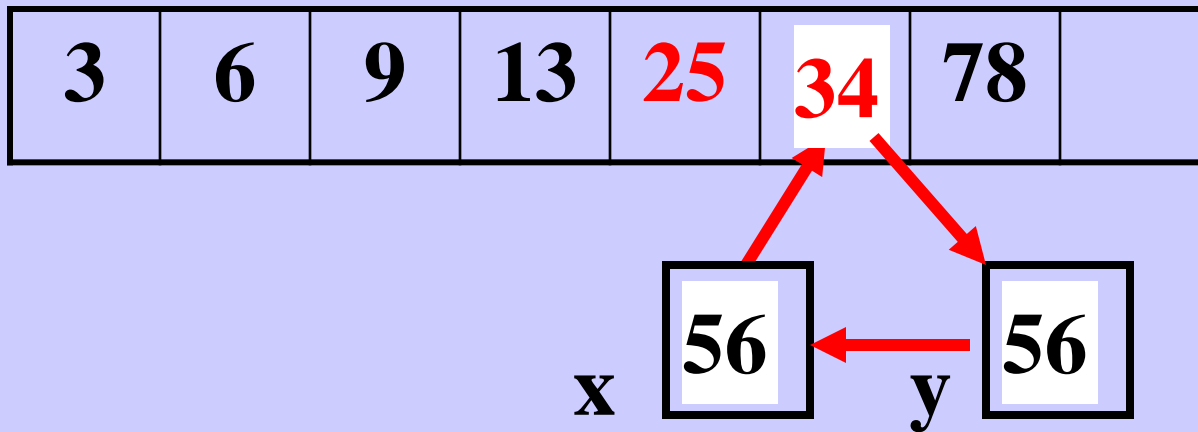
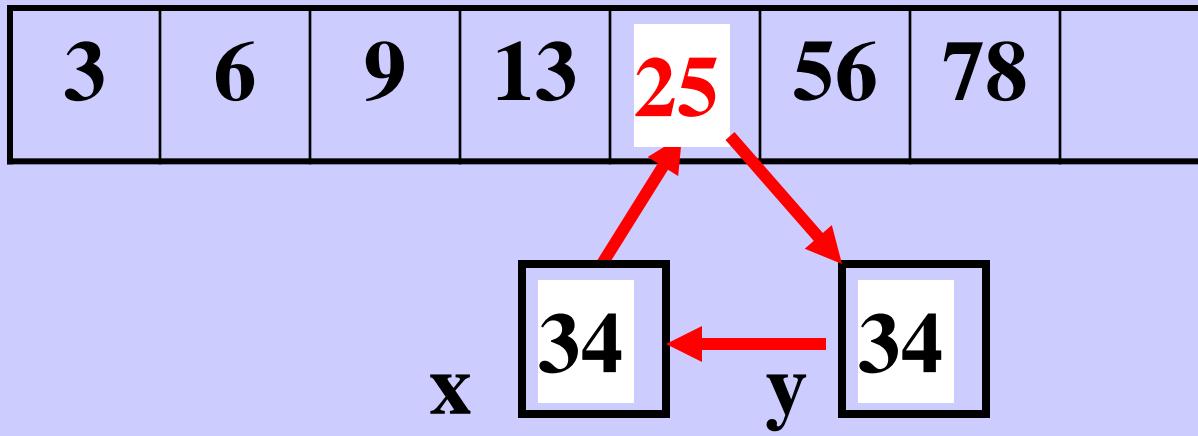
3	6	9	13	34	56	78	
---	---	---	----	----	----	----	--

输入: cin>>x; 25

3	6	9	13	25	34	56	78
---	---	---	----	----	----	----	----

3	6	9	13	25	56	78	
---	---	---	----	----	----	----	--





```

void main(void)
{ int a[6]={1,4,7,10,12};
  int x;
  for(int i=0;i<5;i++)
    cout<<a[i]<<'\\t';
  cout<<endl;
  cout<<"Input x: ";
  cin>>x;
  for(i=0;i<5;i++)
  {   if(a[i]>x)
      break;
  }
}

```

将这个数插入

输入一个数

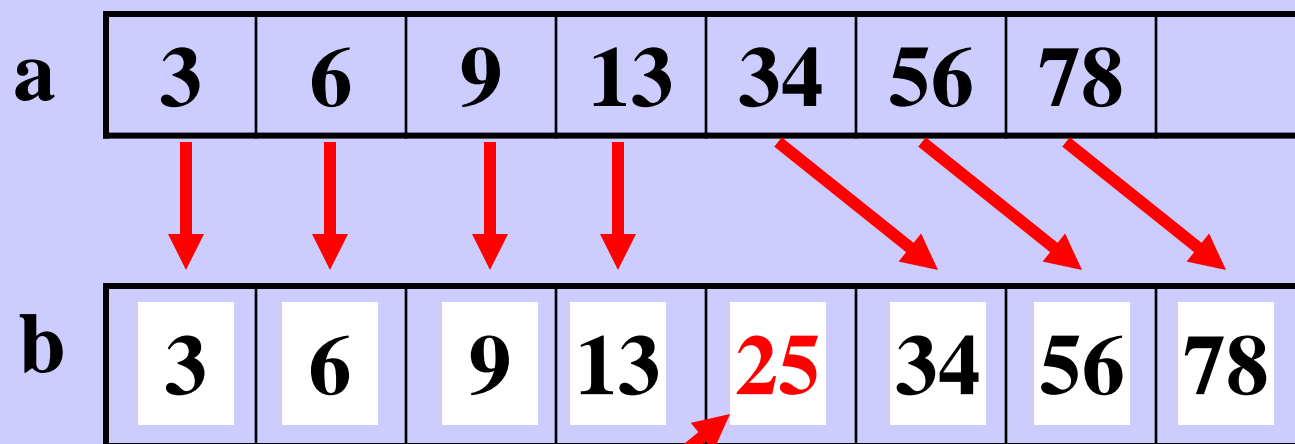
从头比较

大于这个数退出

```

for(int j=i;j<=5;j++)
{   int y=a[j];
    a[j]=x;
    x=y;
}
for( i=0;i<6;i++)
    cout<<a[i]<<'\\t';
cout<<endl;
}

```



输入: `cin>>x;` 25

```
void main(void)
```

```
{ int a[6]={1,4,7,10,12};
```

```
int b[6];
```

```
int x;
```

```
for(int i=0;i<5;i++)
```

```
    cout<<a[i]<<'\\t';
```

```
cout<<endl;
```

```
cout<<"Input x: ";
```

```
cin>>x;
```

```
for(i=0;i<5;i++)
```

```
    if(a[i]<x)
```

```
        b[i]=a[i];
```

```
    else
```

```
        break;
```

重新开始赋值

```
b[i]=x;
```

```
for(int j=i;j<5;j++)
```

```
    b[j+1]=a[j];
```

```
for(i=0;i<6;i++)
```

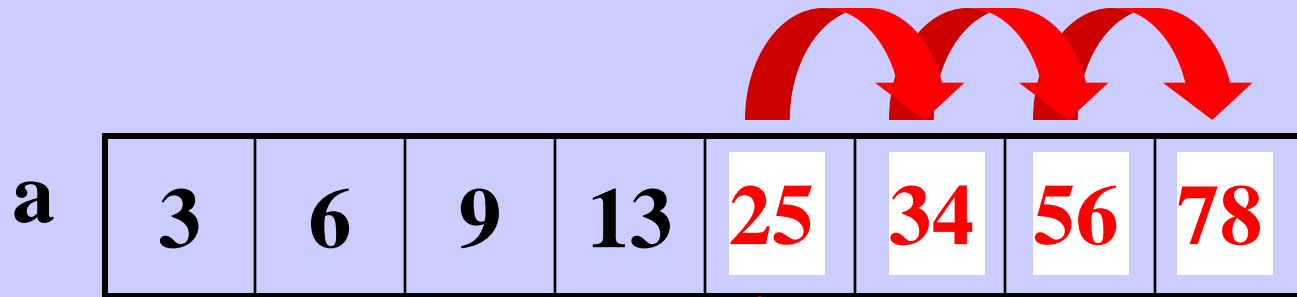
```
    cout<<b[i]<<'\\t';
```

```
cout<<endl;
```

```
}
```

小于这个数赋值

大于这个数退出



输入: `cin>>x;` 25

`for(i=n-1;i>=0;i--)`

从后向前循环


```
void main(void)
{ int a[6]={2,5,8,10,12};
  int x;
  for(int i=0;i<5;i++)
      cout<<a[i]<<'\\t';
  cout<<endl;
  cout<<"Input x: ";
  cin>>x;
  for(i=4;i>0;i--)
  {   if(a[i]>x)
      a[i+1]=a[i];
      else
      break;
  }
```

a[i+1]=x;

赋值

for(i=0;i<6;i++)

cout<<a[i]<<'\\t';

cout<<endl;

}

关键！从后面开始循环

从前向后移数

不大于退出循环

用筛选法求出2~200之间的所有素数。

筛法：首先将1~n个数为数组置初值。2的倍数不是素数，置0； 3的倍数不是素数，置0； 5的倍数不是素数，置0；，依次类推，最后将数组中不是0的元素输出。

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

2	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0
---	---	---	---	---	---	---	---	---	----	---	----	---	----	---	----	---	----	---

2	3	0	5	0	7	0	0	0	11	0	13	0	0	0	17	0	19	0
---	---	---	---	---	---	---	---	---	----	---	----	---	---	---	----	---	----	---

数组作为函数参数

一、数组元素作函数参数

数组元素作函数实参，用法与一般变量作实参相同，是“值传递”。

有两个数据系列分别为：

```
int a[8]={26,1007,956,705,574,371,416,517};
```

```
int b[8]={994,631,772,201,262,763,1000,781};
```

求第三个数据系列 c ， 要求c中的数据是a b中对应数的最大公约数。

```
int a[8]={26, 1007, 956, 705, 574, 371, 416, 517};
```

```
int b[8]={994, 631, 772, 201, 262, 763, 1000, 781};
```

```
    c[8]={2,  1,   4,   3,   2   , 7   , 8,   11}
```

```
int gys(int m,int n)
```

```
{ int r;
```

```
if(m<n) { r=m; m=n; n=r; }
```

```
while(r=m%n) { m=n; n=r; }
```

```
return n;
```

```
}
```

```
void main(void)
```

```
{ int a[8]={26,1007,956,705,574,371,416,517};
```

```
int b[8]={994,631,772,201,262,763,1000,781};
```

```
int c[8];
```

```
for(int i=0;i<8;i++)
```

```
c[i]=gys(a[i],b[i]); //
```

```
for(i=0;i<8;i++)
```

```
cout<<c[i]<<"\t";
```

```
cout<<endl;
```

```
}
```

求m,n的最大公约数,
作为函数值返回

循环求对应数组元素
的最大公约数

二、用数组名作函数参数

在C++中，数组名被认为是**数组在内存中存放的首地址**。

用数组名作函数参数，**实参与形参都应用数组名**。

这时，**函数传递的是数组在内存中的地址**。

实参中的数组地址传到形参中，实参形参**共用同一段内存**。

```
void fun(int a[2])
```

```
{ for(int i=0;i<2;i++)
```

```
    a[i]=a[i]*a[i];
```

```
}
```

```
void main(void)
```

```
{ int b[2]={2,4};
```

```
    cout<<b[0]<<'\t'<<b[1]<<endl;
```

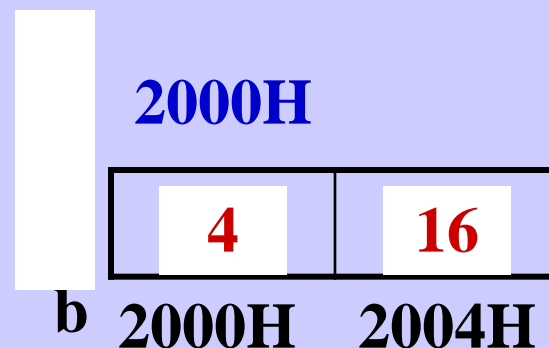
```
    fun(b);
```

```
    cout<<b[0]<<'\t'<<b[1]<<endl;
```

```
}
```

数组b和数组a占据同一段内存

a同样为数组首地址，也是2000H



b就是2000H

输出: 2 4

4 16

```
void sort(int x[ ], int n)
```

```
{ int t,i,j;
```

```
  for( i=0;i<n-1;i++)
```

```
    for(j=0;j<n-i-1;j++)
```

```
      if(x[j]>x[j+1])
```

```
        { t=x[j]; x[j]=x[j+1]; x[j+1]=t;}
```

```
    }
```

```
void main(void)
```

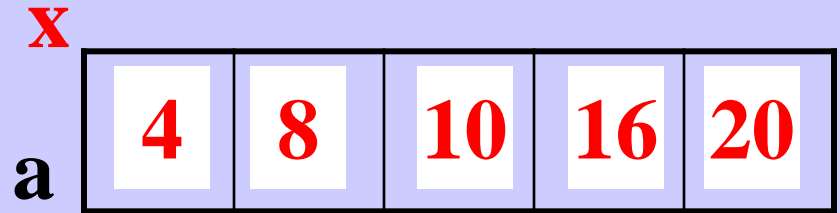
```
{ int a[5]={20,4,16,8,10};
```

```
  sort(a, 5 );
```

```
  for(int i=0;i<5;i++)
```

```
    cout<<a[i]<<"\t";
```

```
}
```



有一个一维数组，内放10个学生成绩，求平均成绩。

数组名作
函数形参

```
float average (float array[ ])  
{ int i;  
  float aver, sum=array[0];  
  for (i=1; i<10; i++)  
    sum=sum+array[i];  
  aver=sum/10;  
  return aver;  
}
```

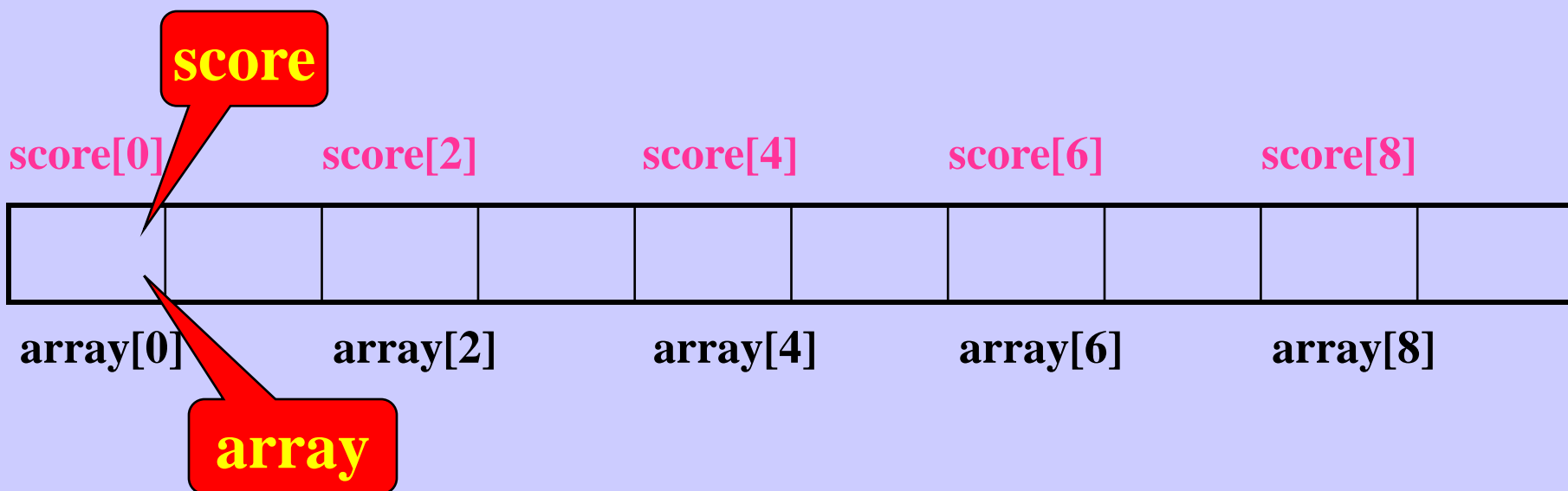
```
void main(void)  
{ static float score[10]={ 100, 90, ...};  
  float aver;  
  aver=average(score);  
  cout<<"aver="<<aver<<"\n";  
}
```

数组名作
函数实参

注意：

- 1、用数组名作函数参数，应在主调函数和被调函数中分别定义数组，且类型一致。
- 2、需指定实参数组大小，形参数组的大小可不指定。数组名作实参实际上是传递数组的首地址。

3、C++语言规定，数组名代表数组在内存中存储的首地址，这样，数组名作函数实参，实际上传递的是数组在内存中的首地址。实参和形参共占一段内存单元，形参数组中的值发生变化，也相当于实参数组中的值发生变化。



程序中的函数p()用于计算:

主函数利用函数完成计算

$$p = \sum_{i=0}^{n-1} ax_i + by_i$$

$$s1 = \sum_{i=0}^7 d_i + 2v_i$$

$$s2 = \sum_{i=0}^9 3v_i + 4w_i$$

```
int d[]={2,3,5,4,9,10,8};
```

```
int v[]={7,6,3,2,5,1,8,9,3,4};
```

```
int w[]={1,2,3,4,5,6,7,8,9,10};
```

```
int p(int a, int x[], int b, int y[], int n)
```

```
{ int i, s;
```

```
  for(i=0,s=0; i<n; i++)
```

```
    s+=a*x[i]+b*y[i];
```

```
  return s;
```

```
}
```

```
void main(void)
```

```
{cout<<"\ns1="<<p(1,d,2,v,8);
```

```
  cout<<"\ns2="<<p(3,v,4,w,10);
```

```
}
```

```
int a[10], i;
```

```
void sub1(void)
```

```
{ for(i=0;i<10;i++) a[i]=i+i; }
```

```
void sub2(void)
```

```
{ int a[10], max,i;
```

```
max=5; for(i=0;i<max;i++) a[i]=i;
```

```
}
```

```
void sub3(int a[])
```

```
{ int i;
```

```
for(i=0;i<10;i++) cout<<a[i]<<'\\t';
```

```
cout<<endl;
```

```
}
```

```
void main(void)
```

```
{ sub1();
```

```
sub3(a);
```

```
sub2();
```

```
sub3(a);
```

```
}
```

输出: 0 2 4 6 8 10 12 14 16 18

0 2 4 6 8 10 12 14 16 18

编写程序，在被调函数中删去一维数组中所有相同的数，使之只剩一个，数组中的数已按由小到大的顺序排列，被调函数返回删除后数组中数据的个数。

例如：

原数组： **2 2 2 3 4 4 5 6 6 6 6 7 7 8 9 9 10 10 10**

删除后： **2 3 4 5 6 7 8 9 10**

用多维数组名作函数参数

同样，实参向形参传递的是数组的首地址。

如果实参、形参是二维数组，则形参可以省略第一维，不可省略第二维，且第二维必须与实参中的维数相等。

```
int score[5][10]
```

```
int array[ ][10]
```

```
int score[5][10]
```

```
int array[3][10]
```

```
int score[5][10]
```

```
int array[ ][8]
```

错误

有一个 3×4 的矩阵，求其中的最大元素。

```
int max_value (int array[ ][4])
```

```
{ int i, j, k, max;
```

形参

```
max=array[0][0];
```

```
for (i=0; i<3; i++)
```

```
for (j=0; j<4; j++)
```

```
if (array[i][j]>max)
```

```
max=array[i][j];
```

```
return (max);
```

函数值

```
void main (void)
```

```
{ static int a[3][4]={ {1,3,5,7},
```

```
{2,4,6,8},{15,17,34,12}};
```

```
cout<<"max is "<<max_value(a)<<"\t";
```

```
}
```

实参

数组a与数组array共用一段内存

字符数组

用来存放字符数据的数组是字符数组，字符数组中的一个元素存放一个字符。

一、字符数组的定义

char 数组名[常量表达式];

类型

数组名

char **c[4];** /*每个元素占一个字节 */

数组大小

c[0]='I'; c[1]='m'; c[2]='_';

二、字符数组的初始化

与数值数组的初始化相同，取其相应字符的**ASCII**值。

```
char c[10]={‘I’, ‘ ’, ‘a’, ‘m’, ‘ ’, ‘a’, ‘ ’, ‘b’, ‘o’, ‘y’};
```

	c[0]								c[9]		
c	‘I’	‘ ’	‘a’	‘m’	‘ ’	‘a’	‘ ’	‘b’	‘o’	‘y’	随机

```
char st1[ ]={65, 66, 68};
```

'A'	'B'	'D'
-----	-----	-----

如果字符个数大于数组长度，做错误处理；如果数值个数小于数组长度，后面的字节全部为 ‘\0’。

如果省略数组长度，则字符数即为数组长度。

```
static char c[ ]={'I', ' ', 'a', 'm', ' ', 'a', ' ', 'g', 'i', 'r', 'l'};
```

同理，也可定义和初始化一个二维或多维的字符数组。分层或省略最后一维。

三、字符数组的引用

```
void main(void)
```

```
{ char c[10]={‘I’, ‘ ’, ‘a’, ‘m’, ‘ ’, ‘a’, ‘ ’, ‘b’, ‘o’, ‘y’};
```

```
    int i;
```

定义

```
    for (i=0; i<10; i++)
```

```
        cout<<c[i];
```

```
    cout<<“\n”;
```

输出

```
}
```

四、字符串和字符串结束标志

C++语言将字符串作为字符数组来处理。

字符串常量：“CHINA”，在机内被处理成一个无名的字符型一维数组。

C	H	I	N	A	'\0'
---	---	---	---	---	------

C++语言中约定用 ‘\0’作为字符串的结束标志，它占内存空间，但不计入串长度。有了结束标志

‘\0’后，程序往往**依据它判断字符串是否结束**，而不是根据定义时设定的长度。

字符串与字符数组的区别：

```
char a[ ]={'C','H','I','N','A'};
```

字符数组

C	H	I	N	A	随机	随机
---	---	---	---	---	----	----

长度占5个字节

```
char c[ ]="CHINA";
```

字符串

C	H	I	N	A	'\0'	随机
---	---	---	---	---	------	----

长度占6个字节

可以用字符串的形式为字符数组赋初值

```
char c[ ]={"I am a boy"}; /*长度11字节, 以 '\0'结尾  
*/  
char a[ ]={'I', ' ', 'a', 'm', ' ', 'a', ' ', 'b', 'o', 'y'};  
/* 长度10字节 */
```

如果数组定义的长度大于字符串的长度, 后面均为 '\0'。

```
char c[10]="CHINA";
```

c	C	H	I	N	A	\0	\0	\0	\0	\0
---	---	---	---	---	---	----	----	----	----	----

'\0'的ASCII为0, 而 ' ' (空格)的ASCII为32。

```
char w[ ]={'T', 'u', 'r', 'b', 'o', '\0'};
```

```
char w[ ]={"Turbo\0"};
```

```
char w[ ]="Turbo\0";
```

```
char w[ ]='Turbo\0';
```

非法

T	u	r	b	o	'\0'
T	u	r	b	o	'\0'
T	u	r	b	o	'\0'


```
char a[2][5]={“abcd”, “ABCD”};
```

a	b	c	d	‘\0’
A	B	C	D	‘\0’

在语句中字符数组不能用赋值语句整体赋值。

```
char str[12];          char str[12]=“The String”;
```

```
str=“The String”;
```

非法，在语句中赋值

定义数组，开辟空间时赋初值

str为字符数组在内存中存储的地址，一经定义，便成为常量，不可再赋值。

字符数组的输入输出

逐个字符的输入输出。这种输入输出的方法，通常是使用循环语句来实现的。如：

```
char str[10];
```

定义

```
cout<<“输入十个字符： ”；
```

```
for(int i=0;i<10;i++)
```

```
cin>>str[i];           //A
```

赋值

.....

A行将输入的十个字符依次送给数组str中的各个元素。

把字符数组作为字符串输入输出。对于一维字符数组的输入，在cin中仅给出**数组名**；输出时，在cout中也只给出**数组名**。

```
void main (void )
```

输入： abcd<CR>

```
{char s1[50],s2[60];
```

string<CR>

```
cout << “输入二个字符串:”;
```

```
cin >> s1;
```

数组名

```
cin >> s2;
```

数组名

```
cout << “\n s1 = “ << s1;
```

```
cout << “\n s2 = “ << s2 << “\n”;
```

```
}
```

输出到 ‘\0’为止

cin只能输入一个单词，不能输入一行单词。

当要把**输入的一行**作为一个字符串送到字符数组中时，则要使用函数**cin.getline()**。这个函数的第一个参数为字符数组名，第二个参数为允许输入的最大字符个数。

cin.getline(数组名, 数组空间数);

首先开辟空间

char s1[80];

.....

参数是数组名

cin.getline(s1, 80);

```
void main (void )
```

```
{  char s3[81];
```

定义

```
    cout<<"输入一行字符串:";
```

```
    cin.getline(s3,80);
```

从键盘接收一行字符

```
    cout<<"s3="<<s3<<"\n";           //B
```

```
}
```

输出到 '\0' 为止

当输入行中的字符个数小于80时，将实际输入的字符串（不包括换行符）全部送给s3；当输入行中的字符个数大于80时，只取前面的80个字符送给字符串。

从键盘输入一行字符，统计其中分别有多少大小写字母，以\$号结束输入。

从键盘输入一行字符，统计其中分别有多少大小写字母。

从键盘输入一行字符，其中的大写变小写，小写变大写。

从键盘接收一行字符，统计有多少个单词数？

we are students.

	w	e			a	r	e		s
	字母	字母	空格	空格	字母	字母	字母	空格	字母
0	1	1	0	0	1	1	1	0	1

不能用字母数或空格数来判断，只能用字母和空格**状态变化**的次数来判断。

设状态变量word，判别到字母时word为1，判别到非字母时word为0。

word的初始值为0，当从0变为1时，单词数加1。

```
void main(void)
{char s[80];
int i=0, word=0,num=0;
cin.getline (s,80);
while(s[i]!='\0')
```

表明前一字符非字母

```
{ if((s[i]>='a'&& s[i]<='z' || s[i]>='A'&& s[i]<='Z')&& word==0)
```

```
{    word=1;
    num++;
}
```

改变状态，防止继续对
下一字母计数

```
else if(s[i]==' ' || s[i]=='\t')
```

```
    word=0;
```

改变状态，碰到下一个
字母时开始计数

```
    i++;
```

```
}
```

```
cout<<"num="<<num<<endl;
```

```
}
```


六、字符串处理函数

C++中没有对字符串变量进行赋值、合并、比较的运算符，但提供了许多字符串处理函数，用户可以调用 `#include "string.h"`

所有字符串处理函数的实参都是字符串数组名

1、合并两个字符串的函数 `strcat (str1, str2)`

空间足够大

```
static char str1[20]={“I am a ”};
```

```
static char str2[ ]={“boy”};
```

```
strcat (str1, str2);
```

I		a	m		a		'\0'	'\0'					
---	--	---	---	--	---	--	------	------	--	--	--	--	--

b	o	y	'\0'
---	---	---	------

I		a	m		a		b	o	y	'\0'			
---	--	---	---	--	---	--	---	---	---	------	--	--	--

将第二个字符串 `str2` 接到第一个字符串 `str1` 后。

注意：第一个字符串要有足够的空间。

2、复制两个字符串的函数 strcpy (str1, str2)

```
static char str1[20]={“I am a ”};
```

```
static char str2[ ]={“boy”};
```

```
strcpy (str1, str2);
```

str1

I		a	m		a		\0	\0					
---	--	---	---	--	---	--	----	----	--	--	--	--	--

str2

b	o	y	\0
---	---	---	----

str1

b	o	y	\0		a		\0	\0					
---	---	---	----	--	---	--	----	----	--	--	--	--	--

```
strcpy ( str1, “CHINA”);
```

字符串正确赋值

str1

C	H	I	N	A	\0								
---	---	---	---	---	----	--	--	--	--	--	--	--	--

```
str1=str2; str1=“CHINA”;
```

均为非法

```
strcpy (“CHINA”, str1);
```

3、比较两个字符串的函数 `strcmp (str1, str2)`

此函数用来比较str1和str2中字符串的内容。函数对字符串中的ASCII字符**逐个两两比较**，直到遇到不同字符或 ‘\0’ 为止。函数值由两个对应字符相减而得。

该函数具有返回值，返回值是两字符串对应的**第一个不同**的ASCII码的差值。

若两个字符串完全相同，函数值为0。

```
if ( strcmp (str1, str2)==0)
```

```
{ ..... }
```

用来判断两字符串是否相等

```
static char str1[20]={“CHINA”};
```

```
static char str2[ ]={“CHINB”};
```

```
cout<< strcmp (str1, str2)<<endl;
```

输出: -1

```
static char str1[20]={“CHINA”};
```

```
static char str2[ ]={“AHINB”};
```

```
cout<<strcmp (str1, str2)<<endl;
```

输出: 2

```
if (str1==str2) cout<<“yes\n”;
```

非法

正确

```
if (strcmp (str1,str2)= =0)    cout<<“yes\n”;
```

4、求字符串长度的函数 `strlen(str1)`

函数参数为数组名，返回值为数组首字母到 ‘\0’ 的长度。

并非数组在内存中空间的大小。长度不包括 ‘\0’。

```
char s[80];
```

```
strcpy(s, "abcd");
```

```
cout<<strlen(s)<<endl;    输出： 4
```

```
cout<<sizeof(s)<<endl;    输出： 80
```

str1	b	o	y	\0		a		\0	\0					
------	---	---	---	----	--	---	--	----	----	--	--	--	--	--

```
cout<<strlen(str1)<<endl;    输出： 3
```

```
char str1[20]={“CHINA”};
```

```
cout<<strlen (str1)<<endl;
```

输出： 5

```
char str1[20]={“a book”};
```

```
cout<<strlen (str1)<<endl;
```

输出： 6

```
char sp[ ]={“\t\v\\0will\n”};
```

```
cout<<strlen (sp)<<endl;
```

输出： 3

```
char sp[ ]={“\x69\082”};
```

```
cout<<strlen (sp)<<endl;
```

输出： 1

5、 **strlwr (str1)**

将str1中的大写字母转换成小写字母。

6、 **strupr (str1)**

将str1中的小写字母转换成大写字母。

7、函数strncmp(字符串1,字符串2 , maxlen)

函数原型为:

int strncmp(char str1[], char str2[],int m)

第三个参数为正整数，它限定了至多比较的字符个数

若字符串1或字符串2的长度小于maxlen的值时，函数的功能与strcmp()相同。

当二个字符串的长度均大于maxlen的值时，maxlen为至多要比较的字符个数。

输出：0

```
cout<<strncmp(“China”,“Chifjsl;kf”,3)<<“\n”;
```

8、函数strncpy(字符数组名1, 字符串2, maxlen)

函数原型为:

```
void strncpy(char str1[ ], char str2[ ],int m)
```

第三个参数为正整数，它限制了至多拷贝的字符个数

若字符串2的长度小于maxlen的值时，函数的功能与strcpy()相同。

当字符串2的长度大于maxlen的值时，maxlen为至多要拷贝的字符个数。

空间足够大

```
char s[90],s1[90];
```

```
strncpy(s,"abcdssfsdfk",3);           //A
```

```
strncpy(s1,"abcdef ", 90);           //B
```

```
cout<<s<<endl;      输出: abc
```

```
cout<<s1<<endl;     输出: abcdef
```

注意，二字符串之间不能直接进行比较，赋值等操作，这些操作必须通过字符串函数来实现。

输入三个字符串按大小输出。

输入n个字符串按大小输出。

```
void changed(char str1[],char str2[])  
{char str3[80];  
  strcpy(str3,str1);  
  strcpy(str1,str2);  
  strcpy(str2,str3);  
}
```

```
void main(void)  
{char s1[80],s2[80],s3[80];  
  cout<<"Input 3 strings:\n";  
  cin.getline(s1);  
  cin.getline(s2);  
  cin.getline(s3);  
  if(strcmp(s1,s2)>0) changed(s1,s2);  
  if(strcmp(s1,s3)>0) changed(s1,s3);  
  if(strcmp(s2,s3)>0)changed(s2,s3);  
  cout<<"sorted:"<<endl<<endl;  
  cout<<s1<<endl<<s2<<endl<<s3<<endl;  
}
```

```
void changed(char str1[],char str2[])
{char str3[80];
 strcpy(str3,str1);
 strcpy(str1,str2);
 strcpy(str2,str3);
}
```

```
void main(void)
{char ss[10][80];
 int i,j;
 cout<<"Input 10 strings:\n";
 for(i=0;i<10;i++)
     cin.getline (ss[i],80);
 for(i=0;i<9;i++)
     for(j=0;j<9-i;j++)
         if(strcmp(ss[j],ss[j+1])>0)
             changed(ss[j],ss[j+1]);
 cout<<"sorted:"<<endl<<endl;
 for(i=0;i<10;i++)
     cout<<ss[i]<<endl;
}
```

用选择法对6个数排序（由小到大）

设定一个变量，放入数组中的最小数的**序号**，然后将其与最上面的数比较交换。

1、min=1 2、a[min]与a[2]比较 3、min=2 4、a[min]与a[3]比较

min	9 a[1]
<div>1</div>	8 a[2]
	5 a[3]
	4 a[4]
	2 a[5]
	0 a[6]

假定元素**序号**
为1的数是最
小的数

即**9**与**8**比较

min	9 a[1]
<div>2</div>	8 a[2]
	5 a[3]
	4 a[4]
	2 a[5]
	0 a[6]

这时，最小数
的序号变为2

即**8**与**5**比较

min=3

a[min]与a[4]比较

min=4

a[min]与a[5]比较

min

3

这时，最小数的序号变为3

9 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

0 a[6]

即5与4比较

min

4

这时，最小数的序号变为4

9 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

0 a[6]

即4与2比较

min=5

a[min]与a[6]比较

min=6

a[min]与a[1]交换

min

5

这时，最小数的序号变为5

9 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

0 a[6]

第一趟比较完毕，
最小数是a[6]，最小数的序号为6

第一趟，循环5次

min

6

0 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

9 a[6]

min=2

a[min]与a[3]比较

min=3

a[min]与a[4]比较

min

2

0 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

9 a[6]

min

3

0 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

9 a[6]

从第二个数开始
比较，假定最小
数的序号为2

min=4

a[min]与a[5]比较

min=5

a[min]与a[6]比较

min

4

0 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

9 a[6]

min

5

0 a[1]

8 a[2]

5 a[3]

4 a[4]

2 a[5]

9 a[6]

min=5

a[min]与a[2]交换

min

5

0 a[1]

2 a[2]

5 a[3]

4 a[4]

8 a[5]

9 a[6]

第二趟比较完毕，
最小数是**a[5]**，最小
数的序号为**5**

第二趟，循环4次

min=3

a[min]与a[4]比较

min=4

a[min]与a[5]比较

min

3

0 a[1]

2 a[2]

5 a[3]

4 a[4]

8 a[5]

9 a[6]

min

4

0 a[1]

2 a[2]

5 a[3]

4 a[4]

8 a[5]

9 a[6]

min=4

a[min]与a[6]比较

min=4

a[min]与a[3]交换

min

4

0 a[1]

2 a[2]

5 a[3]

4 a[4]

8 a[5]

9 a[6]

min

4

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

第三趟，循环3次

min=4

a[min]与a[5]比较

min=4

a[min]与a[6]比较

min

4

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

min

4

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

min=4

a[min]与a[4]交换

min

4

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

第四趟，循环2次

min=5

a[min]与a[6]比较

min=5

a[min]与a[5]交换

min

5

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

min

5

0 a[1]

2 a[2]

4 a[3]

5 a[4]

8 a[5]

9 a[6]

第五趟，循环1次

for (i=1; i<=n-1; i++)

{ min=i ;

for (j=i; j<=n; j++)

if (a[min]>a[j]) min=j ;

t=a[min];

a[min]=a[i];

a[i]=t;

}

总结:

	共有6个数					n
趟数	1	2	3	4	5	i(1~n-1)
次数	5	4	3	2	1	n-i

一般，元素的序号从0开始，因此，程序可以变动如下：

```
for (i=0; i<n-1; i++)
```

```
{ min=i;
```

每一次循环前设置最小数的序号

```
for (j=i; j<n ; j++)
```

```
if (a[min]>a[j])
```

```
min=j ;
```

小循环，找最小数的序号，从 i 找起

```
t=a[min];
```

```
a[min]=a[i];
```

```
a[i]=t;
```

大循环，找到后与 i 交换

```
}
```

调试程序的方法：

- 1) 单步调试：以行为单位，每运行一步，程序就会中断，可以实时查询目前各变量的状态及程序的走向。可以选择是否进入子函数。
- 2) 运行到光标处，可以直接使程序运行到光标处再进行单步调试，这种方法可以不必运行正确的循环而直接到有疑问的地方。

在a数组中查找与x值相同的元素所在的位置，数据从a[1]元素开始存放，请填空：

```
#define MAX 10
```

```
while(x!=    a[i]    )
```

```
void main(void)
```

```
    i--    ;
```

```
{ int a[MAX+1], x, i;
```

```
if(    i!=0    )
```

```
for(i=1;i<=MAX;i++)
```

```
cout<<x<<"the pos:"<<i<<endl
```

```
cin>>    a[i]    ; else
```

```
cout<<"Enter x:";
```

```
cout<<"Not found"<<endl;
```

```
cin>>x;
```

```
a[0]=x; i=MAX;
```



```

void main(void)
{ char str[ ]="SSSWILTECH1\1\11W\1WALLMP1";
  char c; int k;
  for(k=2; (c=str[k])!='\0';k++)
  { switch(c)
    { case 'A' : cout<<'a'; continue;
      case '1': break;
      case 1:  while((c=str[++k])!='\1'&&c!='\0');
      case 9: cout<<'#';
      case 'E' :
                                S W I T C H * # W a M P *
      case 'L': continue;
      default: cout<<c; continue;
    } cout<<'*';
  } cout<<endl;
}

```

以下程序分别在a数组和b数组中放入an+1和bn+1个由小到大的有序数，程序把两个数组中的数按由小到大的顺序归并到c数组中，请填空：

```
void main(void)
{   int a[10]={1,2,5,8,9,10},an=5;

    int b[10]={1,3,4,8,12,18}, bn=5;

    int i,j,k, c[20], max=9999;

    a[an+1]=b[an+1]=max;

    i=j=k=0;
```

```
    while( a[i]!=max||b[j]!=max)
    {   if(a[i]<b[j])
        {   c[k]=a[i];
            k++;
            i++
        }
        else
        {   c[k]=b[j];
            k++;
            j++;
        }
        for(i=0;i<k;i++)cout<<c[i];
        cout<<endl;
    }
```

编写程序，在被调函数中删去一维数组中所有相同的数，使之只剩一个，数组中的数已按由小到大的顺序排列，被调函数返回删除后数组中数据的个数。

例如：

原数组： **2 2 2 3 4 4 5 6 6 6 6 7 7 8 9 9 10 10 10**

删除后： **2 3 4 5 6 7 8 9 10**

第七章 结构体、共同体和枚举类型

定义：

将不同种类型的数据有序地组合在一起，构造出一个新的数据类型，这种形式称为结构体。

结构体是多种类型组合的数据类型。

struct 结构体名

{ 成员列表 } ;

关键字

结构体名

struct student

{ int num;

char name[20];

char sex;

char addr[30];

不同数据类型组成的
成员

};

分号不能少

定义结构体类型变量的方法

一、先定义结构体类型再定义变量名

```
struct student
{
    int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
};
```

```
struct student student1, student2;
```

结构体类型名

变量1

变量2

结构体类型只是一种数据类型，不占内存空间，只有定义结构体类型变量时才开辟内存空间。

```
# define STUDENT struct student
```

```
STUDENT
```

```
{ int num;  
  char name[20];  
  char sex;  
  int age;  
  float score;  
  char addr[30];  
};
```

```
STUDENT student1,student2;
```

凡是**STUDENT**的地方都用**struct student**机械替换。

二、在定义类型的同时定义变量

```
struct student
```

```
{ int num;
```

```
  char name[20];
```

```
  char sex;
```

```
  int age;
```

```
  float score;
```

```
  char addr[30];
```

```
} student1, student2;
```

struct 结构体名

{

成员列表

} 变量名列表;

紧接着定义变量

三、直接定义结构体类型变量

struct

```
{  int  num;  
    char name[20];  
    char sex;  
    int  age;  
    float score;  
    char addr[30];  
} student1, student2;
```

struct

{

成员列表

} 变量名列表;

不出现结构体名。

1、结构体类型的变量在内存**依照其成员的顺序**顺序排列，所占内存空间的大小是其全体成员所占空间的**总和**。

2、在编译时，仅对**变量**分配空间，不对**类型**分配空间。

3、对结构体中各个成员可以单独引用、赋值，其作用与变量等同。

格式：**变量名.成员名** student1 . num

4、结构体的成员可以是另一个结构体类型。

```
struct date
```

```
{ int month;
```

```
    int day;
```

```
    int year;
```

```
};
```

```
struct student
```

```
{ int num;
```

```
    char name[20];
```

```
    struct date
```

```
};
```

```
    birthday;
```

成员类型

成员名

5、成员名可以与程序中的变量名相同，二者分占不同的内存单元，互不干扰。例如，在程序中仍可以定义变量

```
int num;
```

结构体类型变量的引用

1、不能对结构体变量整体赋值或输出，只能分别对各个成员引用。

错误

```
cin>>student1;
```

必须用成员名引用

```
cin>>student1.num; student1.num=100;
```

可以将一个结构体变量整体赋给另外一个相同类型的结构体变量。

```
student2=student1;
```

2、嵌套的结构体变量必须逐层引用。

```
student1.birth.day=25;
```

3、结构体变量中的成员可以同一般变量一样进行运算。

```
student1.birth.day++; student1.score+=60;
```

对局部变量类型的结构体变量初始化

```
void main(void)
```

```
{ struct student
```

```
{ long int num;
```

```
char name[20];
```

```
char sex;
```

```
char addr[30];
```

对变量初始化，一一赋值

```
} student1={901031, "Li Lin", 'M', "123 Beijing Road"};
```

```
cout<<student1.name<<endl;
```

输出: LiLin

关于结构类型变量的使用，说明以下几点：

- 1、同类型的结构体变量之间可以直接赋值。这种赋值等同于各个成员的依次赋值。
- 2、结构体变量不能直接进行输入输出，它的每一个成员能否直接进行输入输出，取决于其成员的类型，若是基本类型或是字符数组，则可以直接输入输出。
- 3、结构体变量可以作为函数的参数，函数也可以返回结构体的值。当函数的形参与实参为结构体类型的变量时，这种结合方式属于值调用方式，即属于值传递。（举例说明）

结构体数组

结构体数组中的每个元素都是一个结构体类型的变量，其中包括该类型的各个成员。数组各元素在内存中连续存放。

一、结构体数组的定义

```
struct student
```

```
{  int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
    char addr[30];  
};
```

```
struct student stu[30];
```

```
struct student
```

```
{  int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
    char addr[30];
```

```
} stu[30];
```

直接定义

二、结构体数组的初始化

```
struct student
```

```
{ int num;
```

```
    char name[20];
```

```
    char sex;
```

```
} stu[3]={ {1011, "Li Lin",'M'}, {1012,"Wang Lan",'F'},  
           {1013,"Liu Fang",'F'};
```

```
struct student
```

```
{ int num;
```

```
    char name[20];
```

```
    char sex;
```

```
} stu[ ]={ {1011,"Li Lin",'M'}, {1012,"Wang Lan",'F'},  
           {1013,"Liu Fang",'F'}};
```

以下程序的结果是：

```
void main(void)  
  
{ struct date  
  
    { int year, month, day;  
  
    } today;  
  
    cout<<sizeof(struct date)<<endl;  
  
}
```

12

根据下面的定义，能打印出字母M的语句是：

```
struct person { char name[9];  
                int age;  
};
```

```
struct person class[10]={ “Jone”,17, “Paul”,19,  
                          “Mary”,18, “Adam”,16  
                          };
```

A) cout<<class[3].name<<endl; 输出： Adam

B) cout<<class[3].name[1]<<endl; 输出： d

C) cout<<class[2].name[1]<<endl; 输出： a

D) cout<<class[2].name[0]<<endl; 输出： M

结构体类型的静态成员

当把结构体类型中的某一个成员的存储类型定义为静态时，表示在这种结构类型的所有变量中，编译程序为这个成员只分配一个存储空间，即这种结构体类型的所有变量共同使用这个成员的存储空间。

<类型> <结构体类型名>:: <静态成员名>;

其中类型要与在结构体中定义该成员的类型一致，结构体类型名指明静态成员属于哪一个结构体。

```
struct s{  
    static int id;
```

```
int eng;
```

数据类型

结构体类型

```
int s::id=50;
```

这时，未定义结构体变量，
但已将静态成员的空间安排好。

若有定义：s s1,s2;

则变量s1,s2的id成员占用同一
存储空间（静态区）。

在结构体中说明的静态成员属于引用性说明，必须在文件作用域中的某一个地方对静态的成员进行定义性说明，且仅能说明一次。

```
int s::id;
```

说明id的初值为0（静态变量的缺省初值均为0）

共用体

C++语言中，允许不同的数据类型使用同一存储区域，即同一存储区域由不同类型的变量共同表示。这种数据类型就是共用体。

union 共用体名

{ 成员表列;

} 变量表列;

union data a, b, c;

union data

{ int i;

char ch;

float f;

} a, b, c;

这几个成员在共用体变量中存放在同一地址，相互覆盖，其长度为最长的成员的长度。

共用体变量的引用

不能整体引用共用体变量，只能引用变量中的成员。

a.i 表示为整型

a.ch 表示为字符型

a.f 表示为浮点型

共用体变量的特点

- 1、共用体的空间在某一时刻只有一个成员在起作用。
- 2、共用体变量中的成员是最后一次放入的成员。
- 3、共用体变量不能在定义时赋初值。
- 4、共用体变量不能作为函数的参数或函数值，但可使用指向共用体的指针变量。
- 5、共用体可以作为结构的成员，结构体也可以作为共用体的成员。

```
union un
```

```
{ int i;
```

```
    double y;
```

```
};
```

```
struct st
```

```
{ char a[10];
```

```
    union un b;
```

```
};
```

```
cout<<sizeof(struct st)<<endl;
```

18

```
union un
{ short int a;
  char c[2];
} w;
```

低字节低地址

2000H

6

高字节高地址

2001H

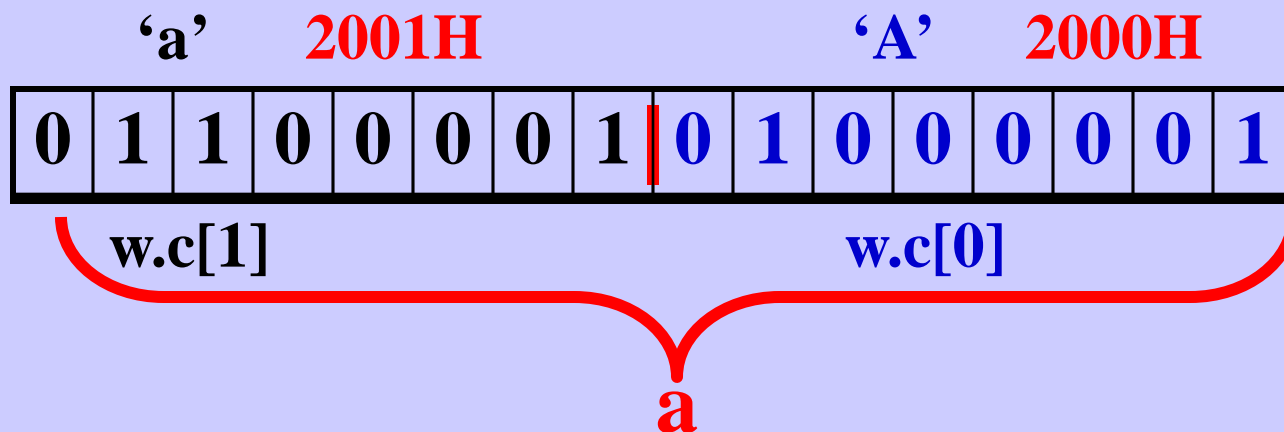
5

65 ?

```
w.c[0]='A'; w.c[1]='a';
```

输出: 060501 56 ?

```
cout<<oct<<w.a<<endl;
```



```
void main(void)
```

```
{ union EXAMPLE
```

```
{ struct { int x, int y;} in;
```

```
int a,b;
```

```
}e;
```

```
e.a=1;
```

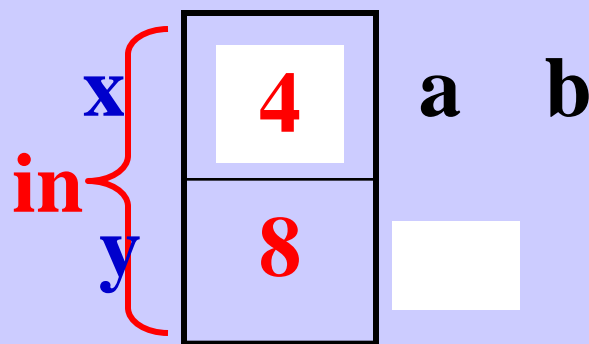
```
e.b=2;
```

```
e.in.x=e.a*e.a;
```

```
e.in.y=e.b+e.b;
```

```
cout<<e.in.x<<'\t'<<e.in.y<<endl;
```

```
}
```



输出： 4 8

枚举类型

如果一个变量只有**几种**可能的值，可以定义为枚举类型。

枚举类型就是将变量的值一一列举出来，变量的值仅限于列举出来的值的范围内。

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
```

数据类型

可能取的值

变量

```
enum weekday weekday, weekend ;
```

weekday 和 **weekend** 值只能是sun 到 sat 其中之一。

另一种定义变量的方法

```
enum {sun, mon, tue, wed, thu, fri, sat} weekday,  
weekend ;
```

其中sun, mon, ..., sat称为枚举元素或枚举常量，为用户定义的标识符，所代表的意义由用户决定，在程序中体现出来。

1、枚举元素为常量，不可赋值运算。 sun=0; mon=1;

2、在定义枚举类型的同时，编译程序按顺序给每个枚举元素一个对应的序号，序号从0开始，后续元素依次加1。

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
```

0 , 1, 2, 3, 4, 5, 6

3、可以在定义时人为指定枚举元素的序号值。

```
enum weekday {sun=9, mon=2, tue, wed, thu, fri, sat};
```

9 , 2, 3, 4, 5, 6, 7

4、只能给枚举变量赋枚举值，若赋序号值必须进行强制类型转换。

```
day=mon;    day=1;    day=(enum weekday)1;
```


5、枚举元素可以用来进行比较判断。

```
if (workday== mon)
```

```
if (workday>sun)
```

6、枚举值可以进行加减一个整数n的运算，得到其前后第n个元素的值。

```
workday=sun;
```

```
workday== tue
```

```
workday=(week)(workday+2);
```

7、枚举值可以按整型输出其序号值。

```
workday=tue;
```

```
cout<<workday;
```

2

```
void main(void)  
{  
  
  enum team{ qiaut, cubs=4, pick, dodger=qiaut-2;};  
  
  cout<<qiaut<<'\\t'<<cubs<<'\\t';  
  
  cout<<pick<<'\\t'<<dodger<<endl;  
  
}
```

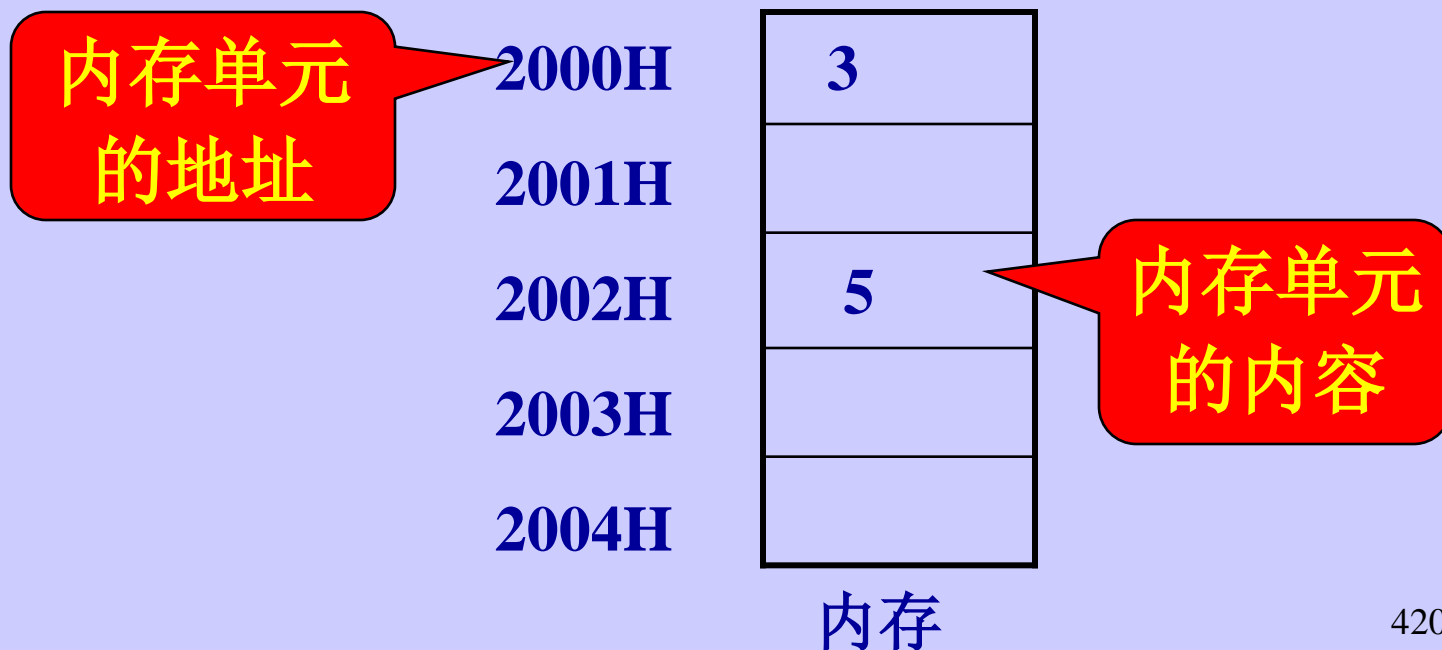
输出： 0 4 5 -2

第八章 指针和引用

指针的概念

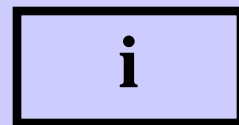
数据在内存中是如何存取的？

系统根据程序中定义变量的类型，给变量分配一定的长度空间。字符型占1个字节，整型数占4个字节.....。内存区的每个字节都有编号，称之为**地址**。



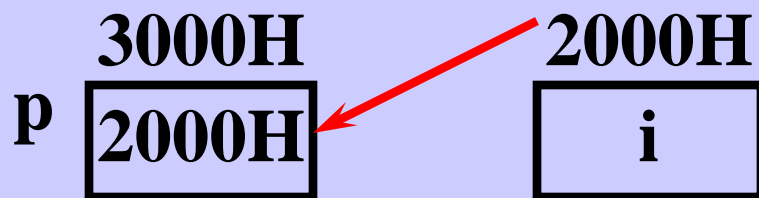
1、直接访问

按变量地址存取变量的值。**cin>>i**；实际上放到定义 **i** 单元的**地址**中。



2、间接访问

将变量的地址存放在另一个单元**p**中，通过 **p** 取出变量的地址，再针对变量操作。



一个变量的地址称为该变量的指针。

如果在程序中定义了一个变量或数组，那么，这个变量或数组的地址（指针）也就确定为一个**常量**。



变量的**指针**和指向变量的**指针变量**

变量的指针就是**变量的地址**，当变量定义后，其指针（地址）是一常量。

`int i;` **&i : 2000H** 

可以**定义一个变量**专门用来**存放**另一变量的**地址**，这种变量我们称之为**指针变量**。在编译时同样分配一定字节的存储单元，未赋初值时，该存储单元内的值是随机的。

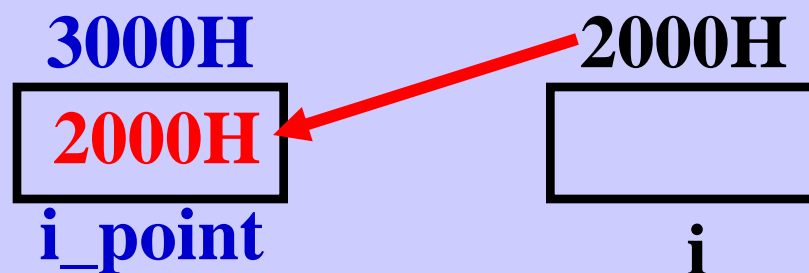
指针变量定义的一般形式为：

类型标识符 *变量名  **指针类型**  **变量名**
`int *i_point;`

指针**变量**同样也可以赋值：

```
int i, *i_point;
```

```
i_point=&i;
```



也可以在定义**指针变量**时赋初值：

```
int i;
```

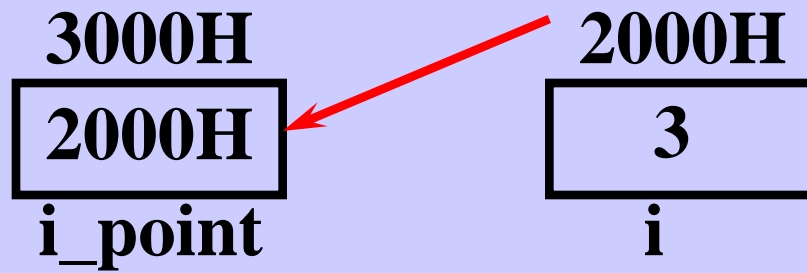
```
int *i_point=&i;
```



一个指针变量只能指向同一类型的变量。即整型指针变量只能放整型数据的地址，而不能放其它类型数据的地址。

* 在**定义语句**中只表示变量的类型是指针，没有任何计算意义。

* 在语句中表示“指向”。&表示“地址”⁴²³。



`int i;`
`int *i_point=&i;`

表示类型 (Indicates type)

`*i_point=3;`

表示指向 (Indicates pointing)

指针变量的引用

指针变量只能存放地址，不要将非地址数据赋给指针变量。

```
int *p, i;  p=100;    p=&i;

void main(void)
{ int a=10, b=100;

  int *p1, *p2;
  p1=&a; p2=&b;

  cout<<a<<'\t'<<b<<endl;

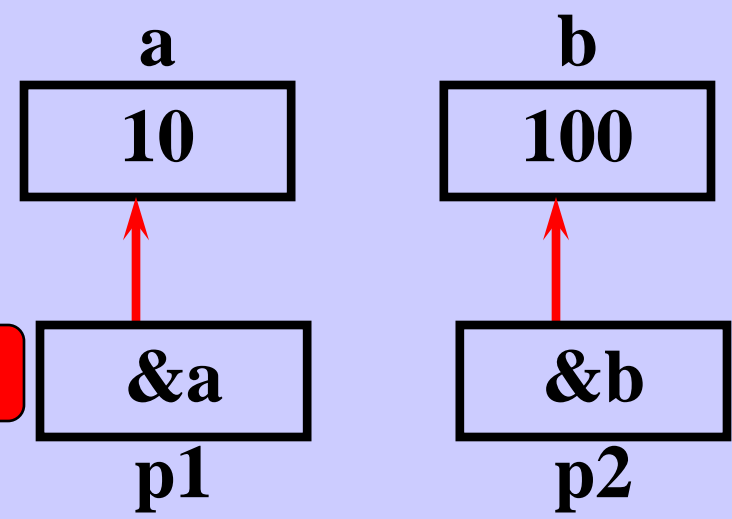
  cout<<*p1<<'\t'<<*p2<<endl;
}
```

非法

指针变量赋值

表示指向

指针变量引用



10 100

10 100

```
void main(void)
```

```
{ int a, b;
```

```
int *p1, *p2;
```

指针变量赋值

```
p1=&a; p2=&b;
```

```
*p1=10; *p2=100;
```

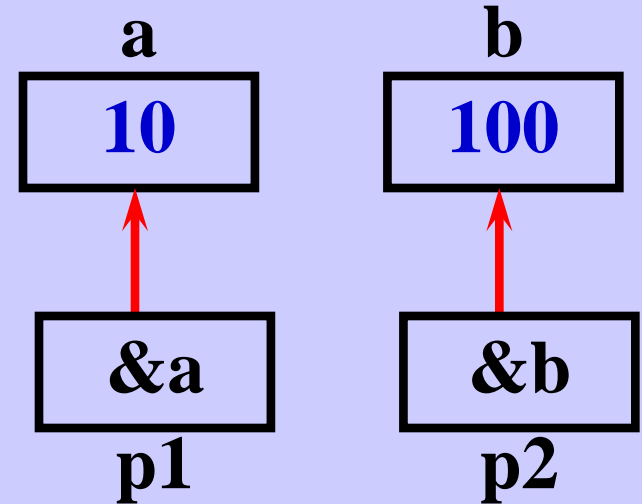
通过指针对
变量赋值

```
cout<<a<<'\t'<<b<<endl;
```

```
cout<<*p1<<'\t'<<*p2<<endl;
```

```
}
```

指针变量引用



```
void main(void)
```

```
{ int a, b;
```

```
int *p1, *p2;
```

```
*p1=10; *p2=100;
```

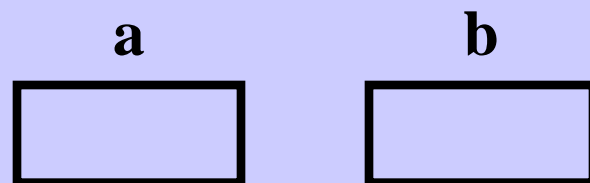
```
cout<<a<<'\t'<<b<<endl;
```

```
cout<<*p1<<'\t'<<*p2<<endl;
```

```
}
```

但指针变量未赋值，即
指针指向未知地址

通过指针对
变量赋值



随机

p1

随机

p2

绝对不能对未赋值的指针变量作“指向”运算。

```
int i, *p1;
```

```
p1=&i;
```

用指针变量前，必须
对指针变量赋值

输入a, b两个整数，按大小输出这两个数。

```
void main(void)
```

```
{ int *p1, *p2, *p, a, b;
```

```
cin>>a>>b;
```

```
p1=&a; p2=&b;
```

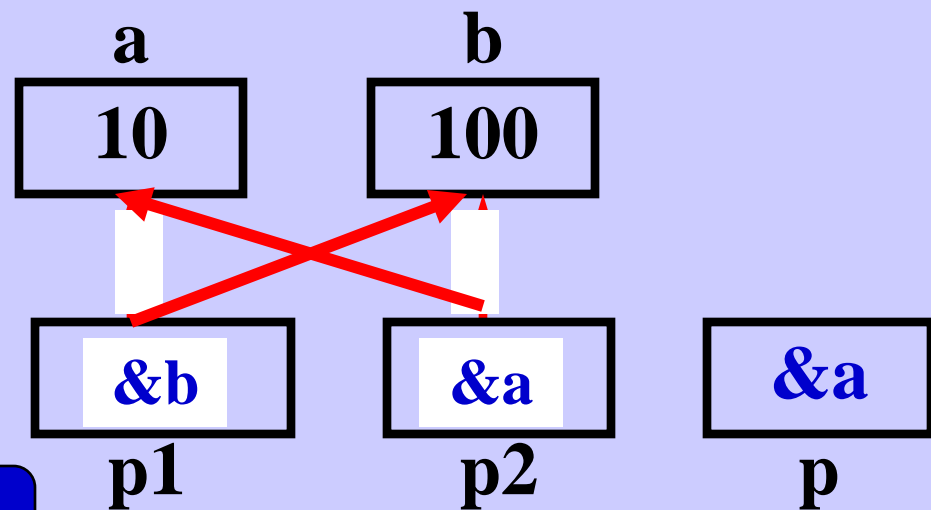
```
if (a<b)
```

```
{ p=p1; p1=p2; p2=p; }
```

```
cout<<a<<'\t'<<b<<endl;
```

```
cout<<*p1<<'\t'<<*p2<<endl;
```

```
}
```



10 100

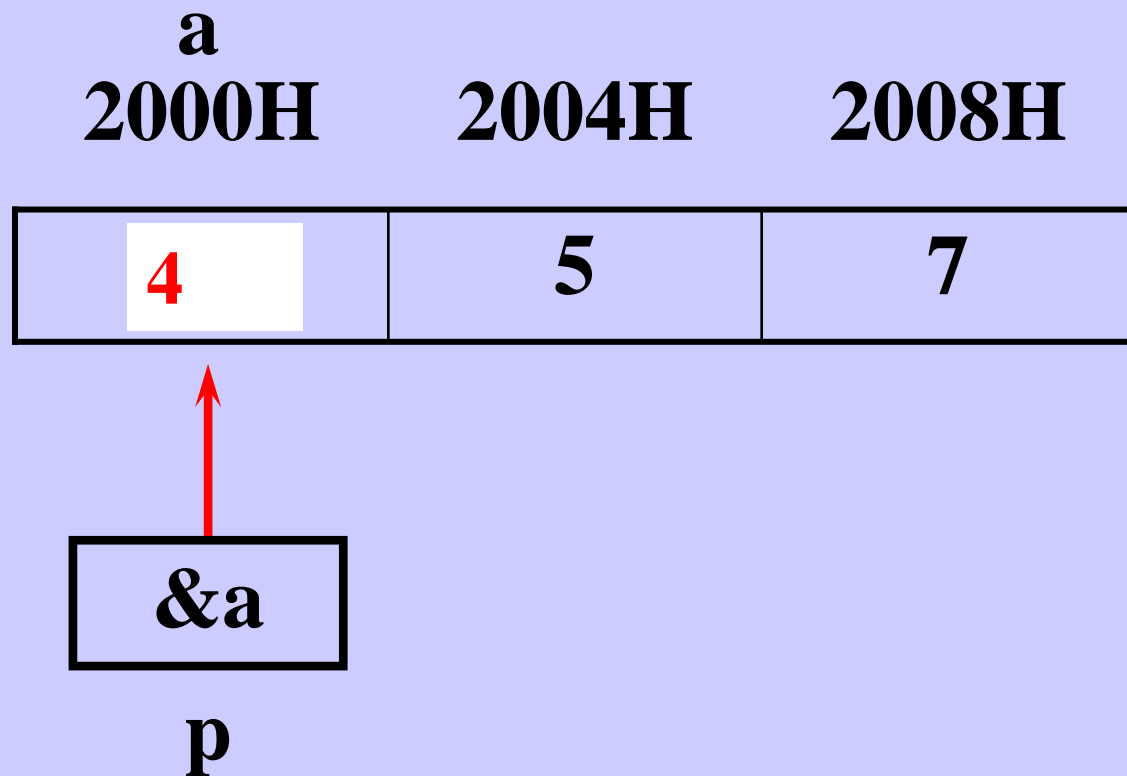
100 10

虽然变量不变，但指向变量的指针发生变化

++, --, * 优先级相同，都是右结合性。

```
int a=3, *p;
```

```
p=&a;
```

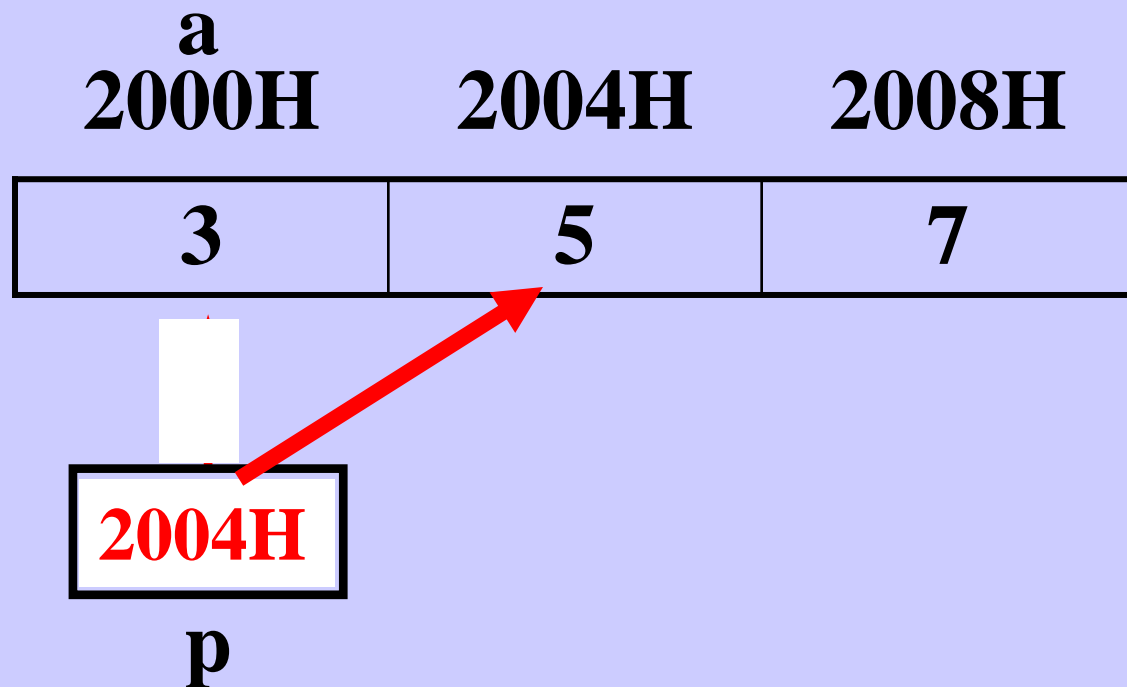


$(*p)++$; 相当于 $a++$ 。表达式为3, $a=4$

++, --, * 优先级相同，都是右结合性。

```
int a=3, *p;
```

```
p=&a;
```

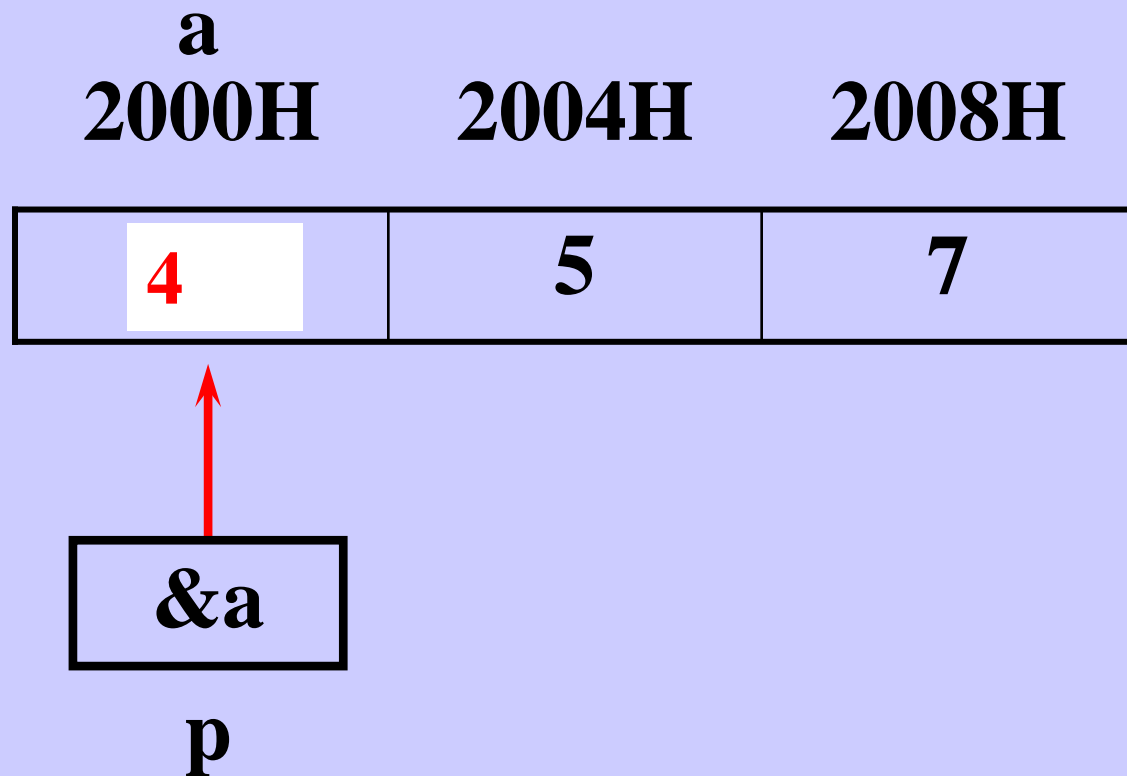


*p++; *(p++)首先*p，然后p=p+1,指针指向下一个int单元 表达式为3, p=2004H。

++, --, * 优先级相同，都是右结合性。

```
int a=3, *p;
```

```
p=&a;
```



```
++*p
```

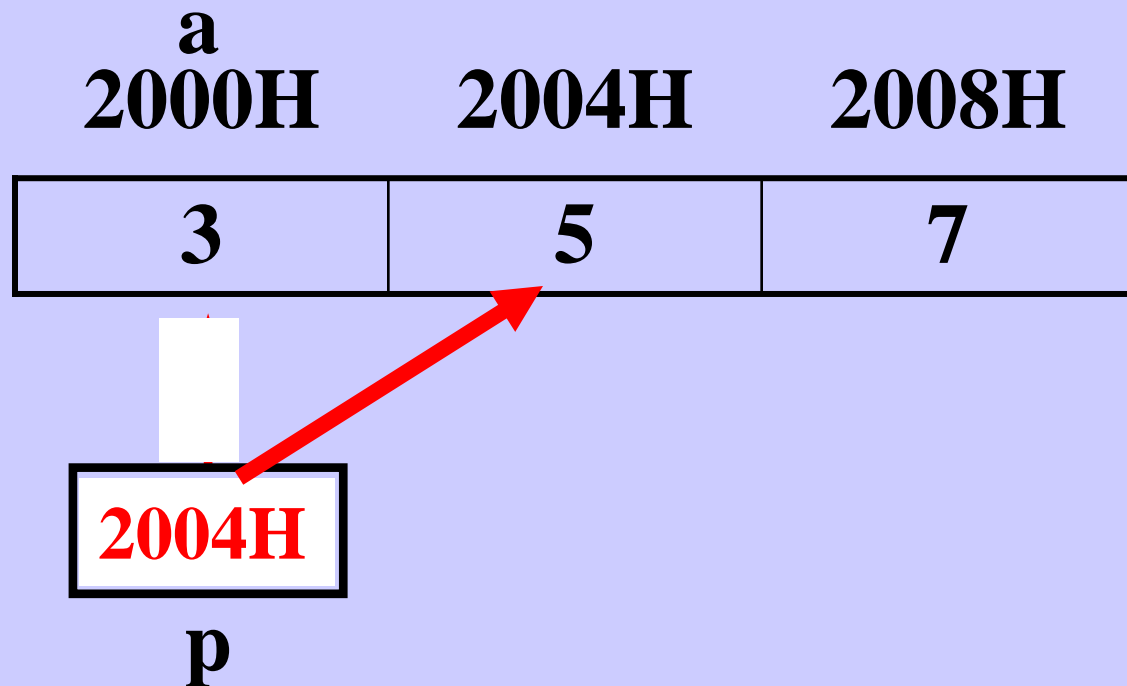
```
++(*p)
```

```
*p=*p+1 a=4
```

++, --, * 优先级相同，都是右结合性。

```
int a=3, *p;
```

```
p=&a;
```



++p** ***(++p)**, 首先: **p=p+1**, 然后取p**。即取**p**所指的下一个**int**单元的内容。

表达式为5 **p=2004H**

指针变量作为函数参数：

函数的参数可以是指针类型，它的作用是将一个变量的地址传送到另一个函数中。

指针变量作为函数参数与变量本身作函数参数不同，变量作函数参数传递的是具体值，而指针作函数参数传递的是内存的地址。

输入a, b两个整数，按大小输出这两个数。

```
swap(int *p1, int *p2)
```

```
{ int t;
```

```
  t=*p1; *p1=*p2; *p2=t;
```

```
}
```

```
void main(void)
```

```
{ int *point1, *point2, a,b;
```

```
  cin>>a>>b;
```

```
  point1=&a; point2=&b;
```

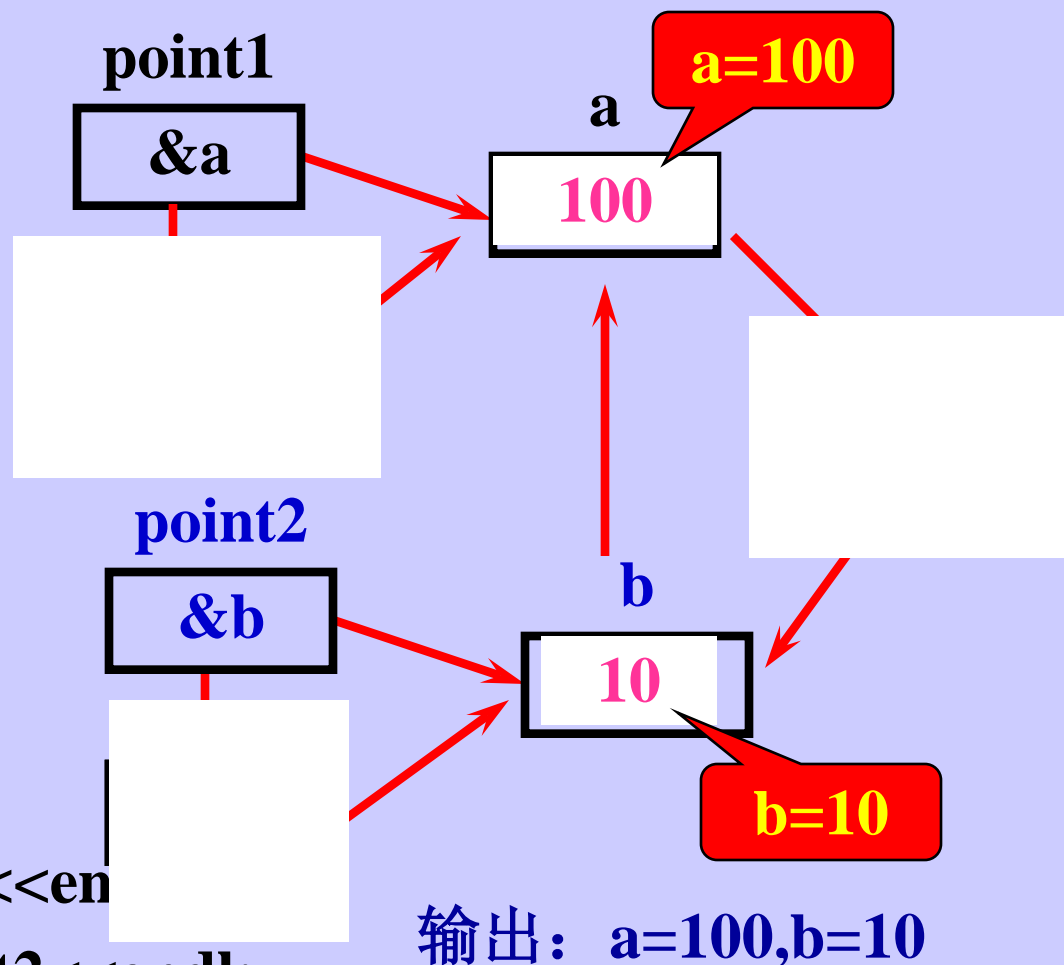
```
  if (a<b)
```

```
      swap (point1, point2);
```

```
  cout<<"a="<<a<<"<<b<<endl;
```

```
  cout<<*point1<<"<<*point2<<endl;
```

```
}
```



100,10

输入a, b两个整数，按大小输出这两个数。

```
swap(int x, int y)
```

```
{ int t;
```

```
  t=x;  x=y;  y=t;
```

```
}
```

```
void main(void)
```

```
{ int *point1, *point2, a,b;
```

```
  cin>>a>>b;
```

```
  point1=&a; point2=&b;
```

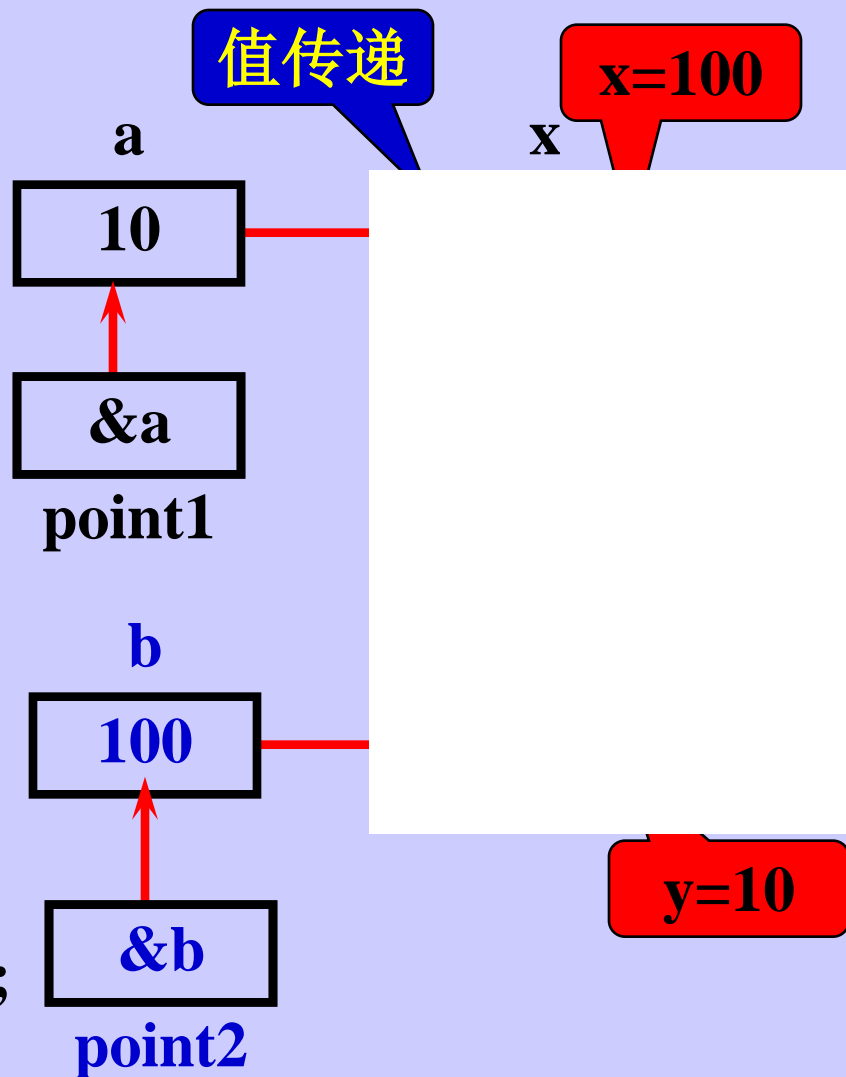
```
  if (a<b)
```

```
      swap (a, b);
```

```
  cout<<"a="<<a<<"<<"b="<<b<<"\n";
```

```
  cout<<*point1<<"\t"<<*point2;
```

```
}
```



输出: a=10,b=100

10,100

用指针变量作函数参数，在被调函数的执行过程中，应使指针变量所指向的参数值发生变化，这样，函数在调用结束后，其变化值才能保留回主调函数。

函数调用不能改变实参指针变量的值，但可以改变实参指针变量所指向变量的值。

用指针变量作函数参数，可以得到多个变化了的值。

```
void grt(int *x , int *y , int *z)
```

```
{  cout<< ++*x<<'<< ++*y<<'<<*(z++)<<endl;}
```

```
int  a=10, b=40, c=20;
```

```
void main(void)
```

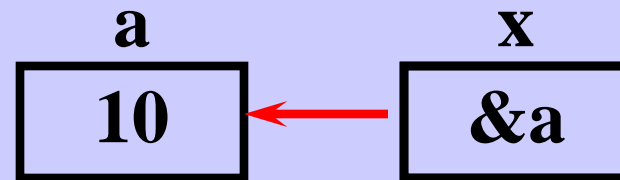
```
{  prt (&a, &b, &c);
```

11, 41, 20

```
    prt (&a, &b, &c);
```

12, 42, 20

```
}
```



++*x: *x=*x+1

***(z++): *z ; z=z+1**

```
int s( int *p)
```

```
{  int sum=10;
```

```
    sum=sum + *p;
```

```
    return sum;
```

```
} void main(void)
```

```
{  int a=0, i, *p, sum;
```

```
    for (i=0; i<=2; i++)
```

```
    {  p=&a;
```

```
        cin>>*p;
```

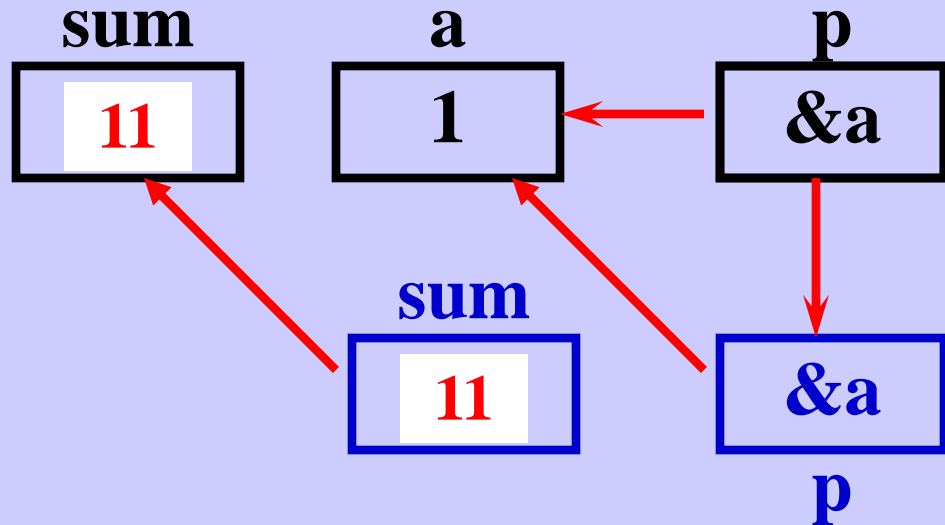
```
        sum=s(p);
```

```
        cout<<"sum="<<sum<<endl;
```

```
    }
```

```
}
```

输入: 1 3 5<CR>



sum=11

sum=13

sum=15

```
sub( int *s)
```

```
{ static int t=0;
```

```
  t=*s + t;
```

```
  return t;
```

```
}
```

```
void main(void)
```

```
{ int i, k;
```

```
  for (i=0; i<4; i++)
```

```
  { k=sub(&i);
```

```
    cout<<"sum="<<k<<"\t";
```

```
  }
```

```
  cout<<"\n";
```

```
}
```

i=0 t=*s+t=0 k=0 sum=0

i=1 t=*s+t=1 k=1 sum=1

i=2 t=*s+t=3 k=3 sum=3

i=3 t=*s+t=6 k=6 sum=6

sum=0

sum=1

sum=3

sum=6

```
int *p;
```

```
void main(void)
```

```
{ int a=1, b=2, c=3;
```

```
  p=&b;
```

```
  pp(a+c, &b);
```

```
  cout<<"(1)"<<a<<" "<<b<<" "<<*p<<endl;
```

```
}
```

```
pp(int a, int *b)
```

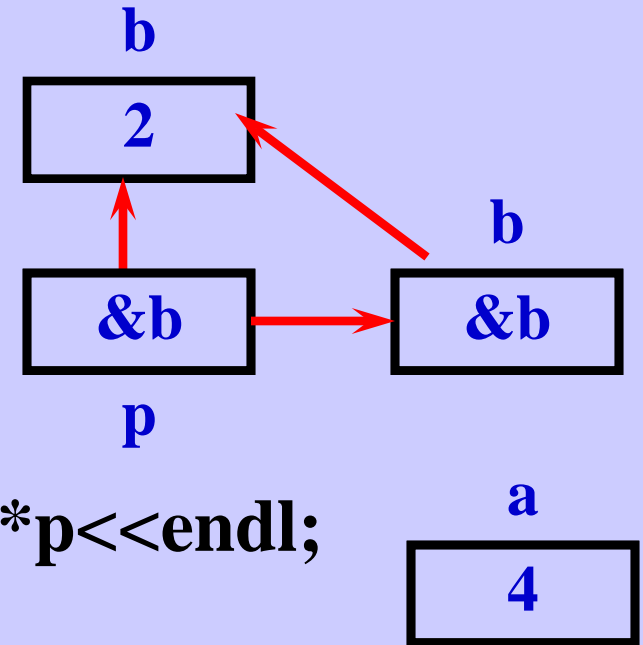
```
{ int c=4;
```

```
  *p=*b+c;
```

```
  a=*p-c;
```

```
  cout<<"(2)"<<a<<" "<<*b<<" "<<*p<<endl;
```

```
}
```



(2) 2 6 6

(1) 1 6 6

$*p = *b + 4 = 2 + 4 = 6$

$a = 6 - c = 2$

举例：最大最小值、方程根

数组的**指针**和指向数组的**指针变量**

数组与变量一样，在内存中占据单元，有地址，一样可以用指针来表示。C++规定：**数组名就是数组的起始地址**；又规定：**数组的指针就是数组的起始地址**。数组元素的指针就是数组元素的地址。

一、指向数组元素的指针变量的定义与赋值

```
int a[10], *p;
```

```
p=&a[0];
```

数组第一个元素的地址

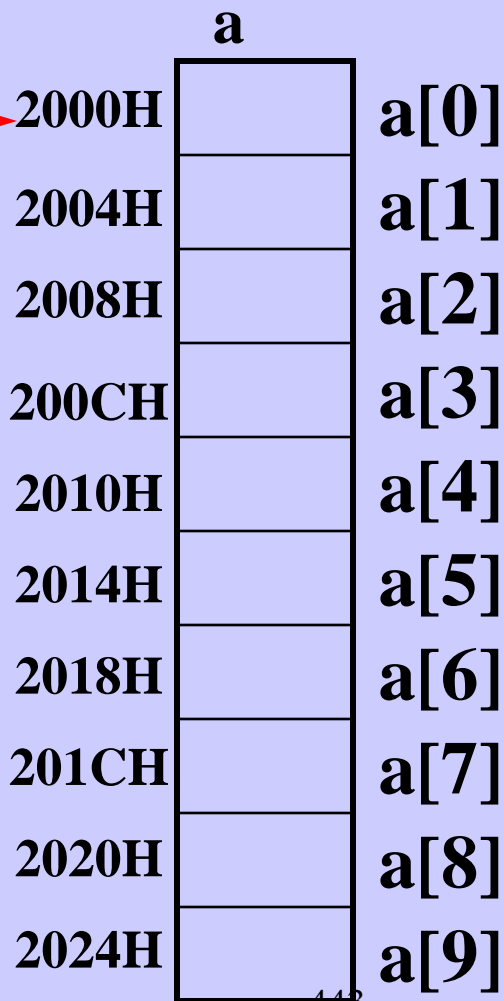
```
p=a;
```

直接用数组名赋值

&a[0]

p

p是变量，a为常量。



若数组元素为int型，则指向其的指针变量也应定义为int型。

```
int a[10];
```

这两种情况均为赋初值

```
int *p=a;    int *p=&a[0];
```

二、通过指针引用数组元素

```
int a[10];
```

```
int *p=a;
```

```
*p=1;
```

```
a[0]=1;
```

C++规定，**p+1**指向数组的下一个元素，而不是下一个字节。

```
*(p+1)=2;
```

```
a[1]=2;
```

指针变量也重新赋值

```
*++p=2;
```

```
p=p+1; *p=2; p=2004H
```

&a[0]

p

2000H

2004H

2008H

200CH

2010H

2014H

2018H

201CH

2020H

2024H

a

1

2

a[0]

a[1]

a[2]

a[3]

a[4]

a[5]

a[6]

a[7]

a[8]

a[9]

$*(p+1)=2;$ $a[1]=2;$

$*(a+1)=2;$

$*(a+1)$ 与 $a[1]$ 等同。

$*++p=2;$ $p=p+1;$ $*p=2;$ $p=2004H$

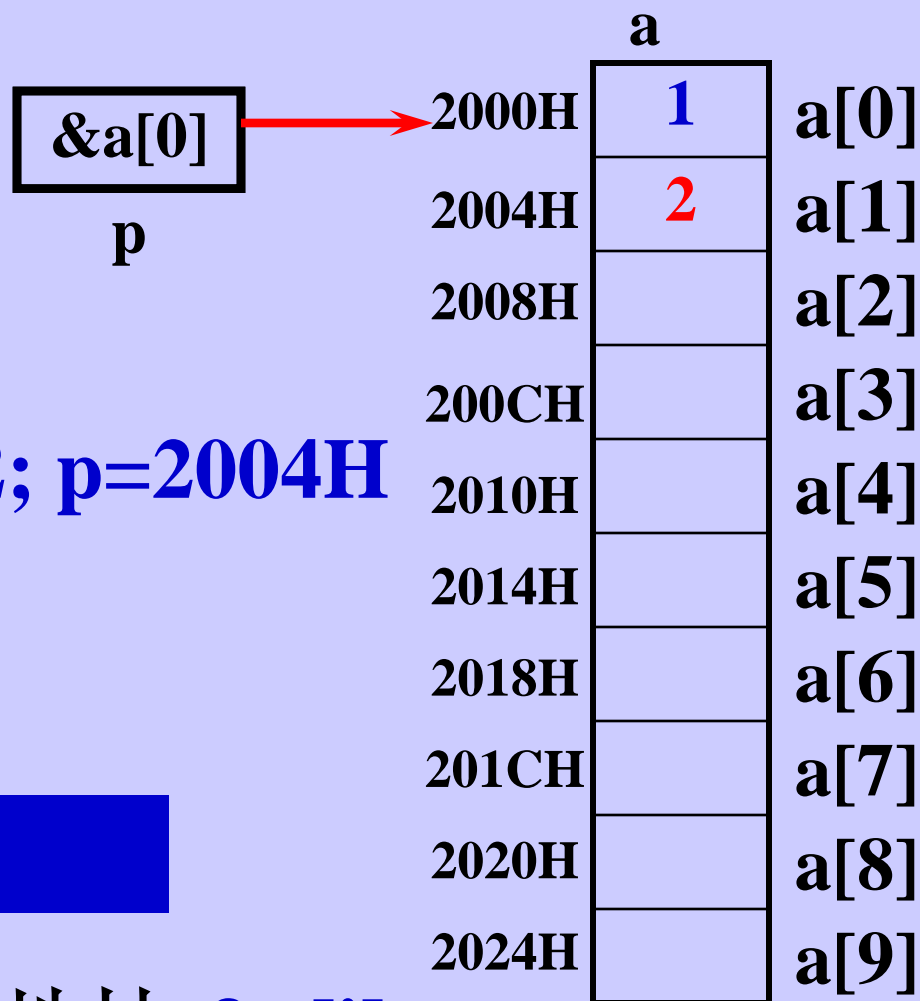
$*++a=2;$

错误

a 为常量，不可赋值。

$p+i$ 或 $a+i$ 均表示 $a[i]$ 的地址 $\&a[i]$

$*(a+i) \longleftrightarrow a[i]$ $*(p+i) \longleftrightarrow p[i]$



用指向数组的指针变量输出数组的全部元素

```
void main(void)
```

```
{ int a[10], i;
```

```
int *p;
```

```
for (i=0; i<10; i++)
```

```
cin>>a[i];
```

输入数组元素

指针变量赋初值

指向下一元素

```
for (p=a; p<a+10; p++)
```

```
cout<<*p<<'\t';
```

输出指针指向的数据

```
}
```

```
void main(void)
```

```
{ int a[10], i;
```

```
int *p=a;
```

```
for (i=0; i<10; i++)
```

```
cin>>a[i];
```

```
for (i=0; i<10; i++)
```

```
cout<<*p++<<'\t';
```

*p, p=p+1

输出数据后指针加1

```
}
```

```
void main(void)
```

```
{  int  x[ ]={1,2,3};
```

```
    int  s, i, *p;
```

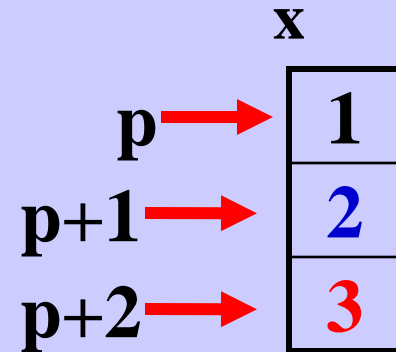
```
    s=1; p=x;
```

```
    for (i=0; i<3; i++)
```

```
        s*=(p+i);
```

```
    cout<<s<<endl;
```

```
}
```



i=0 **s=s*(*(**p+0**)) =s*1=1**

i=1 **s=s*(*(**p+1**)) =s*2=2**

i=2 **s=s*(*(**p+2**)) =s*3=6**

6

```
static int  a[ ]={1, 3,5, 7, 11, 13};
```

```
main( )
```

```
{  int *p;
```

```
    p=a+3;
```

```
    cout<<*p<<'\t'<<(*p++)<<endl;
```

```
    cout<<*(p-2)<<'\t'<<*(a+4)<<endl;
```

```
}
```

p

&a[3]



1	a[0]
3	a[1]
5	a[2]
7	a[3]
11	a[4]
13	a[5]

11 7
5 11

三、数组名作函数参数

数组名可以作函数的实参和形参，传递的是**数组的地址**。这样，实参、形参共同指向同一段内存单元，内存单元中的数据发生变化，这种变化会反应到主调函数内。

在函数调用时，**形参数组并没有另外开辟新的存储单元**，而是以实参数组的首地址作为形参数组的首地址。这样**形参数组的元素值发生了变化也就使实参数组的元素值发生了变化**。

1、形参实参都用数组名

```
void main(void)
```

```
{ int array[10];
```

```
.....
```

```
f(array, 10);
```

```
.....
```

```
}
```

形参数组,必须进行类型说明

```
f(int arr[ ], int n)
```

```
{
```

```
.....
```

```
}
```

实参数组

用数组名作形参，因为接收的是地址，所以可以不指定具体的元素个数。

指向同一
存储区间

array, **arr** arr[0]

2000H		array[0]
2004H		array[1]
2008H		array[2]
200CH		array[3]
2010H		array[4]
2014H		array[5]
201CH		array[6]
2020H		array[7]
2024H		array[8]
2028H		array[9]

2、实参用数组名，形参用指针变量

```
void main(void)
```

```
{ int a [10];
```

```
.....
```

```
f(a, 10);
```

```
.....
```

```
}
```

```
f(int *x, int n )
```

```
{
```

形参指针

```
.....
```

```
}
```

实参数组

3、形参实参都用指针变量

```
void main(void)
```

```
{ int a [10], *p;
```

```
  p=a;
```

```
  ....
```

```
  f(p, 10);
```

```
  ....
```

```
}
```

形参指针

```
f(int *x, int n )
```

```
{
```

```
  ....
```

```
}
```

实参指针

实参指针变量调用前必须赋值

4、实参为指针变量，形参为数组名

```
void main(void)
```

```
{ int a [10], *p;
```

```
    p=a;
```

```
    .....
```

```
    f(p, 10);
```

```
    .....
```

```
}
```

形参数组

```
f(int x[ ], int n )
```

```
{
```

```
    .....
```

```
}
```

将数组中的n个数按相反顺序存放。

```
void inv(int x[ ], int n)
{ int t, i, j, m=(n-1)/2;
  for (i=0; i<=m; i++)
  { j=n-1-i;
    t=x[i]; x[i]=x[j]; x[j]=t;
  }
}

void main(void)
{ int i, a[10]={3,7,9,11,0,6,7,5,4,2};
  inv(a,10);
  for (i=0; i<10; i++)
    cout<<a[i]<<'\t';
}
```

x与a数组指向同一段内存

	x, a	
x[0]	3	a[0]
x[1]	7	a[1]
x[2]	9	a[2]
x[3]	11	a[3]
x[4]	0	a[4]
x[5]	6	a[5]
x[6]	7	a[6]
x[7]	5	a[7]
x[8]	4	a[8]
x[9]	2	a[9]

```

void inv(int *x, int n)
{
    int *p, t, *i, *j, m=(n-1)/2;
    i=x; j=x+n-1; p=x+m;
    for (; i<=p; i++,j--)
        { t=*i; *i=*j; *j=t;
        }
}

void main(void)
{
    int i, a[10]={3,7,9,11,0,6,7,5,4,2};
    inv(a,10);
    for (i=0;i<10; i++)
        cout<<a[i]<<'\t';
}

```

用指针变量来
接受地址

		x, a	
i →	x[0]	3	a[0]
	x[1]	7	a[1]
	x[2]	9	a[2]
	x[3]	11	a[3]
p →	x[4]	0	a[4]
	x[5]	6	a[5]
	x[6]	7	a[6]
	x[7]	5	a[7]
j →	x[8]	4	a[8]
	x[9]	2	a[9]

输入10个整数，将其中最小的数与第一个数对换，把最大的数与最后一个数对换。写3个函数：①输入10个数；②进行处理；③输出10个数。

编写函数 `int fun(int x, int *pp)`,其功能是，求出能整除`x`且不是偶数的各整数，并按照从小到大的顺序放在`pp`指向的内存中，函数返回值为这些整数的个数。若`x`的值为30，数组中的数为1，3，5，15，函数返回4。

```
int fun(int x, int *pp)
{ int k=0;
  for(int i=1;i<x;i++)
  { if(i%2&& x%i==0)
    { *(pp++)=i;
      k++;
    }
  }
  return k;
}
```

```
void main(void)
{ int a[1000],x,n;
  cin>>x;
  n=fun(x,a);
  for(int i=0;i<n;i++)
    cout<<a[i]<<'\\t';
  cout<<endl;
}
```

输入一行字符串，将字符串中所有下标为奇数位置上的字母转换为大写（若不是小写字符则不必转换）。

```
void change(char *pchar)
```

```
{ while(*pchar)
```

```
{ if(*pchar>='a' &&*pchar<='z')
```

```
    *pchar=*pchar-32;
```

```
    pchar++;
```

```
    if(*pchar==0)
```

```
        break;
```

```
    pchar++;
```

```
}
```

```
}
```

```
void main(void)
```

```
{ char str[100];
```

```
    cin.getline(str,100);
```

```
    change(str);
```

```
    cout<<str<<endl;
```

```
}
```

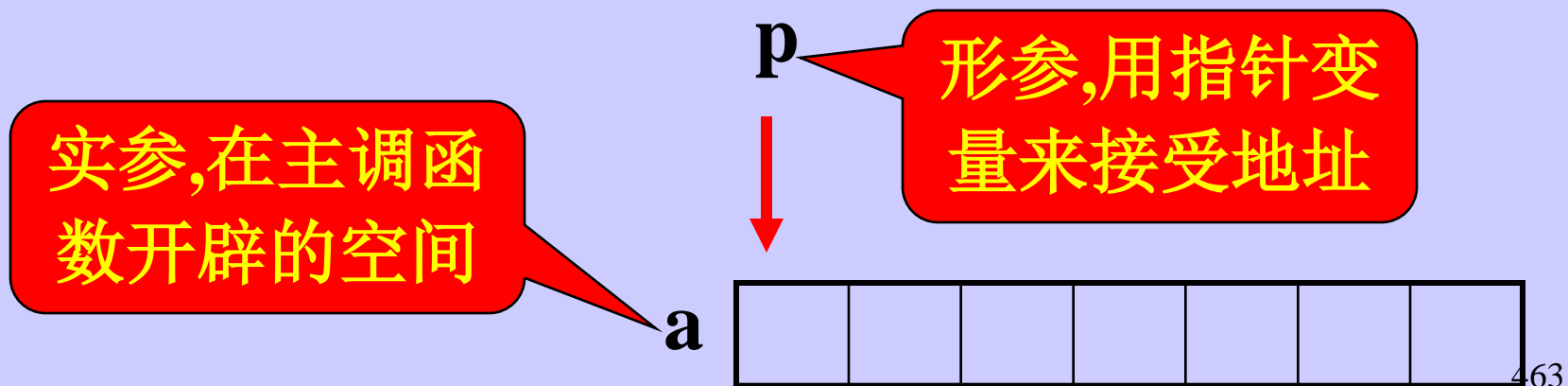
数组名作函数参数

数组名可以作函数的实参和形参，传递的是**数组的地址**。这样，实参、形参共同指向同一段内存单元，内存单元中的数据发生变化，这种变化会反应到主调函数内。

在函数调用时，**形参数组并没有另外开辟新的存储单元**，而是以实参数组的首地址作为形参数组的首地址。这样**形参数组的元素值发生了变化也就使实参数组的元素值发生了变化**。

既然数组做形参没有开辟新的内存单元，接受的只是实参数组的首地址，那么，这个首地址也可以在被调函数中用一个**指针变量**来接受，通过在被调函数中对这个**指针变量的指向**进行操作而使实参数组发生变化。

实际上在被调函数中只开辟了p的空间，里面放的是a的值。



四、指向多维数组的指针和指针变量

用指针变量也可以指向多维数组，表示的同样也是多维数组的首地址。

```
int a[3][4]; //首地址为2000H
```

2000H		2008H		2010H	2014H		201cH	2020H		2028H	202cH
a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

a		2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

可以将a数组看作一个一维数组，这个一维数组的每个元素又是一个具有4个int型数据的一维数组，这样，**我们就可以利用一维数组的概念来标记一些写法。**

	a	2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

a[0]=*(a+0) a+0为a[0]的地址&a[0]，其值为2000H。

a[1]=*(a+1) a+1为a[1]的地址&a[1]，其值为2010H。

a[2]=*(a+2) a+2为a[2]的地址&a[2]，其值为2020H。

a[0]为一维数组名，其数组有四个int型的元素：

a[0][0], a[0][1], a[0][2], a[0][3]

同样，**a[0]**代表一维数组的首地址，所以，a[0]为&a[0][0]。

	a	2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

把a[0]看成
一维数组b

a[0]代表一维数组的首地址,也就是一维数组名,a[0]为
&a[0][0]。

a[0]为&a[0][0] a[0][0]=*(a[0]+0) b[0] = *(b+0)

a[0]+1为&a[0][1] a[0][1]=*(a[0]+1) b[1] = *(b+1)

a[0]+2为&a[0][2] a[0][2]=*(a[0]+2) b[2] = *(b+2)

a[0]+3为&a[0][3] a[0][3]=*(a[0]+3) b[3] = *(b+3)

行

列

a[1]+2为&a[1][2] a[1][2]=*(a[1]+2)

a[i][j]=*(a[i]+j)

	a	2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

行

列

$a[1]+2$ 为 $\&a[1][2]$ $a[1][2]=*(a[1]+2)$ $a[i][j]=*(a[i]+j)$

a 为二维数组名， $a+1$ 为 $a[1]$ 的地址，也就是数组第一行的地址，所以 a 为行指针。

$a[1]$ 为一维数组名， $a[1]+1$ 为 $a[1][1]$ 的地址，也就是数组第一行第一列的地址，所以 $a[1]$ 为列指针。

a		2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

可以看到：a, a+0, *(a+0), a[0], &a[0][0]表示的都是2000H，即二维数组的首地址。

实际上，a[0], a[1], a[2]并不是实际的元素，它们在内存并不占具体的存储单元，只是为了我们表示方便起见而设计出的一种表示方式。

a为行指针，加1移动一行。

***a或a[0]为列指针，加1移动一列。**

	a	2000H	2004H	2008H	200CH
2000H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
2010H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2020H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

a[1]=*(a+1) *(a+1)+2=&a[1][2]

***(*(a+1)+2)=a[1][2]**

**** (a+1)=*(a[1])=*(*(a+1)+0)=a[1][0]**

((*(a+1))[1]=*(*(a+1)+1)=a[1][1]

***(a+1)[1]=*((a+1)[1])=*((*(a+1)+1))=**(a+2)=a[2][0]**

注意二维数组的各种表示法，a为常量。

	a	1900H	1904H	1908H	190CH
1900H	a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1910H	a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
1920H	a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

int a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};

设数组的首地址为1900H， 则：

a为 1900H

*a为 1900H

a+2为 1920H

*a+2为 1908H

*(a+1)+2为 1918H

**a为 1

*(a+9)为 19

(a+1)[1]为 1920H

```
void main(void)
```

p+1 a[0]+1

```
{ static int
```

```
a[3][4]={1,3,5,7,9,11,13,15,17,19,21,23};
```



p

```
int *p; a[0]是列指针
```

```
for(p=a[0]; p<a[0]+12 ; p++)
```

```
{ if((p-a[0])%4==0) cout<<endl;
```

a+1

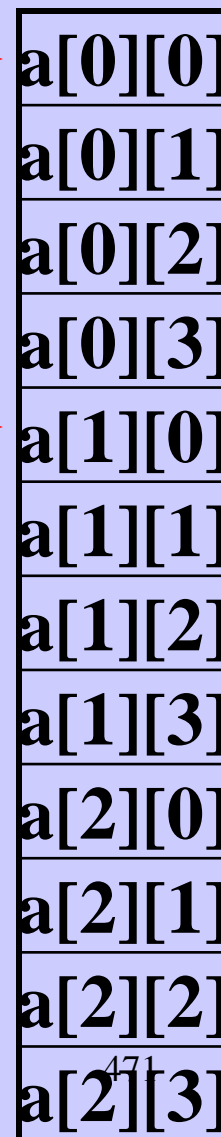
```
cout<<*p<<'\t';
```

类型不匹配!

```
} for(p=a; p<a+12 ; p++)
```

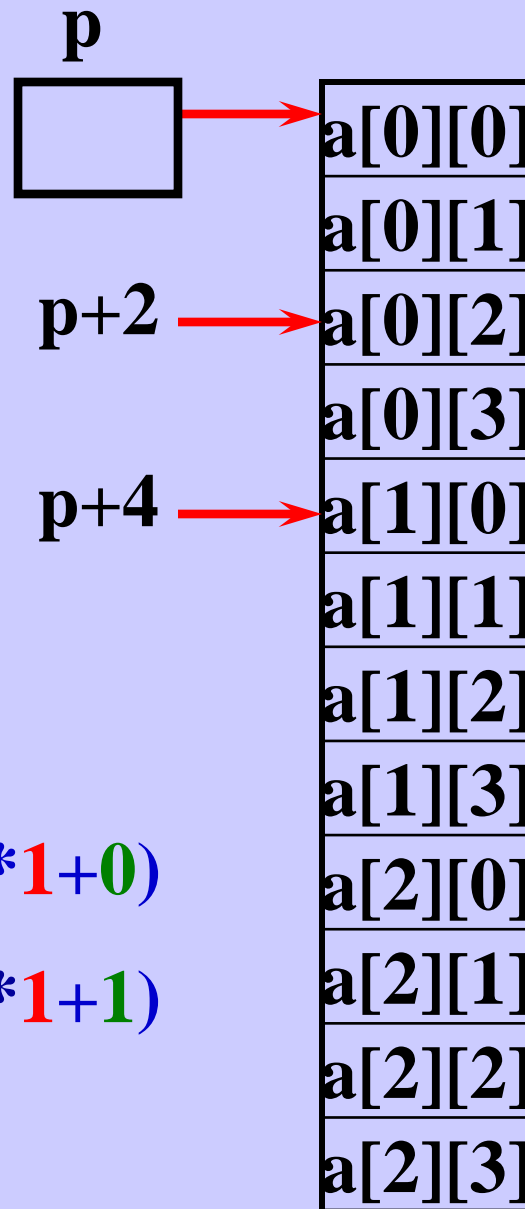
```
}
```

1	3	5	7
9	11	13	15
17	19	21	23



`int a[3][4],*p;`

如何用 **p** 表示二维数组中的任一元素？



`p=a[0] *p *(a[0]+0) a[0][0]`

`p+2`

`p+1 *(a[0]+1) a[0][1]`

`p+4`

`p+2 *(a[0]+2) a[0][2]`

`p+3 *(a[0]+3) a[0][3]`

`p+4 *(a[0]+4) a[1][0] *(p+4*1+0)`

`p+5 *(a[0]+5) a[1][1] *(p+4*1+1)`

`a[i][j] *(p+4*i+j) *(p+m*i+j)`

m为第二维的维数。是个常数


```
fun(int *q1, int *q2, int *q3)
```

```
{ *q3=*q2+*q1; }
```

```
void main ( void)
```

```
{ int i , j, a[3][3]={1,1},*p1,*p2,*p3;
```

```
p1=a[0], p2=a[0]+1,p3=a[0]+2;
```

```
for(i=2;i<9;i++)
```

```
    fun(p1++, p2++, p3++);
```

```
for(i=0;i<3; i++)
```

```
    for(j=0;j<3;j++)
```

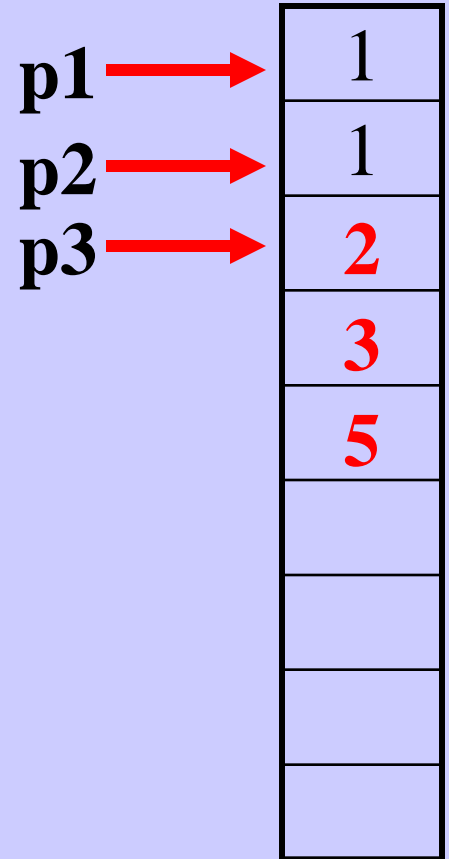
```
    { cout<<a[i][j]<<'\t';
```

```
        cout<<endl;
```

```
}
```

程序输出的第一行为1

第二行为1 第三行为2



i=2 *q3=2

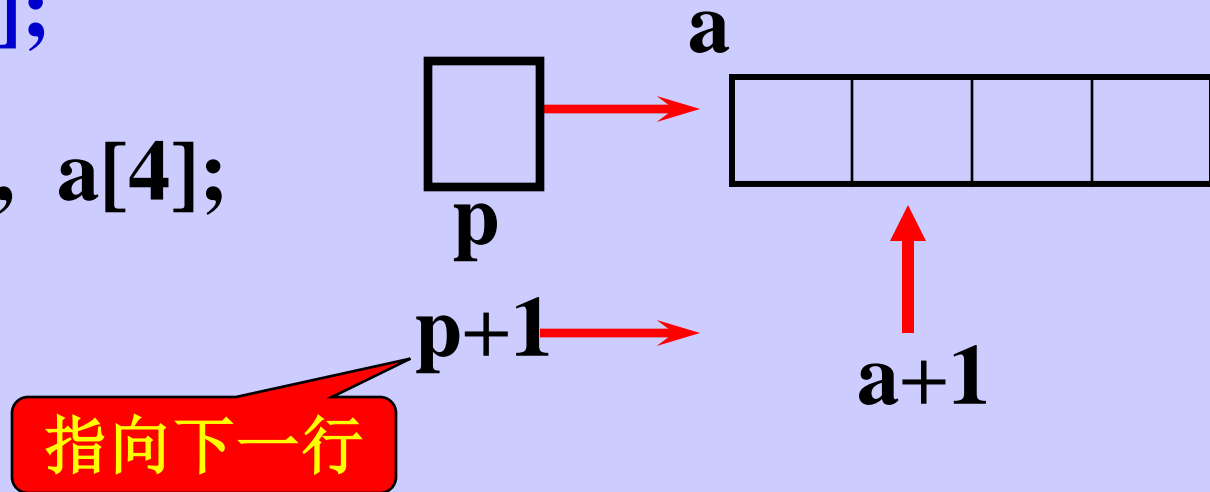
i=3 *q3=3

i=4 *q3=5

指向由m个整数组成的**一维数组**的指针变量

```
int (*p)[m];
```

```
int (*p)[4], a[4];
```



`p+1`指针加16个字节。`a+1`指针加4个字节。

```
int (*p)[4], a[3][4];
```

a		2000H	2004H	2008H	200CH
p	→	a[0][0]	a[0][1]	a[0][2]	a[0][3]
p+1	→	a[1][0]	a[1][1]	a[1][2]	a[1][3]
p+2	→	a[2][0]	a[2][1]	a[2][2]	a[2][3]

p为行指针，可以直接将二维数组名**a**赋给**p**。这样，**a**与**p**等同。 **p=a**

a[2][3] ***(*(a+2)+3)**

p[2][3] ***(*(p+2)+3)**

两种表示
完全等价

a为常量
p为变量

若有以下的定义和语句，则下面各个符号的正确含义是：

int a[3][4] , (*p)[4];

p=a;

		a	2000H	2004H	2008H	200CH
p	→		a[0][0]	a[0][1]	a[0][2]	a[0][3]
p+1	→		a[1][0]	a[1][1]	a[1][2]	a[1][3]
p+2	→		a[2][0]	a[2][1]	a[2][2]	a[2][3]

1. **p+1** **a**数组第一行元素的首地址 为行指针

2. ***(p+2)** **a**数组第二行元素的首地址 为列指针 **&a[2][0]**

3. ***(p+1)+2** **&a[1][2]** 为列指针

4. ***(p+2)** 数据元素 ***(p+2)=a[0][2]**

若有以下的定义和语句，则对a数组元素的非法引用是：

```
int a[2][3], (*pt)[3];
```

```
pt=a;
```

1) `pt[0][0]` 2) `*(pt+1)[2]`

3) `*(pt[1]+2)` 3) `*(a[0]+2)`

`*(pt+1)[2]` 右结合性 $= *((pt+1)[2])$
 $= *(*pt+1+2)$
 $= *(*pt+3) = pt[3][0]$ ₄₁₇

多维数组的指针作函数参数

主要注意的是函数的**实参**究竟是行指针还是列指针，从而决定函数**形参**的类型。要求**实参、形参一一对应，类型一致**。（举例）

求二维数组a[3][4]的平均值。

```
void main(void)
```

```
{    float score[3][4] = { {65,67,70 ,60}, {80,87,90,81},  
    {90,99,100,98} };
```

```
    float sum=0;
```

```
    for(int i=0;i<3;i++)
```

```
        for(int j=0;j<4;j++)
```

```
            sum=sum+score[i][j];
```

```
    cout<<“aver=“<<sum/12<<endl;
```

```
void main(void)
```

```
{    float score[3][4] = { {65,67,70 ,60}, {80,87,90,81},  
    {90,99,100,98} };
```

```
    float aver;
```

```
    aver=fun1(score, 3);
```

```
    aver=fun2(*score, 12);
```

```
p 161 8.15
```


字符串的**指针**和指向字符串的**指针变量**

字符串的表示形式

string

数组首地址

1、用字符数组实现

I		l	o	v	e		C	h	i	n	a	\0
---	--	---	---	---	---	--	---	---	---	---	---	----

```
void main(void )
```

```
{ char string[ ]="I love China";
```

```
    cout<<string;
```

```
}
```

string为数组名，代表数组的首地址，是常量。

string

I			l	o	v	e		C	h	i	n	a	\0
---	--	--	---	---	---	---	--	---	---	---	---	---	----

```
char string[20];
```

```
string="I love China";
```

错误！常量不能赋值

```
strcpy(string, "I love China");
```

正确赋值形式

```
cin.getline(string); //从键盘输入
```

2、用字符**指针**表示字符串

```
void main(void)
```

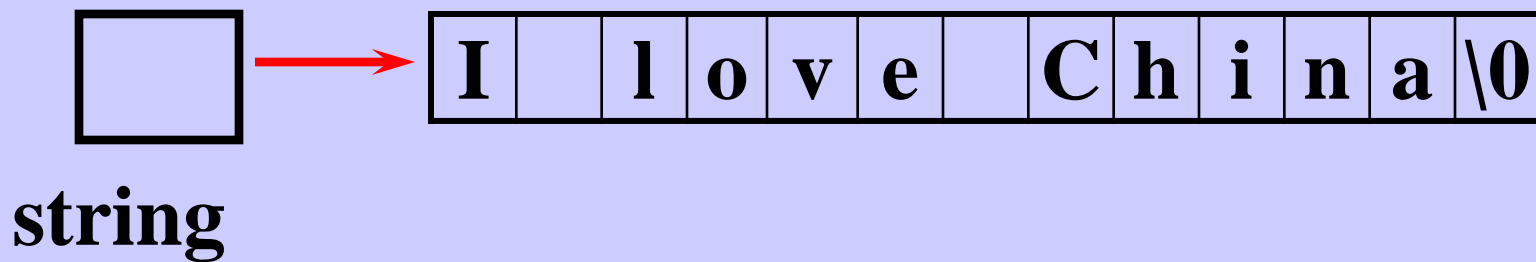
指针变量

```
{ char *string="I love China";
```

```
    cout<<string;
```

字符串常量

```
}
```



将内存中字符串常量的首地址赋给一个指针变量

```
void main(void)
```

```
{ char *string;
```

```
    string="I love China";
```

```
}
```

指针变量赋值，合法

具体字符

```
*string="I love China";
```

```
char *string;
```

```
cin.getline(string);
```

指针未赋值就作指向运算

将字符串a复制到字符串b。

```
void main(void)
```

```
{ char a[ ]="I am a boy", b[20];
```

```
int i;
```

```
for(i=0; *(a+i)!='\0'; i++)
```

```
    *(b+i)=*(a+i);
```

```
    *(b+i)='\0';
```

```
cout<<a<<endl;
```

```
cout<<b<<endl;
```

```
}
```

$i=0$ $*(b+i)=*(a+i)$

$b[i]=a[i]$

$i=1$

$i=2$

a

I		a	m		a		b	o	y	\0
---	--	---	---	--	---	--	---	---	---	----



I		a							y	\0
---	--	---	--	--	--	--	--	--	---	----

b

必须以\0结束

```
void main(void)
```

```
{ char a[ ]="I am a boy", b[20];
```

```
char *p1, *p2;
```

```
int i;
```

```
p1=a; p2=b;
```

```
for(; *p1!='\0'; p1++,p2++)
```

```
    *p2=*p1;
```

```
    *p2='\0';
```

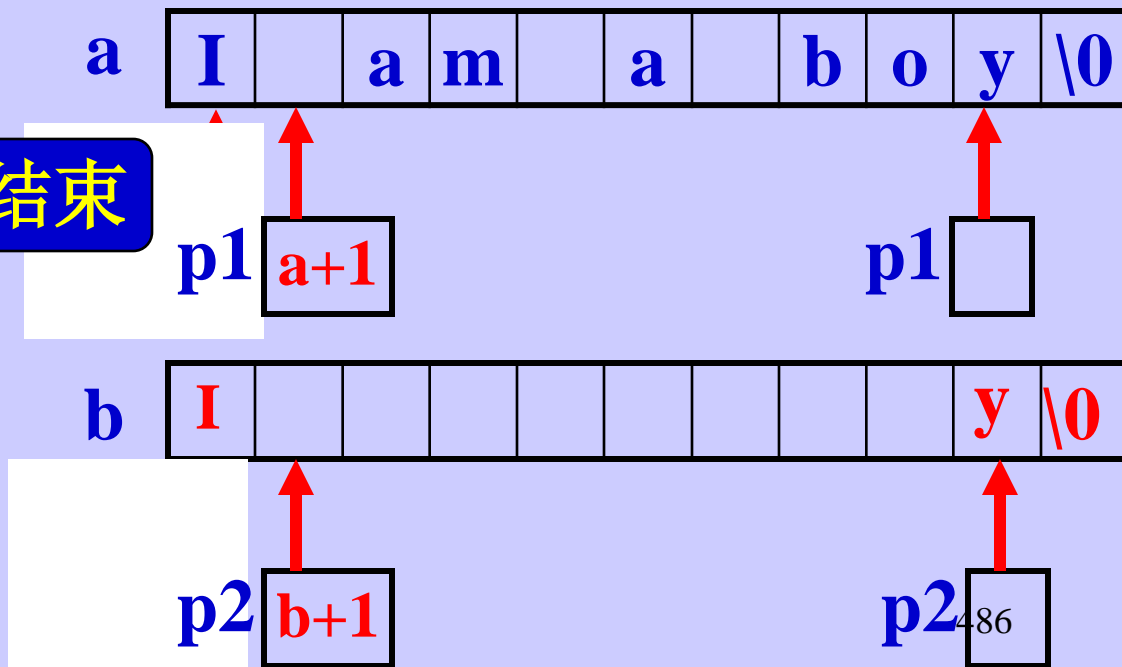
必须以\0结束

```
cout<<a<<endl;
```

```
cout<<b<<endl;
```

```
}
```

***p2=*p1**



```
void main(void)
```

```
{ char a[ ]="I am a boy", b[20];
```

```
char *p1, *p2;
```

```
for(; *p1!='\0'; )
```

```
int i;
```

```
*p2++=*p1++;
```

```
p1=a; p2=b;
```

```
*p2='\0';
```

```
for(; *p1!='\0'; p1++,p2++)
```

```
*p2=*p1;
```

```
while(*p2++=*p1++);
```

```
*p2='\0';
```

```
for(; *p2++=*p1++ ; );
```

```
cout<<a<<endl;
```

```
cout<<b<<endl;
```

```
for(; (*p2++=*p1++)!='\0' ; );
```

```
}
```

以下程序判断输入的字符串是否“回文”，若是回文，输出YES。

```
void main(void)
{ char s[81], cr, *pi, *pj;
  int i, j, n;
  cin.getline(s);  n=strlen(s);
  pi=__s____; pj=__s+n-1____; //pi指向串开始, pj指向最后
  while(*pi==' ') pi++;
  while(*pj==' ') pj--;
  while( ( __pi<pj____ ) &&(*pi==*pj) )
      { pi++; __pj--____; }
  if((pi<pj) cout<<"NO"<<endl;
  else  cout<<"YES\n";
}
```


字符串**指针**作函数参数

将一个字符串从一个函数传递到另一个函数，可以用地址传递的办法。即用**字符数组名作参数或用指向字符串的指针变量作参数**。在被调函数中可以改变原字符串的内容。

将字符串a复制到字符串b。

```
void main(void)
```

```
{ char a[ ]={"I am a teacher"};
```

```
char b[ ]={"You are a student"};
```

```
copy_string(a , b);
```

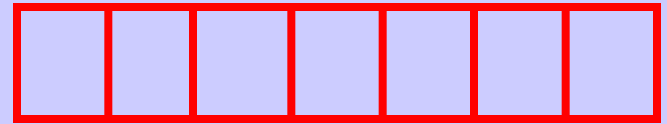
```
cout<<a<<endl;
```

```
cout<<b<<endl;
```

```
}
```

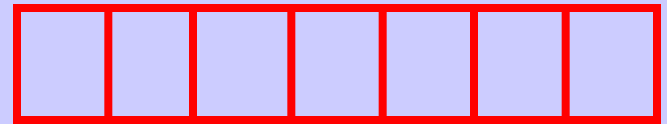
from

a



to

b



from与a一个地址, **to**与b一个地址

```
copy_string( char from[],char to[])
```

```
{ int i;
```

```
for (i=0; from[i]!='\0'; i++)
```

```
to[i]=from[i];
```

```
to[i]='\0';
```

```
}
```

将字符串a复制到字符串b。 `copy_string(char *from, char *to)`

```
{ for ( ; *from!='\0'; )
```

```
for( ; *from++=*to++; ) ;
```

```
*to='\0';
```

```
void main(void) }
```

```
{ char a[ ]={"I am a teacher"};
```

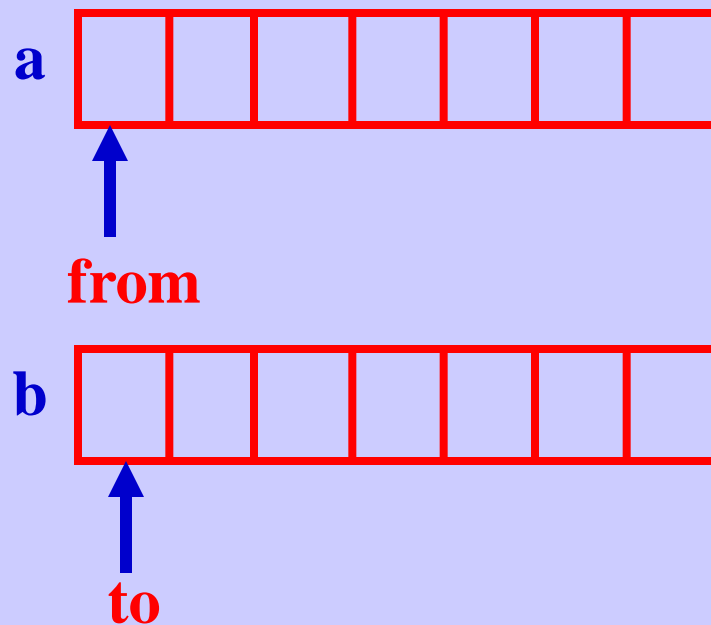
```
char b[ ]={"You are a student"};
```

```
copy_string(a,b);
```

```
cout<<a<<endl;
```

```
cout<<b<<endl;
```

```
}
```



也可以用字符指针来接受数组名

字符指针变量与字符数组

字符**数组**和字符**指针变量**都可以实现字符串的存储和运算，区别在于：

字符数组名是常量，定义时必须指明占用的空间大小。

字符指针变量是变量，里面存储的是字符型地址，可以整体赋值，**但字符串必须以 ‘\0’ 结尾。**

```
void fun(char *s)
```

```
{ int i, j;
```

```
for( i=j=0; s[i]!='\0'; i++)
```

```
    if(s[i]!='c')
```

```
        s[j++] = s[i];
```

```
    s[j] = '\0';
```

必须以\0结束

```
return;
```

```
}
```

当s[i]等于字符 'c'时，i前进，j不

动

输出： hane

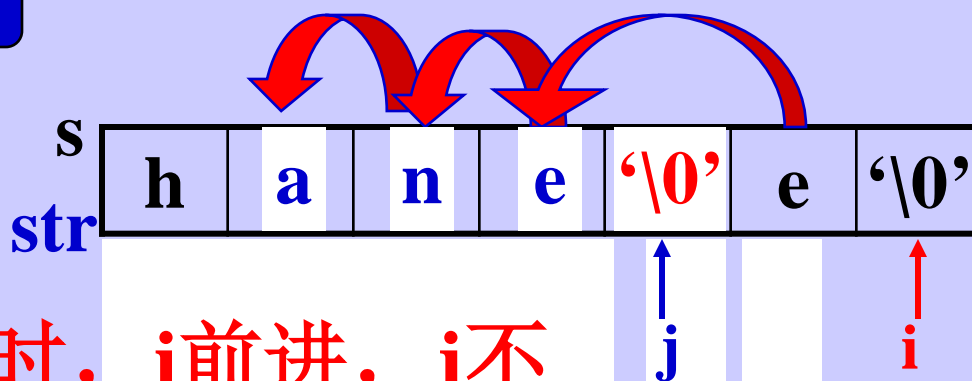
```
void main(void)
```

```
{ char str[80]= "hcance";
```

```
    fun(str);
```

```
    cout<<str<<endl;
```

```
}
```



```
void swap(char *s1, char *s2)
```

```
{ char t;
```

```
    t=*s1; *s1=*s2; *s2=t;
```

```
}
```

AD

BC

```
void main(void)
```

BB

```
{ char *s1="BD", *s2="BC", *s3="AB";
```

```
    if(strcmp(s1,s2)>0) swap(s1,s2);
```

```
    if(strcmp(s2,s3)>0) swap(s2,s3);
```

```
    if(strcmp(s1,s2)>0) swap(s1,s2);
```

```
    cout<<s1<<endl<<s2<<endl<<s3<<endl;
```

```
}
```

有一字符串，包含n个字符。写一函数，将此字符串中从第m个字符开始的全部字符复制成为另一个字符串。

```
void main(void)
```

```
{  char str1[100]={“I am a student”},str2[100];
```

```
    int m;
```

```
    cin>>m;
```

```
    fun(str1,str2,m);
```

```
    cout<<str2<<endl;
```

```
}
```

```
void fun(char *p1, char *p2, int m)
```

```
{  int n=strlen(p1);
```

原字符串长度

```
    if(n<m)
```

```
    { cout<<"No trans\n";*p2=0;  return;}
```

```
    for(int i=m, p1=p1+m-1; i<n;i++)
```

```
        *p2++=*p1++;
```

从第m个字符开始

```
    *p2='\0';
```

```
    return;
```

```
}
```


函数的指针和指向函数的指针变量

可以用指针变量指向变量、字符串、数组，也可以指向一个函数。

一个存放地址的指针变量空间可以存放数据的地址（整型、字符型），也可以存放数组、字符串的地址，还可以存放函数的地址。

函数在编译时被分配给一个入口地址。这个入口地址就称为函数的地址，也是函数的指针。像数组一样，C++语言规定，函数名就代表函数的入口地址

专门存放函数地址的**指针变量**称为**指向函数的指针变量**。

函数类型 (*指针变量名)(参数类型);

同时该函数具有两个整型形参

`int (*p)(int, int);`

且该函数的返回值为整型数

直接用函数名为指针变量赋值。

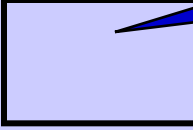
```
int max (int x, int y)
```

```
{ return x>y?x:y;
```

```
}
```

空间的内容只能放函数的地址

p



p=max;

这时，指针变量p中放的是max函数在内存中的入口地址。

函数名**max**代表函数在内存中的入口地址，**是一个常量**，不可被赋值。

而指向函数的指针变量 **p** 可以先后指向不同的**同种类型**的函数。但不可作加减运算。

int (*p)(int , int);

定义

p=max;

p=min;

赋值

如何用指向函数的指针变量调用函数？

```
int max(int x, int y)
{ return x>y?x:y; }
```

```
void main(void)
{ int a, b, c;
  cin>>a>>b;
  c=max(a,b);
  cout<<c<<endl;
}
```

给指针变量赋值

一般的调用方法

c>(*p)(a,b)

```
int max(int x, int y)
{ return x>y?x:y; }
```

```
void main(void)
{ int a, b, c, max(int,int);
  int (*p)(int, int );
  p=max ;
  cin>>a>>b;
  c=p(a,b);
  cout<<c<<endl;
}
```

定义指向函数的指针变量

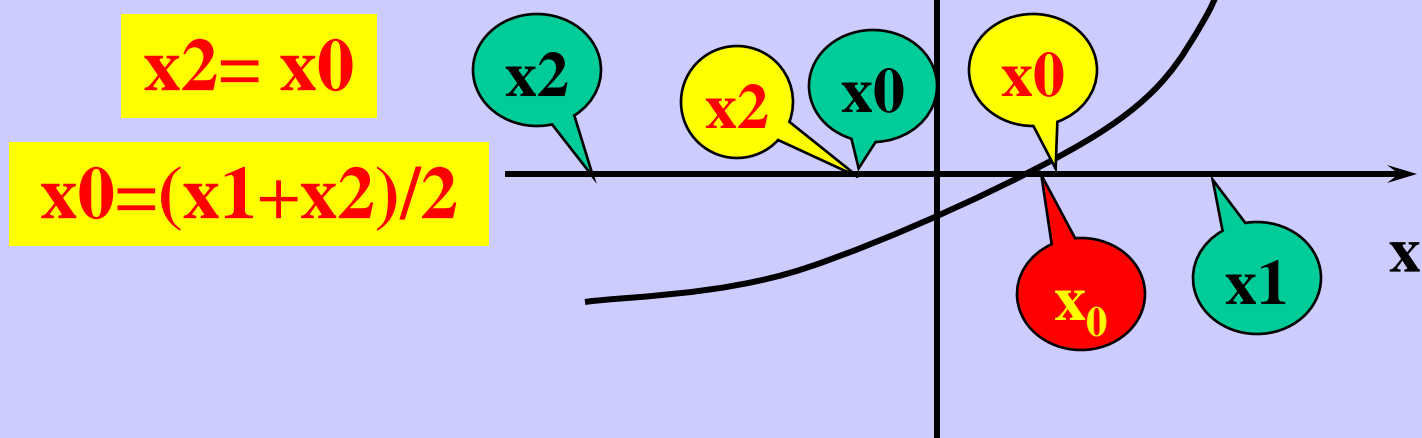
通过指针变量调用

实际上就是用p替换函数名

指向函数的指针变量作函数参数（实现通用函数）

用二分法求方程的解。 $f1(x)=x^2-3$

二分法求解方程



1、在 x 轴上取两点 x_1 和 x_2 , 要确保 x_1 与 x_2 之间有且只有方程唯一的解。

2、做 $x_0 = (x_1 + x_2) / 2$ 。

3、若 $|f(x_0)|$ 满足给定的精度, 则 x_0 即是方程的解, 否则, 若 $f(x_0) \cdot f(x_1) < 0$, 则方程的解应在 x_1 与 x_0 之间, 令 $x_2 = x_0$, 继续做2。同理, 若 $f(x_0) \cdot f(x_1) > 0$, 则方程的解应在 x_2 与 x_0 之间, 令 $x_1 = x_0$, 继续做2, 直至满足精度为止。

用二分法求方程的解。 $f1(x)=x^2-3$

```
float f1(float x)
```

```
{return x*x-3;}
```

已知x求f1(x)

```
void main(void)
```

```
{ float x1, x2, x0;
```

```
do
```

```
{ cout<<"Input two real number\n";
```

```
cin>>x1>>x2;
```

输入初值

```
}while (f1(x1)*f1(x2)>0);
```

判断x1与x2之间
是否有方程根

```
do
```

求中值

```
{ x0=(x1+x2)/2;
```

```
if ( (f1(x1)*f1(x0)) <0 )
```

```
x2=x0;
```

循环迭代

```
else x1=x0;
```

```
}while (fabs (f1(x0))>=1e-6);
```

```
cout<<x0<<endl;
```

循环结束条件

```
}
```

当求解方程 $f_2(x)=3x^2-5x-3$ 时，同样

```
float f2(float x)
```

```
{return 3*x*x-5*x-3;}
```

```
main( )
```

```
{ float x1, x2, x0;
```

```
do
```

```
{ cout<<"Input two real number\n";
```

```
cin>>x1>>x2;
```

```
}while (f2(x1)*f2(x2)>0);
```

```
do
```

```
{ x0=(x1+x2)/2;
```

```
if ( (f2(x1)*f2(x0)) <0 )
```

```
x2=x0;
```

```
else x1=x0;
```

```
}while (fabs (f2(x0))>=1e-6);
```

```
cout<<x0<<endl;
```

```
}
```

可以看到，虽然算法相同，仅是方程不同，两个程序不能通用。

可以用指向函数的指针变量，设计相同算法的通用函数。


```
float devide(float (*fun)(float ))
```

```
{ float x1, x2, x0;
```

```
do
```

```
{ cout<<"Input
```

```
cin>>x1>>x2;
```

```
}while ( fun(x1) * fun(x2) >0);
```

```
do
```

```
{ x0=(x1+x2)/2;
```

```
if ( fun(x1) * fun(x0) <0 )
```

```
x2=x0;
```

```
else x1=x0;
```

```
}while (fabs ( fun(x0) )>=1e-6);
```

```
return x0;
```

```
}
```

形参用指向函数的
指针变量fun
接受函数名

用fun调用函数，
实际调用的是实参函数

```
float f1(float x)
```

```
{return x*x-3;}
```

```
float f2(float x)
```

```
{return 3*x*x-5*x-3;}
```

```
main( )
```

```
{
```

```
float y1,y2;
```

```
y1=devide(f1);
```

```
y2=devide(f2);
```

```
cout<<y1<<"\t"<<y2;
```

```
}
```

调用函数，
实参为函数名f1

实参：实际的函数名（函数地址）

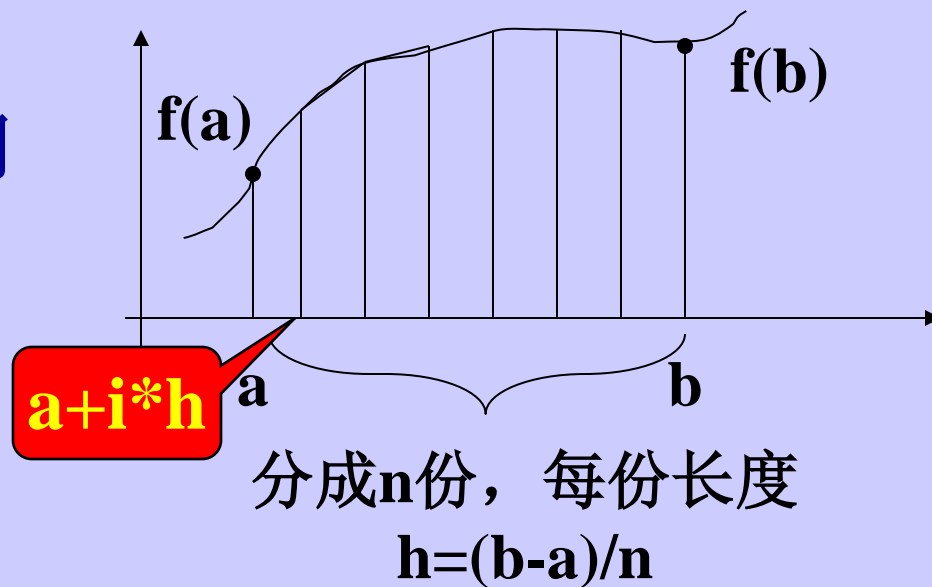
形参：指向函数的指针变量

与实参函数的类型完全一致（返回值、参数）

通用函数：所有的内部函数调用都用函数指针调用

梯形法求定积分的公式

积分结果为曲线与x轴之间部分的面积，也即为每份面积之和。



任意一份：高： h

上底： $f(a+i*h)$ 下底： $f(a+(i+1)*h)$

$$S = \sum [(上底 + 下底) * 高 / 2]$$

$$= \sum [(f(a+i*h) + f(a+(i+1)*h)) * h / 2]$$

其中 $i=0 \sim (n-1)$ $a+n*h=b$

$$S = \sum [(上底 + 下底) * 高 / 2]$$

$$= \sum [(f(a+i*h) + f(a+(i+1)*h)) * h / 2]$$

其中 $i=0 \sim (n-1)$ $a+n*h=b$

将上式展开，除 $f(a)$ ， $f(b)$ 外，每一底边都参与运算两次，所以有：

$$S = [(f(a) + f(b)) / 2 + \sum f(a+i*h)] * h \quad (i=1 \sim (n-1))$$

$$y = (f(a) + f(b)) / 2;$$

$$h = (b - a) / n;$$

$$\text{for}(i=1; i < n; i++)$$

$$y = y + f(a + i * h);$$

$$y = y * h;$$

初值

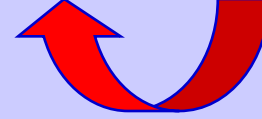
计算累加和

可利用这一公式，计算所有函数的定积分，也即做成通用公式，具体的函数由指向函数的指针变量来代替，在主调函数中为这个指针变量赋值。

```
float f1(float x)
```

```
{ return x*x*x+2*x*x+9; }
```

pf=f1



```
float jifen(float (*pf)(float ), float a, float b)
```

```
{ y=(pf(a)+pf(b))/2;
```

```
h=(b-a)/2;
```

```
for(i=1;i<n;i++)
```

```
y=y+pf(a+i*h);
```

```
y=y*h;
```

```
return y;
```

```
void main(void)
```

```
{ float y;
```

```
y=jifen(f1, 1.0, 3.0);
```

```
cout<<y<<endl;
```

```
}
```

```
y2=jifen(f2, 2.0, 5.0);
```

返回指针值的函数

被调函数返回的不是一个数据，而是一个地址。所以函数的类型为指针类型。

类型标识符 *函数名(参数表)

指出返回是什么类型的地址

`int *max(x, y)`

```
int *max (int *x, int *y)
```

```
{ int *pt;
```

```
if (*x>*y)
```

```
pt=x;
```

```
else pt=y;
```

```
return pt;
```

```
}  
返回类型是指针
```

```
void main(void)
```

```
{ int a, b , *p;
```

```
cin>>a>>b;
```

```
p=max(&a,&b);
```

```
cout<<*p<<endl;
```

```
}
```

用指针类型接收



输出： 4

该指针所指向的空间是在主调函数中开辟的。

```
char *strc(char *str1, char *str2)
```

```
{ char *p;
```

```
for(p=str1;*p!='\0'; p++);
```

p指向str1的最后

```
do { *p++=*str2++; }while (*str2!='\0');
```

```
*p='\0';
```

最后 '\0' 结

str2向p赋值

```
return (str1); 束
```

```
}
```

```
void main(void)
```

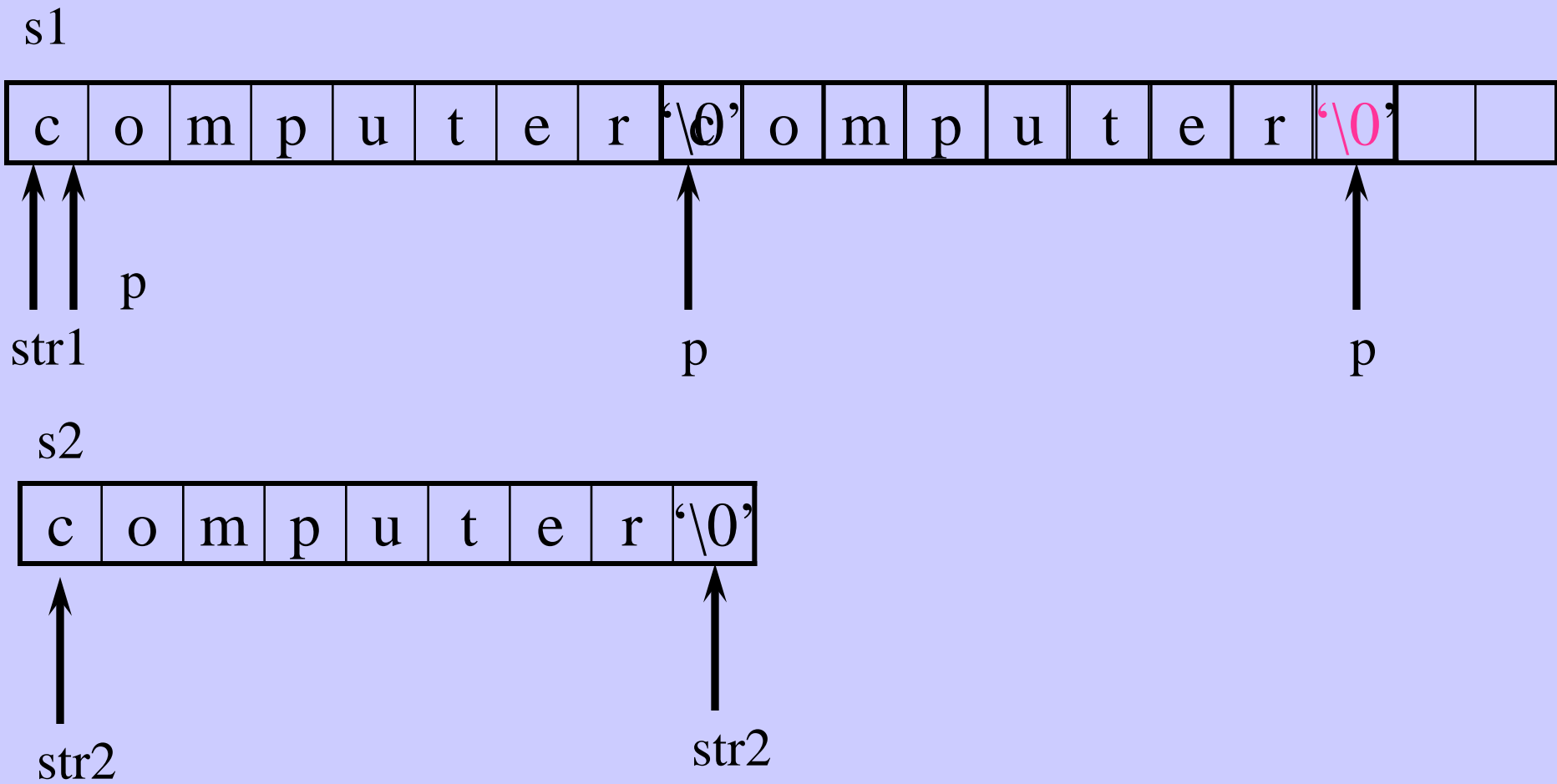
```
{char s1[80]="computer", s2[ ]="language", *pt;
```

```
pt=strc(s1, s2);
```

```
cout<<pt<<endl;
```

computerlanguage

```
}
```

已知函数 `int *f(int *)`，有以下说明语句：

`int *p, *s, a;`

函数正确的调用形式是：

A) `a=*f(s)` B) `*p=f(s)` C) `s=f(*p)` D) `p=f(&a)`

指针数组和指向指针的指针

指针数组的概念

一个数组，其元素均为指针类型的数据，称为指针数组。也就是说，指针数组中的每一个元素都是指针变量，可以放地址。

类型标识 *数组名[数组长度说明]

```
int *p[4];
```

p为数组名，内有四个元素，每个元素可以放一个int型数据的地址

p	
p[0]	地址
p[1]	地址
p[2]	地址
p[3]	地址

```
int (*p)[4];
```

p为指向有四个int型元素的一维数组的行指针

```
void main(void)
```

```
{    float a[]={100,200,300,400,500};
```

```
    float *p[]={&a[0],&a[1],&a[2],&a[3],&a[4]};
```

```
    int i;
```

```
    for(i=0;i<5;i++)
```

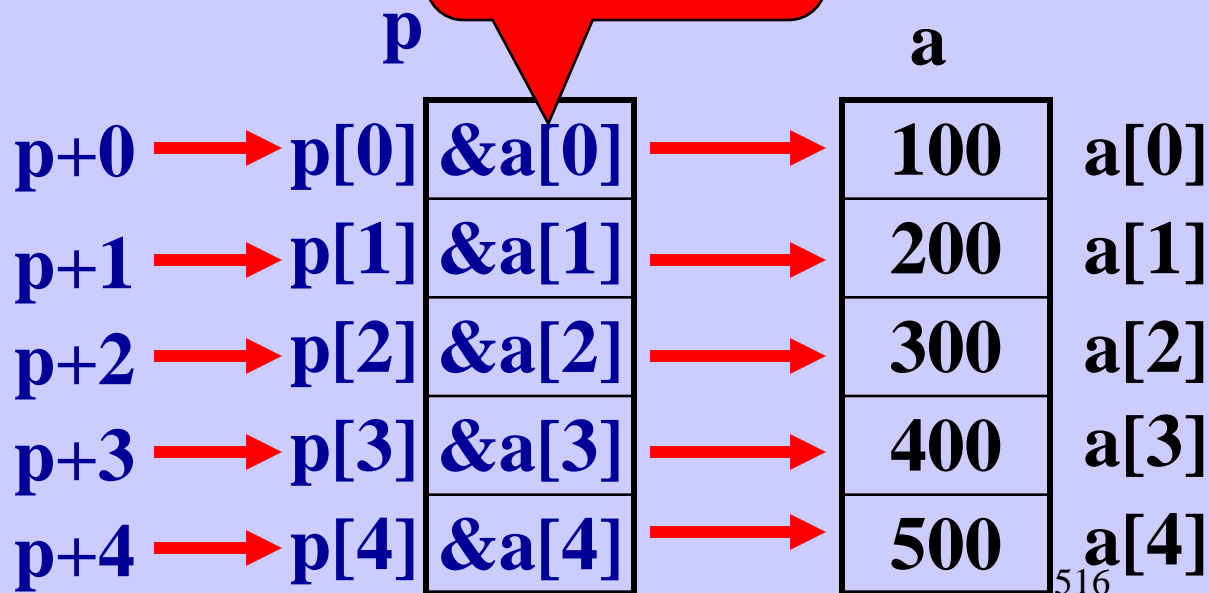
```
        cout<<*p[i]<<'\t';
```

```
    cout<<endl;
```

```
}
```

```
*p[i]=*(*p+i) )
```

**p数组元素
内放地址**



```
void main(void)
```

```
{ int a[12]={1,2,3,...11,12};
```

```
    int *p[4], i;
```

```
    for(i=0; i<4;i++)
```

```
        p[i]=&a[i*3];
```

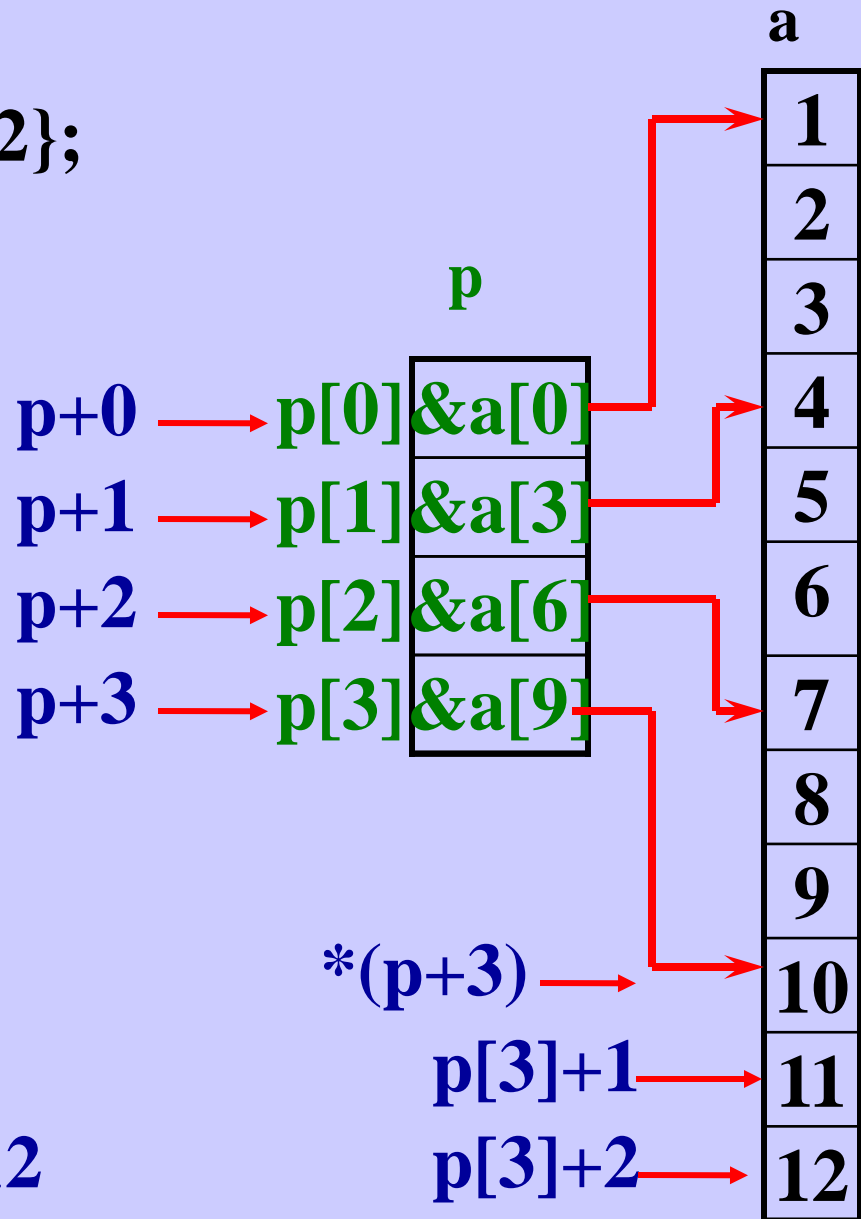
```
    cout<<p[3][2]<<endl;
```

```
}
```

```
    p[3][2]=*(*p+3)+2)
```

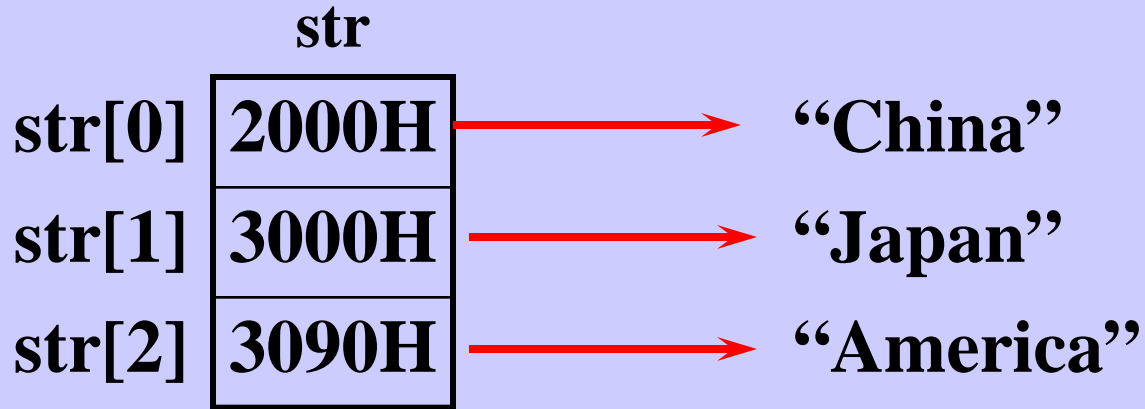
```
    =*(p[3]+2)
```

12



常用字符指针数组，数组中的元素指向字符串首地址，这样比字符数组节省空间。

```
char *str[ ]={"China", "Japan", "America"};
```



str[1][4] $*(*(\text{str}+1)+4)$ **n**

str[2] $*(*(\text{str}+2)+0)$ **A*

****str $*(*(\text{str}+0)+0)$ **C****

将若干字符串按字母顺序（由小到大）输出。

```
void main(void)  
  
{ void sort( );  
  
void print( );  
  
char *alpha[ ]={"Follow me", "Basic", "Great Wall",  
"FORTRAN", "Computer design"};  
  
int n=5;  
  
sort (alpha, n);  
  
print(alpha, n);  
  
}
```

```
void sort(char *alpha[ ], int n)
```

```
{ char *temp;
```

```
  int i, j, k;
```

```
  for(i=0;i<n-1;i++)
```

```
  { k=i;
```

```
    for(j=i+1; j<n; j++)
```

```
      if(strcmp(alpha[k], alpha[j])>0)
```

```
        k=j;
```

```
  if (k!=i)
```

```
  { temp=alpha[i];
```

```
    alpha[i]=alpha[k];
```

```
    alpha[k]=temp;
```

```
  }
```

```
}
```

```
}
```

```
void print(char *alpha[ ], int n)
```

```
{ int i;
```

```
  for(i=0;i<n;i++)
```

```
    cout<<alpha[i]<<endl;
```

```
}
```

选择法排序

交换地址

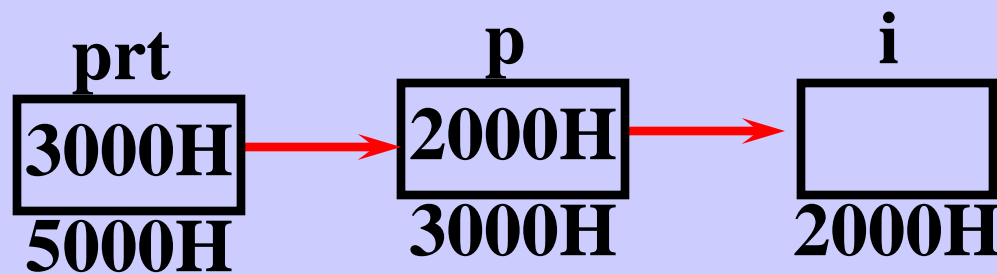
alpha[0]
alpha[1]
alpha[2]
alpha[3]
alpha[4]

“Follow me”
“Basic”
“Great Wall”
“FORTRAN”
“Computer design”

指向指针的指针变量

```
int i,*p;
```

```
p=&i;
```



同样，`p`也有地址，可以再引用一个指针变量指向它。

```
prt=&p; p=&i
```

```
int i, *p, **prt;
```

称`prt`为指向指针的指针变量。其基类型是指向整型数据的指针变量，而非整型数据。

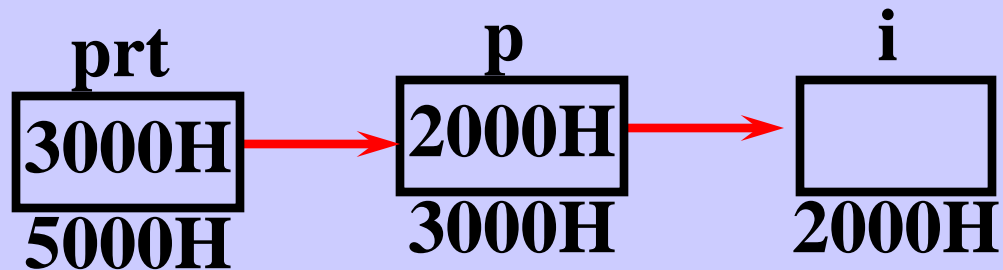
p=&i; prt=&
p;

*p=i;

**prt=i;

prt=&i; prt=p;

非法,基类
型不符



```
void main(void)
```

```
{ a[5]={1,3,5,7,9};
```

```
    int *num[5]={&a[0],&a[1],&a[2],&a[3],&a[4]};
```

```
    int **p, i;
```

```
    p=num;
```

```
    p=&num[0]
```

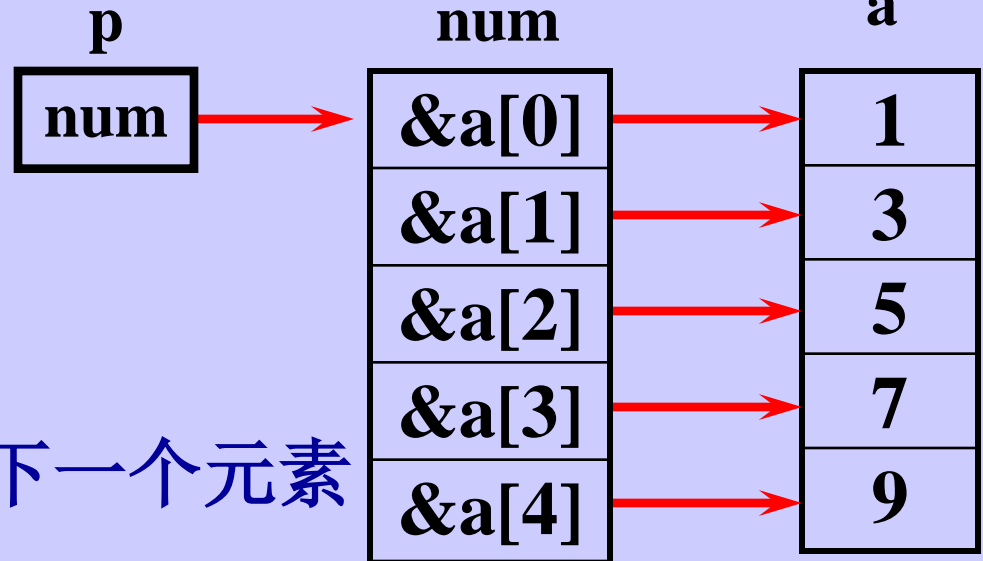
```
    for(i=0;i<5;i++)
```

```
    { cout<<**p<<'\t';
```

```
        p++;
```

```
    }
```

```
} p=p+1;指向num数组下一个元素
```



1 3 5 7 9

```

void main(void)

{char *alpha[ ]={"Follow me", "Basic", "Great Wall",
                 "FORTRAN", "Computer design"};

char **p;

int i;

for(i=0; i<5; i++)
{
    p=alpha+i;
    cout<<*p<<endl;
}

```

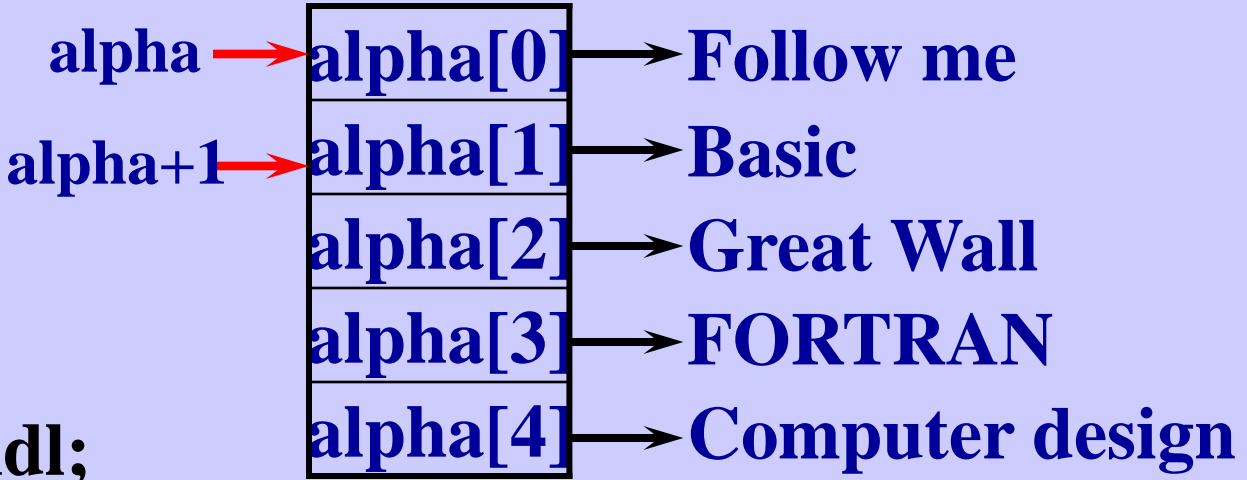


Diagram illustrating the array `alpha` and its elements:

Index	String
<code>alpha[0]</code>	Follow me
<code>alpha[1]</code>	Basic
<code>alpha[2]</code>	Great Wall
<code>alpha[3]</code>	FORTRAN
<code>alpha[4]</code>	Computer design

Execution flow for the loop:

Iteration	<code>i</code>	<code>p</code>
1	0	<code>p=alpha</code>
2	1	<code>p=alpha+1</code>

```
void main(void)
```

```
{ char *n[4]={“China”, “Japan”, “England”, “Germany”};
```

```
char **p;
```

```
int i;
```

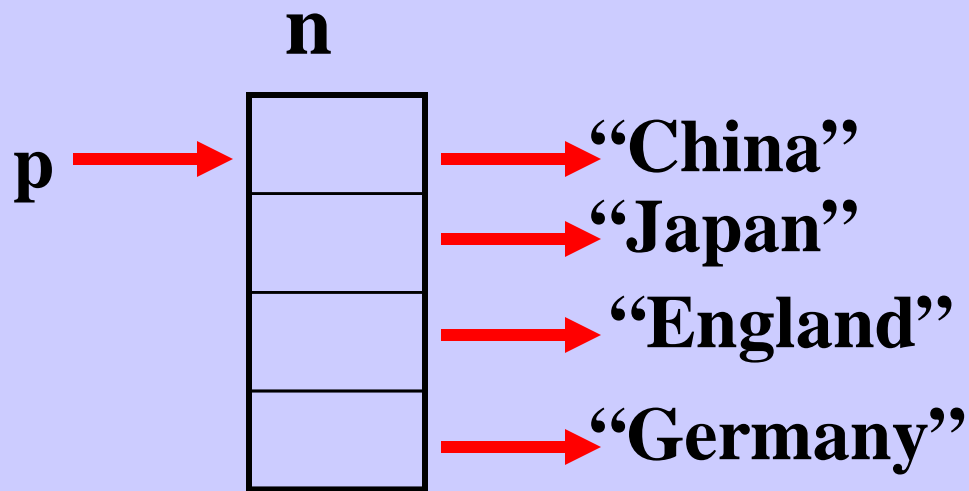
```
p=n;
```

```
for(i=0; i<4;i++,p++)
```

```
cout<<(char) (*(*p+2)+1)<<endl;
```

```
}
```

$$*(*p+2)+1=*(p[0]+2)+1=p[0][2]+1$$



输出: j

q

h

S

以下程序的输出结果是：

```
char *alpha[6]={“ABCD”,“EFGH”,“IJKL”,“MNOP”,  
                “QRST”,“UVWX”};
```

```
char **p;
```

```
main( )
```

```
{ int i;
```

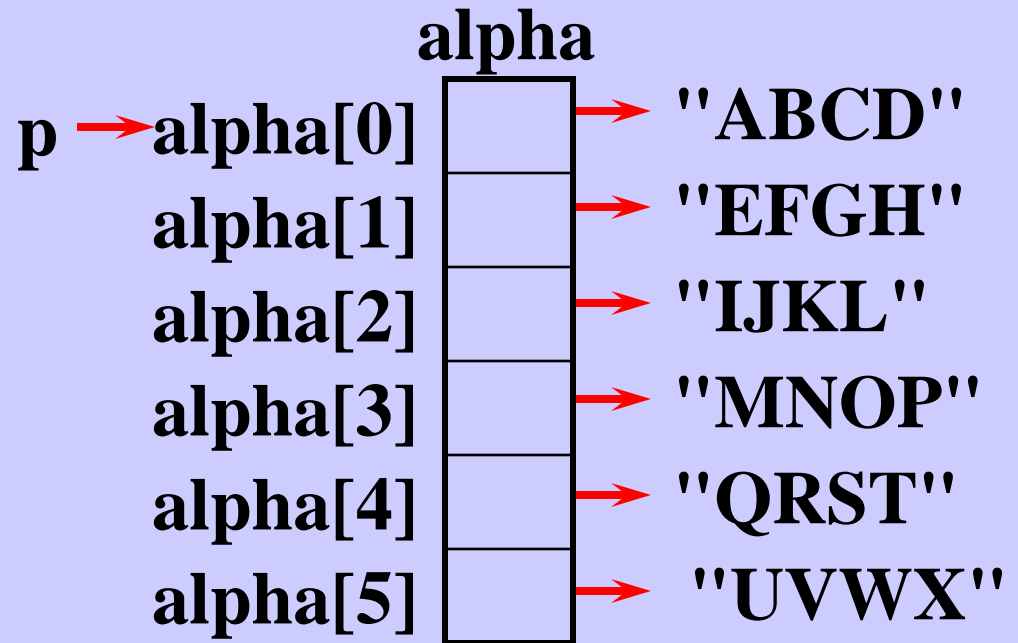
```
  p=alpha;
```

```
  for(i=0;i<4;i++)
```

```
    cout<<*(p[i]);
```

```
  cout<<endl;
```

```
}
```



$*(p[i])=*(*(p+i))=*(*(p+i)+0)$

输出： AEIM

以下程序的输出结果是：

```
char *alpha[6]={“ABCD”,“EFGH”,“IJKL”,“MNOP”,  
                “QRST”,“UVWX”};
```

```
char **p;
```

```
main( )
```

```
{ int i;
```

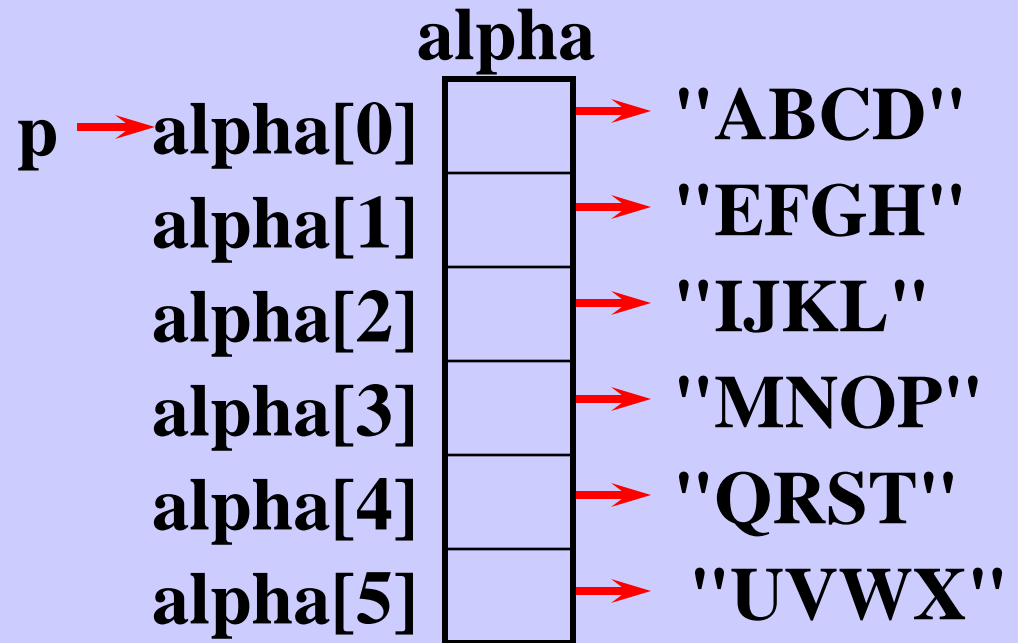
```
  p=alpha;
```

```
  for(i=0;i<4;i++)
```

```
    cout<<(*p)[i];
```

```
    cout<<endl;
```

```
}
```



$(*p)[i] = *(p+0)[i] = p[0][i]$

输出： ABCD

若有以下定义，则下列哪种表示与之等价。

char s[3][5]={“aaaa”,“bbbb”,“cccc”};

A) char **s1= {“aaaa”,“bbbb”,“cccc”};

B) char *s2[3]= {“aaaa”,“bbbb”,“cccc”};

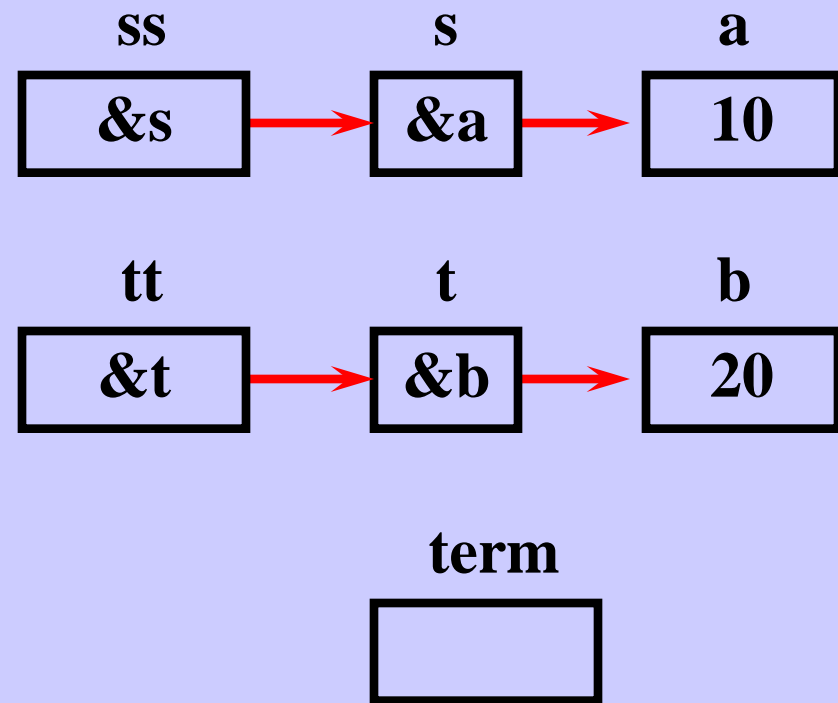
C) char s3[][3]= {“aaaa”,“bbbb”,“cccc”};

D) char s4[][4]= {“aaaa”,“bbbb”,“cccc”};

以下程序调用函数swap_p将指针s和t所指单元(a和b)中的内容交换，请填空。

```
void swap_p (int **ss, int **tt)
{   int term;
    term=**ss;
    **ss=**tt;
    **tt=term;
}

main( )
{   int a=10, b=20, *s,*t;
    s=&a; t=&b;
    swap_p(&s,&t);
    printf(“%d  %d”,a,b);
}
```



假设有说明：

```
char *argv[ ]={"hello", "nanjing", "jiangsu"};
```

```
char **pargv=argv;
```

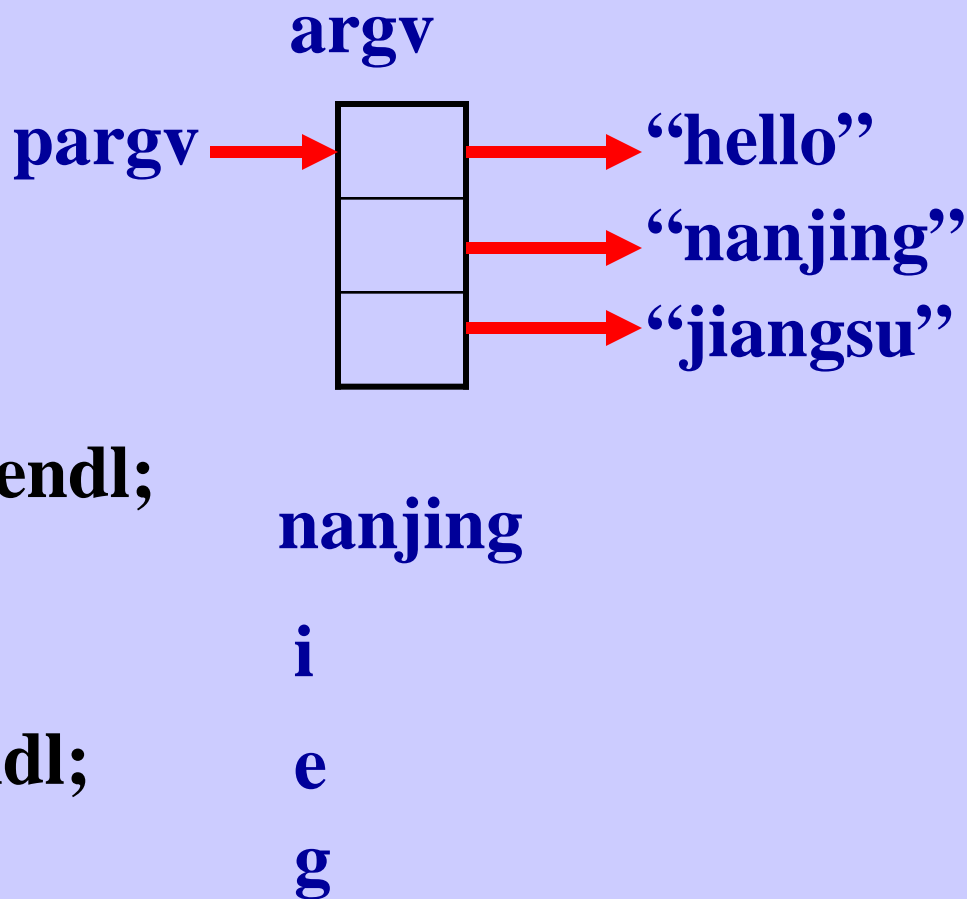
以下语句的输出结果如何？

```
cout<<*(pargv+1)<<endl;
```

```
cout<<(char)(**pargv+1)<<endl;
```

```
cout<<*(*pargv+1)<<endl;
```

```
cout<<*(*pargv+2)+4)<<endl;
```



```
void main()
```

```
{  char *s[]={“1995”,“1996”,“1997”,“1998”};
```

```
  char **sp[ ]={s+3,s+2,s+1,s};
```

```
  char ss[5];
```

```
  ss[0]**sp[0];
```

```
  ss[1]=*(*sp[1]+1);
```

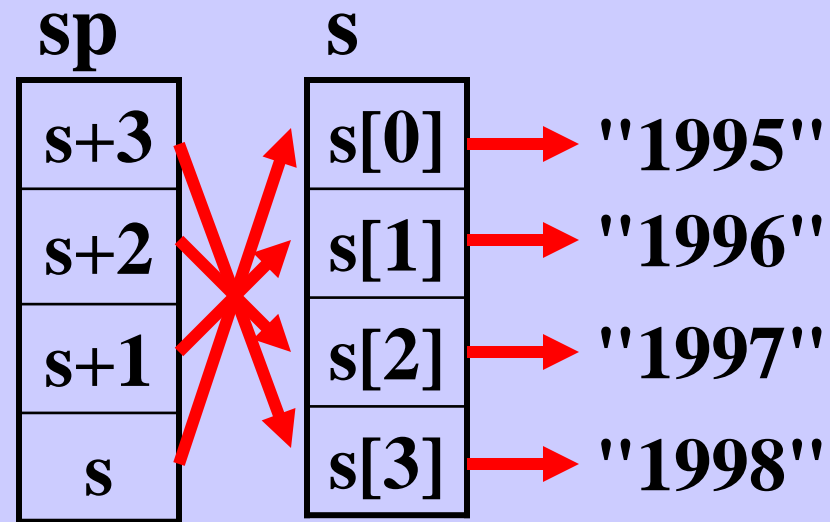
```
  ss[2]=*(*sp[2]+2);
```

```
  ss[3]=*sp[3][3]+6;
```

```
  ss[4]='\0';
```

```
  cout<<ss<<endl;
```

```
}
```



SS

1	9	9	7	0
---	---	---	---	---

总结:

通过行指针引用二维数组元素

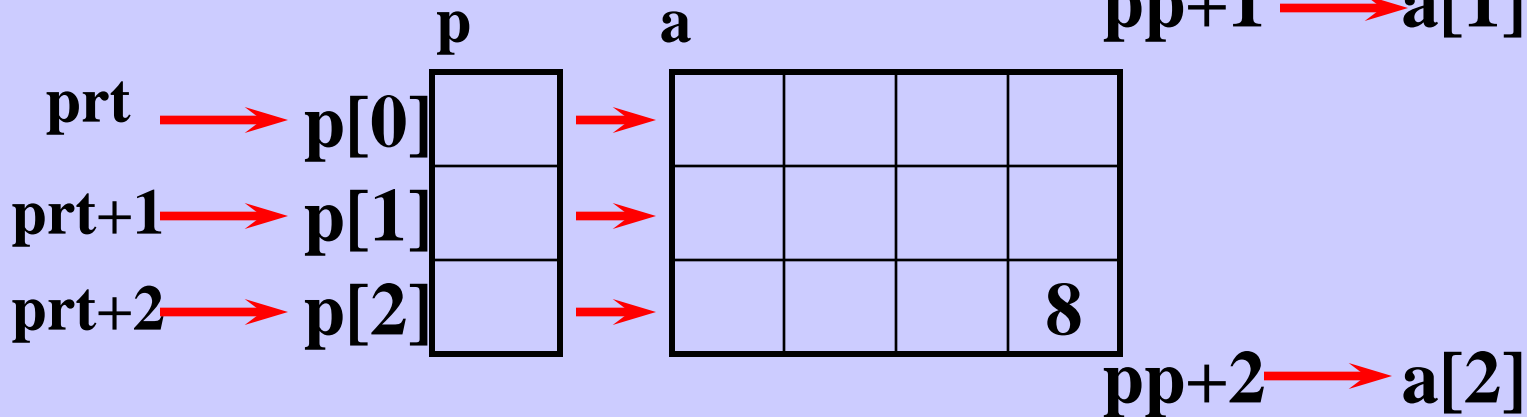
```
int a[3][4], *p[3], (*pp)[4], **prt;
```

```
for(i=0;i<3;i++)
```

```
    p[i]=a[i];
```

```
pp=a;
```

```
prt=p;
```



`a[2][3]` `*(*(pp+2)+3)`

`a[2][3]` `*(*(p+2)+3)`

`a[2][3]` `*(*(a+2)+3)`

`a[2][3]` `*(*(prt+2)+3)`

`a[2][3]` `pp[2][3]` `p[2][3]` `prt[2][3]`

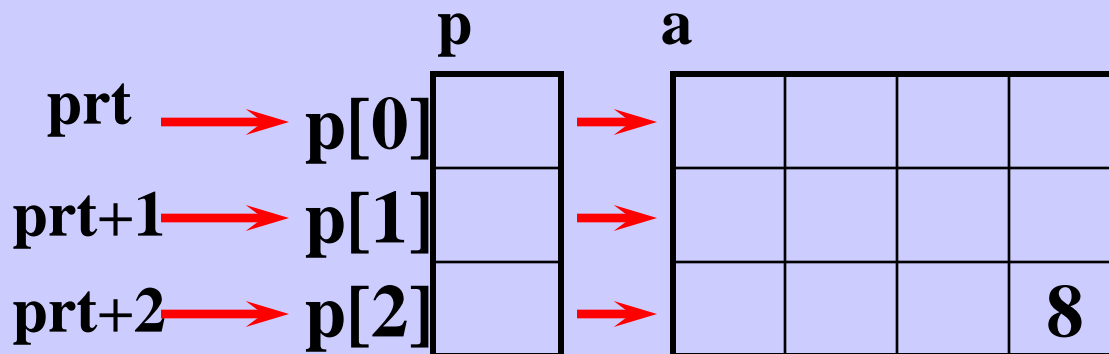
```
int  a[3][4], *p[3], (*pp)[4], **prt;
```

```
for(i=0;i<3;i++)
```

```
    p[i]=a[i];
```

```
pp=a;
```

```
prt=p;
```



a: 二维数组名, **常量**

p: 指针数组名, **常量**

pp: 指向具有四个元素的一维数组的指针**变量**

prt : 指向指针的**指针变量**

指针数组作main()函数的形参

程序是由main()函数处向下执行的。main函数也可以带参数。

其它函数由main函数调用的，即在程序中调用的，但main函数却是由DOS系统调用的，所以main函数实参的值是在DOS命令行中给出的，是随着文件的运行命令一起给出的。

可执行文件名 实参1 实参2 实参n

可执行文件S9_16.EXE

S9_16 CHINA JAPAN AMERICAN<CR>

三个形参

main函数形参的形式:

main(int argc, char * argv[])

main(int argc, char **argv)

argc为命令行中参数的个数（包括文件名）；

argv为指向命令行中参数（字符串）的指针数组。

S9_16 CHINA JAPAN AMERICAN<CR>

文件名

实参1

实参2

实参3

argc=4

argv

argv

argv[0]

→ “S9_16.EXE”

argv[1]

→ “CHINA”

argv[2]

→ “JAPAN”

argv[3]

→ “AMERICAN”

S9_16 CHINA JAPAN AMERICAN<CR>

文件名

实参1

实参2

实参3

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

S9_16.EXE

```
{ while (argc>1)
```

argc=4

CHINA

```
{ cout<<*argv<<endl;
```

JAPAN

```
++argv;
```

argv

argv

argv[0]

→ “S9_16.EXE”

argv[1]

→ “CHINA”

argv[2]

→ “JAPAN”

argv[3]

→ “AMERICAN”

```
--argc;
```

```
}
```

```
}
```


B9_1 Beijing China<CR>

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

```
{ while (argc-- >1)
```

Beijing

```
    cout<< *(++argv)<<endl;
```

China

```
}
```

argc=3

argv

argv[0]	→ “B9_1.EXE”
argv[1]	→ “Beijing”
argv[2]	→ “China”

argc=3 / 2 Beijing

argc=2 / 1 China

argc=1

小结

1、指针变量可以有空值，即指针变量不指向任何地址。

```
int *p;    #include "iostream.h"    #define NULL 0  
p=0;      int *p; p=NULL;
```

2、两指针可以相减，不可相加。若要进行相减运算，则两指针必须指向同一数组，相减结果为相距的数组元素个数

```
int  a[10],*p1,*p2;  
p2-p1 : 9  
p1=a;  p2=a+9;
```

3、指向同一数组的两个指针变量可以比较大小： $p2 > p1$

在内存动态分配存储空间

在定义变量或数组的同时即在内存为其开辟了指定的固定空间。

```
int n, a[10];
```

```
char str[100];
```

一经定义，即为固定地址的空间，在内存不能被别的变量所占用。

在程序内我们有时需要根据实际需要开辟空间，如输入学生成绩，但每个班的学生人数不同，一般将人数定得很大，这样占用内存。

```
#define N 100
```

```
.....
```

```
float score[N][5];
```

```
cin>>n;
```

```
for(int i=0;i<n;i++)
```

```
    for(j=0;j<5;j++)
```

```
        cin>>score[i][j];
```

```
.....
```

无论班级中有多少个学生，程序均在内存中开辟 100×5 个实型数空间存放学生成绩，造成内存空间的浪费。

如何根据需要在程序的运行过程中**动态分配**存储内存空间？

```
int n;
```

```
cin>>n;
```

```
float score[n][5];
```

错误！数组的维数
必须是常量

利用 **new** 运算符可以在程序中动态开辟内存空间。

```
new 数据类型[单位数];
```

```
new int[4];
```

在内存中开辟了4个**int**型的数据空间，即**16**个字节

new int;

在内存中开辟出四个字节的空间

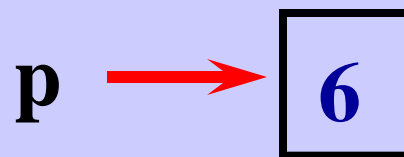
new 相当于一个函数，在内存开辟完空间后，返回这个空间的首地址，这时，**这个地址必须用一个指针保存下来**，才不会丢失。

int *p;

p=new int;



***p=6;**



new开辟的空间

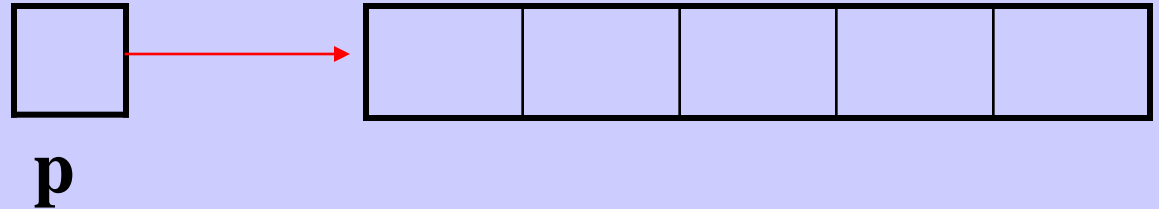
可以用***p**对这个空间进行运算。

同样，利用new运算符也可以开辟连续的多个空间(数组)。

```
int n,* p;
```

```
cin>>n;
```

```
p=new int[n];
```



`p`指向新开辟空间的首地址。

```
for(int i=0;i<n;i++)
```

```
    cin>>p[i];
```

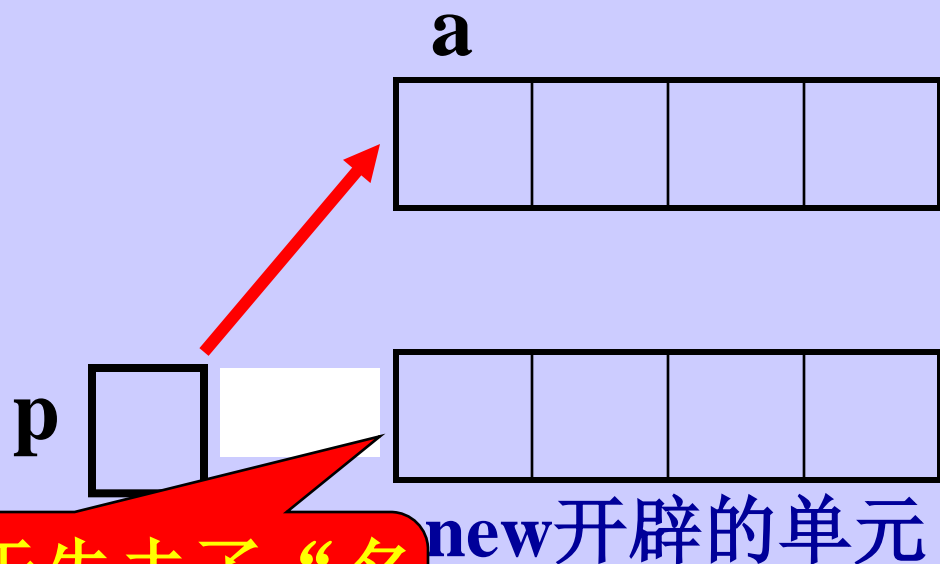
可以用`p[i]`的形式来引用新开辟的内存单元。

注意：用new开辟的内存单元没有名字，指向其**首地址的指针**是引用其的唯一途径，若指针变量重新赋值，则用new开辟的内存单元就在内存中“丢失”了，别的程序也不能占用这段单元，直到重新开机为止。

```
int * p, a[4];
```

```
p=new int[4];
```

```
p=a;
```



该段内存由于失去了“名字”，再也无法引用

用 new 运算符分配的空间，不能在分配空间时进行初始化。

同样，用new开辟的内存单元如果程序不“主动”收回，那么这段空间就一直存在，直到重新开机为止。

delete运算符用来将动态分配到的内存空间归还给系统，使用格式为：

```
delete p;
```

```
int *point;
```

```
point=new int;
```

```
.....
```

```
delete point;
```

注意：在此期间，point指针不能重新赋值，只有用new开辟的空间才能用delete收回。

delete也可以收回用new开辟的连续的空间。

```
int *point;
```

```
cin>>n;
```

```
point=new int[n];
```

```
.....
```

```
delete [ ]point;
```

当内存中没有足够的空间给予分配时，new运算符返回空指针NULL（0）。

以下程序求两个数的大者，请填空。

```
void main(void )
```

```
{ int *p1, *p2;
```

```
    p1= new int ;
```

```
    p2= new int ;
```

```
    cin>> *p1>>*p2 ;
```

```
    if (*p2>*p1)    *p1=*p2;
```

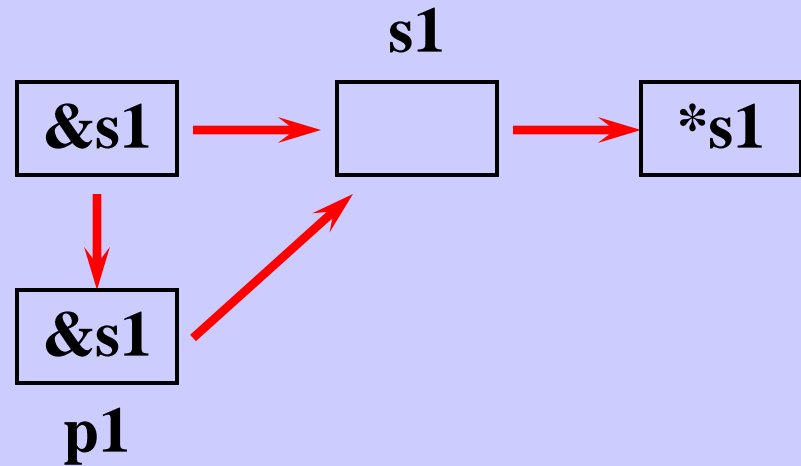
```
    delete p2;
```

```
    cout<<"max="<< *p1 <<endl;
```

```

main( )
{  int *s1, *s2;
    sub1(&s1,&s2);  sub2(&s1,&s2);
    cout<<*s1<<'\t'<<*s2<<endl;
    sub3(s1, s2);  sub4(s1,s2);
    cout<<*s1<<'\t'<<*s2<<endl;
}

```



```

sub1( int  **p1, int  **p2)
{ *p1=new int ; *p2=new int ; }
sub2(int  **p1, int  **p2)
{ **p1=10; **p2=20; **p1=**p2; }
sub3(int  *p1, int  *p2)
{ p1=new int ; p2=new int ; }
sub4( int  *p1, int  *p2)
{ *p1=1; *p2=2; *p2=*p1; }

```

20	20
1	1

引用

对变量起另外一个名字（外号），这个名字称为该变量的引用。

<类型> &<引用变量名> = <原变量名>;

其中**原变量名**必须是一个已定义过的变量。如：

```
int max ;
```

```
int &refmax=max;
```

refmax并没有重新在内存中开辟单元，只是**引用**
max的单元。max与refmax**在内存中占用同一地址**，
即同一地址两个名字。

```
int max ;
```

```
int &refmax=max;
```

```
max=5 ;
```

```
refmax=10;
```

```
refmax=max+refmax;
```



max与refmax同一地址

对引用类型的变量，说明以下几点：

1、引用在定义的时候要初始化。

`int &refmax;`

错误，没有具体的引用对象

`int &refmax=max;`

max是已定义过的变量

2、对引用的操作就是对被引用的变量的操作。

3、引用类型变量的初始化值不能是一个常数。

如：`int &ref1 = 5;` // 是错误的。

`int &ref=i;`

4、引用同变量一样有地址，可以对其地址进行操作，即将其地址赋给一指针。

```
int a, *p;
```

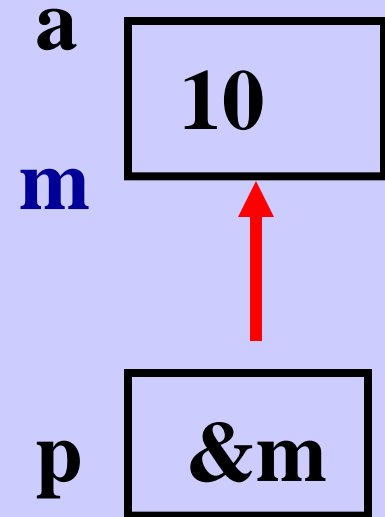
&是变量的引用

```
int &m=a;
```

```
p=&m;
```

&是变量的地址

```
*p=10;
```



5、可以用动态分配的内存空间来初始化一个引用变量。

```
float &reff = * new float ; //用new开辟一个空间，取一个别名reff
```

```
reff= 200;    //给空间赋值
```

```
cout << reff ; //输出200
```

```
delete &reff; //收回这个空间
```

这个空间只有别名，但程序可以引用到。

```
float *p, a;
```

```
p=new float;
```

```
float a=* new float;
```

错误！

指针与引用的区别：

- 1、指针是通过地址**间接**访问某个变量，而引用是通过别名**直接**访问某个变量。
- 2、引用必须初始化，而**一旦被初始化后不得再作为其它变量的别名。**

当&a的前面有**类型符**时
（如int &a），它必然
是对引用的声明；如果前面
无类型符（如cout<<&a），
则是取变量的地址。

```
int m=10;
```

```
int &y=10;          int &z;
```

```
float &t=&m;   int &x=m;
```

以下的声明是非法的

1、企图建立数组的引用 **int & a[9];**

2、企图建立指向引用的指针 **int & *p;**

3、企图建立引用的引用 **int & &px;**

对常量（用**const**声明）的引用

```
void main(void)  
{  
    const int &r=8; //说明r为常量，不可赋值  
    cout<<"r="<<r<<endl;  
    // r+=15;           //r为常量，不可作赋值运算  
    cout<<"r="<<r<<endl;  
}
```

引用与函数

引用的用途主要是用来作函数的参数或函数的返回值。

引用作函数的形参，实际上是在被调函数中对实参变量进行操作。

```
void change(int &x, int &y)//x,y是实参a,b的别名
```

```
{ int t;
```

```
    t=x; x=y; y=z;
```

```
}
```

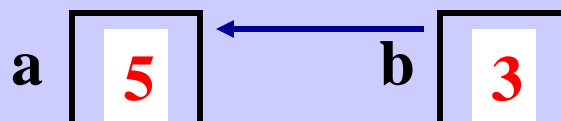
```
void main(void)
```

```
{ int a=3,b=5;
```

```
    change(a,b); //实参为变量
```

```
    cout<<a<<'\t'<<b<<endl;
```

```
}
```



输出： 5 3

引用作为形参，实参是变量而不是地址，这与指针变量作形参不一样。

形参为整型引用

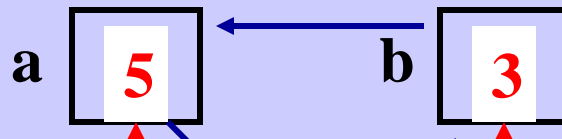
```
void change(int &x, int &y)
{   int  t;
    t=x; x=y; y=z;
}

void main(void)
{   int  a=3,b=5;
    change(a,b); //实参为变量
    cout<<a<<'\t'<<b<<endl;
}
```

形参为指针变量

```
void change(int *x, int *y)
{   int  t;
    t=*x; *x=*y; *y=z;
}

void main(void)
{   int  a=3,b=5;
    change(&a,&b); //实参为地址
    cout<<a<<'\t'<<b<<endl;
}
```



```
void dd(int &x, int &y, int z)
```

```
{  x=x+z;      x=8      x=13
```

```
   y=y-x;      y=-4     y=-17
```

```
   z=10;       z=10     z=10
```

```
   cout<<"(2)"<<x<<"\t"<<y<<"\t"<<z<<endl;
```

```
}
```

```
void main(void)      (2) 8    -4    10
```

```
{  int  a=3,b=4,c=5;  (2) 13   -17   10
```

```
   for(int i=0;i<2;i++) (1) 13   -17   5
```

```
       dd(a,b,c);
```

```
   cout<<"(1)"<<a<<"\t"<<b<<"\t"<<c<<endl;
```

```
}
```



```
void f1( int *px)  {  *px+=10;}
```

```
void f2(int &xx)  {  xx+=10;}
```

```
void main(void)
```

```
{    int x=0;
```

x=0

```
    cout<<"x="<<x<<endl;
```

x=10

```
    f1(&x);
```

x=20

```
    cout<<"x="<<x<<endl;
```

```
    f2(x);
```

```
    cout<<"x="<<x<<endl;
```

```
}
```

函数的返回值为引用类型

可以把函数定义为引用类型，这时函数的返回值即为某一变量的引用（别名），因此，它相当于返回了一个变量，所以可对其返回值进行赋值操作。这一点类同于函数的返回值为指针类型。

```
int a=4;
```

```
int &f(int x)
```

函数返回a的引用，即a的别名

```
{ a=a+x;
```

```
    return a;
```

```
}
```

```
void main(void)
```

```
{ int t=5;
```

输出 9 (a=9)

```
    cout<<f(t)<<endl;
```

```
    f(t)=20;
```

先调用，再赋值 a=20

```
    cout<<f(t)<<endl;
```

输出25 (a=25)

```
    t=f(t);
```

先调用，再赋值 t=30

```
    cout<<f(t)<<endl; }
```

输出60 (a=60)

一个函数返回引用类型，**必须返回某个类型的变量。**

语句：`getdata()=8;`

就相当于 `int &temp=8;`

`temp=8 ;`

注意：由于函数调用返回的引用类型是在函数运行结束后产生的，所以**函数不能返回自动变量和形参。**

返回的变量的引用，**这个变量必须是全局变量或静态局部变量，即存储在静态区中的变量。**

我们都知道，函数作为一种程序实体，它有名字、类型、地址和存储空间，一般说来函数不能作为左值（即函数不能放在赋值号左边）。但如果将函数定义为返回引用类型，因为返回的是一个变量的别名，就可以将函数放在左边，即给这个变量赋值。

```
int &f(int &x)
{  static int t=2;
   t=x++;
   return t;
}

void main(void)
{  int a=3;
   cout<<f(a)<<endl;
   f(a)=20;
   a=a+5;
   cout<<f(a)<<endl;
   a=f(a);
   cout<<f(a)<<endl;
}
```

输出 3 a=4 t=3
t=20 a=5
a=10
输出 10 a=11
a=11
输出 11 a=12

const类型变量

当用const限制说明标识符时，表示所说明的数据类型为常量类型。可分为const型常量和const型指针。

可用const限制定义标识符量，如：

```
const int MaxLine = 1000;
```

```
const float Pi = 3.1415926
```

用const定义的标识符常量时，一定要对其初始化。在说明时进行初始化是对这种常量置值的唯一方法，不能用赋值运算符对这种常量进行赋值。如：

```
MaxLine = 35;
```

const 型指针

1)禁写指针

声明语句格式为： 数据类型 * const 指针变量名

如： int r=6;

```
int * const pr=&r;
```

则指针pr被禁写，即pr将始终指向一个地址，成为一个指针常量。它将不能再作为左值而放在赋值号的左边。（举例说明）

同样，禁写指针一定要在定义的时候赋初值。

虽然指针被禁写，但其间接引用并没有被禁写。即可以通过pr对r赋值。*pr=8;


```
void main(void)
```

```
{
```

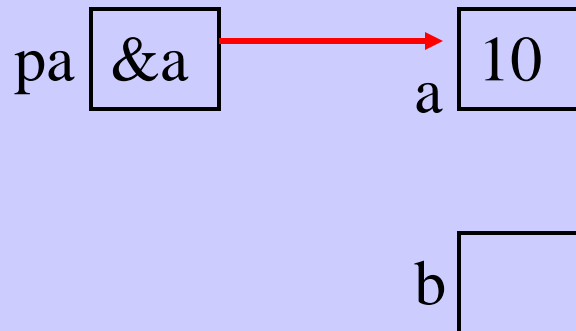
```
    int a,b;
```

```
    int *const pa=&a; //一定要赋初值，pa是常量，不能在程序中  
                      //被改变
```

```
    *pa=10;           //可以间接引用
```

```
    pa=&b;             //非法，pa为常量
```

```
}
```



2) 禁写间接引用

声明语句格式如下：

```
const 数据类型 *指针变量名;
```

所声明的指针指向一禁写的实体，即间接引用不能被改写。如：

```
const int *p;
```

所以程序中不能出现诸如 `*p=` 的语句，但指针`p`并未被禁写，因而可对指针`p`进行改写。

```
void main(void)
```

```
{
```

```
    int a=3,b=5;
```

```
    const int *pa=&b;    //可以不赋初值
```

```
    pa=&a;                //指针变量可以重新赋值
```

```
    cout<<*pa<<endl; //输出3
```

```
    *pa=10;              //非法，指针指向的内容不能赋值
```

```
    a=100;              //变量可以重新赋值
```

```
    cout<<*pa<<endl; //输出100
```

```
}
```

即不可以通过指针对变量重新赋值

3)禁写指针又禁写间接引用

将上面两种情况结合起来，声明语句为下面的格式

`const 数据类型 *const 指针变量名`

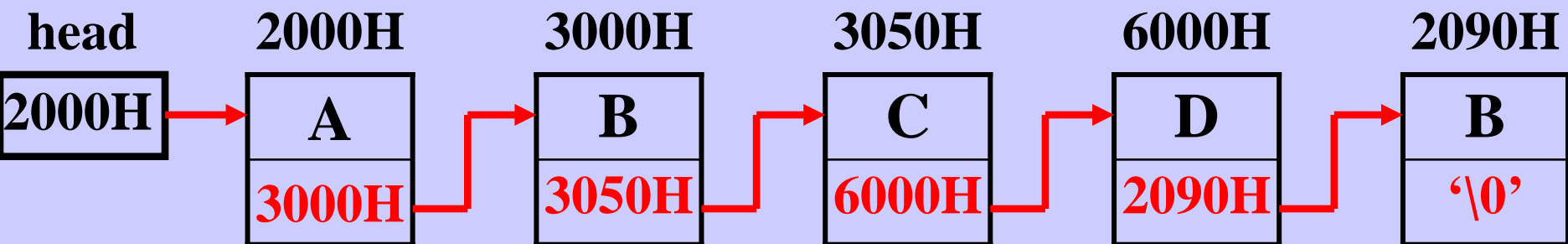
如：`const int *const px=&x`

说明：`px`是一个指针常量，它指向一禁写的实体，并且指针本身也被禁写，诸如：`px=` `*px=` 此类的语句都是非法的。

在定义时必须赋初值。

用指针处理链表

一、链表概述



链表是由一个个结点组成，每一个结点是一个结构体类型的变量，各个结点的类型相同，但其地址不一定连续。具体结点的个数根据需要动态开辟。

每个结点由两部分组成，第一部分放若干数据，第二部分是指针变量，放下一结点的地址。链表头是一指针变量，放第一个结点的地址，若结点的第二部分的值为NULL，表示此链表结束。

二、如何处理链表

1、建立链表

链表结点的结构:

```
struct student
{
    int num;

    float score;

    struct student *next;
};
```

指向同一结构体类型的指针变量

```
#define STU struct student

STU

{
    int num;

    float score;

    STU *next;
};
```

指向同一结构体类型的指针变量

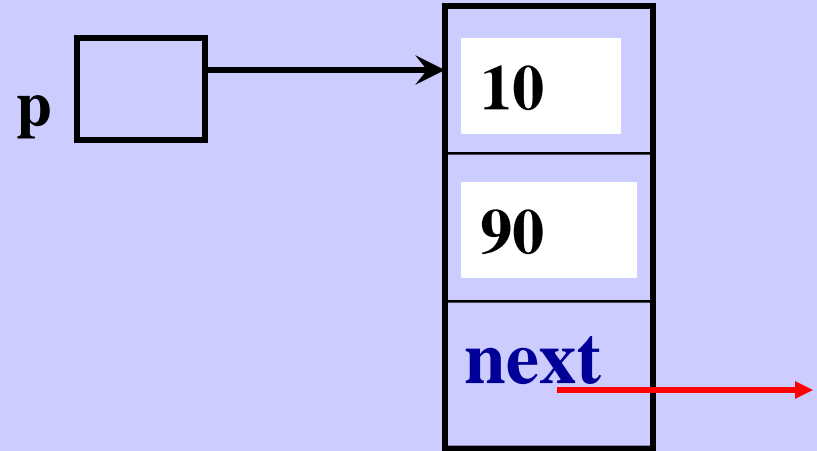
```
struct student
```

```
{ int num;
```

```
    float score;
```

```
    struct student *next;
```

```
};
```



```
struct student *p; //定义了结构体类型的指针
```

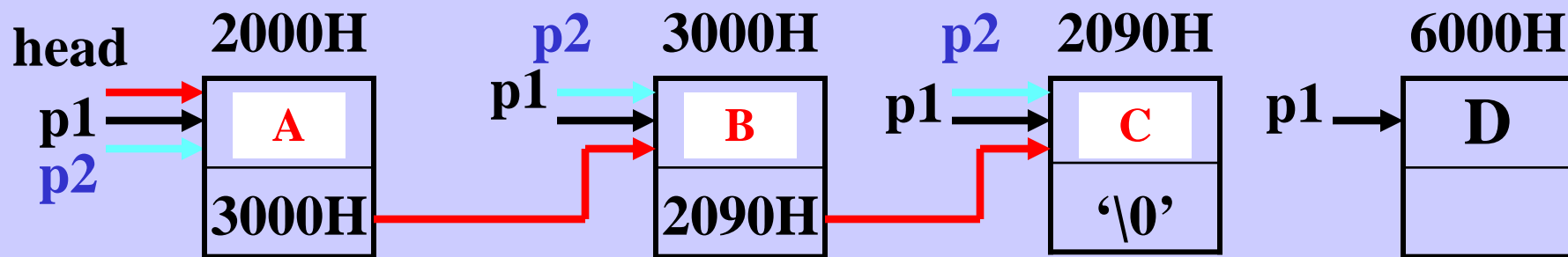
```
p=new student; //用new开辟一结构体空间，将地址赋给p
```

```
p->num=10; //为新开辟的结构体空间中的num成员赋值
```

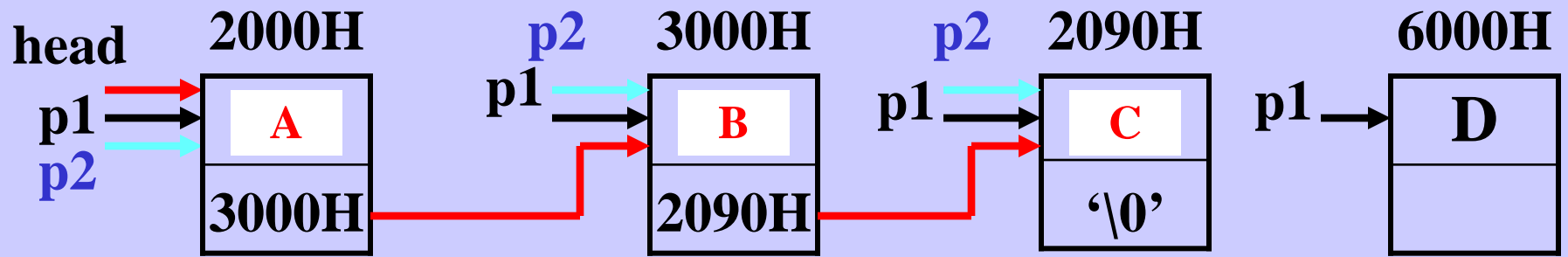
```
p->score=90;
```

用指针引用结构体内的成员

(*p).num



- 1、首先定义两个结构体类型的指针 `STU *p1, *p2;`
- 2、用new在内存中开辟一个结构体变量的空间，将地址赋给p1。
`p1=new student; /* STU struct student */`
- 3、将数据赋给刚开辟的变量空间。
`cin>>p1->num>>p1->score;`
- 4、若输入的数据有效，将首地址作为链表头，`head=p1`；令`p2=p1`，p1继续用new开辟新的内存空间。
`p1=new student; /* STU struct student */`
- 5、将下一个数据赋给新开辟的变量空间。
`cin>>p1->num>>p1->score;`
- 6、若输入的数据有效，将p2与p1连接起来，`p2->next=p1` 再令`p2=p1`，p1继续用new开辟新的内存空间。做5。若输入的数据无效，p2就是链表的尾，则`p2->next=NULL`。



```
STU  *p1, *p2, *head;  
head=NULL;  
p1=p2=new student;  
cin>>p1->num>>p1->score;  
if (p1->num!=0)  
    head=p1;
```

第一结点

```
p1=new student;  
cin>>p1->num>>p1->score;  
if (p1->num!=0)  
{ p2->next=p1; p2=p1;}
```

第二结点

```
p1=new student;  
cin>>p1->num>>p1->score;  
if (p1->num!=0)  
{ p2->next=p1; p2=p1;
```

第三结点

```
p1=new student;  
cin>>p1->num>>p1->score;  
if (p1->num==0)  
    p2->next=NULL;  
return (head);
```

返回链表头

最后结点

```
STU *creat( )
```

```
{ STU *head, *p1,*p2;
```

```
  n=0;
```

n为全局变量，表示结点数

```
  head=NULL;
```

```
  p1=p2=new student;
```

```
  cin>>p1->num>>p1->score;
```

```
  while (p1->num!=0)
```

```
  {   n=n+1;
```

```
      if (n==1) head=p1;
```

```
      else      p2->next=p1;
```

```
      p2=p1;
```

```
      p1=new student;
```

开辟新结点

```
      cin>>p1->num>>p1->score;
```

```
  }
```

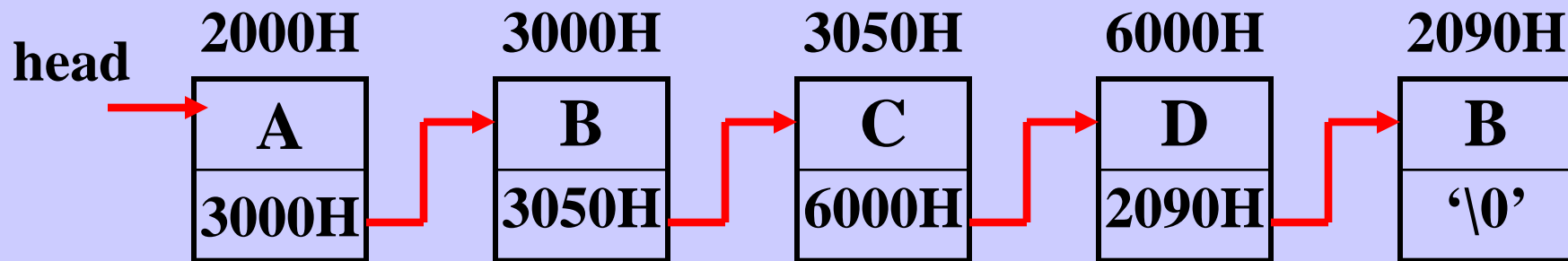
向新结点输入数据

```
  p2->next=NULL;
```

```
  return(head);
```

不满足输入条件，结束

```
}
```



2、输出链表

```
void print(STU * head)
```

```
{ STU *p;
```

```
  p=head;
```

```
  while(p!=NULL)
```

```
  { cout<<p->num<<'\t'<<p->score<<'\n';
```

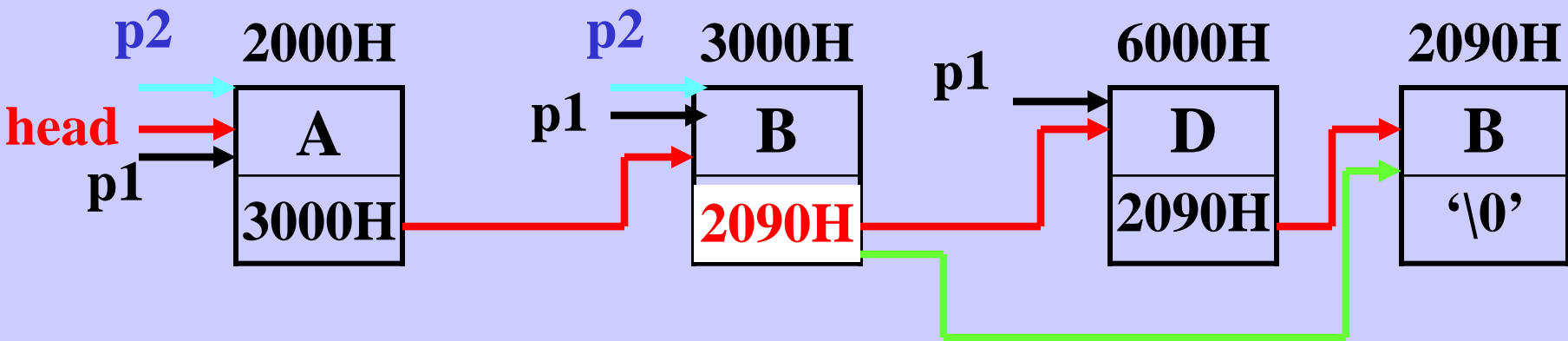
```
    p=p->next;
```

p指向下一结点

```
  }
```

```
}
```

3、删除链表



- 1、首先定义两个结构体类型的指针 `STU *p1, *p2;`
- 2、将链表的表头赋给p1, `p1=head;`
- 3、判断p1所指向的结点是否是要删除的结点 `p1->num == a1`。
- 4、若`p1->num!=a1`, `p2=p1`; p1指向下一个结点`p1=p1->next`,继续判断下一个结点是否是要删除的结点。继续做3。
- 5、若`p1->num==a1`, 则p1当前指向的结点就是要删除的结点, 将p2的指针成员指向p1所指的下一个结点。

这样就删除了一个结点。

`p2->next=p1->next;`

特殊情况：

- 1、若链表为空链表，返回空指针。
- 2、删除的结点为头结点时，head指向下一个结点
- 3、链表内没有要删除的结点，返回提示信息。

```

struct student *del(struct student * head, int num)
{
    struct student *p1,*p2;
    if (head==NULL)
    {
        cout<<"list null\n"; return NULL;
    }
    p1=head;
    while (num!=p1->num&&p1->next!=NULL)
    {
        p2=p1; p1=p1->next;
    }
    if(num==p1->num)
    {
        if (num==head->num) head=p1->next;
        else p2->next=p1->next;
        n=n-1;
    }
    else cout<<"Not found\n";
    return head;
}

```

链表为空

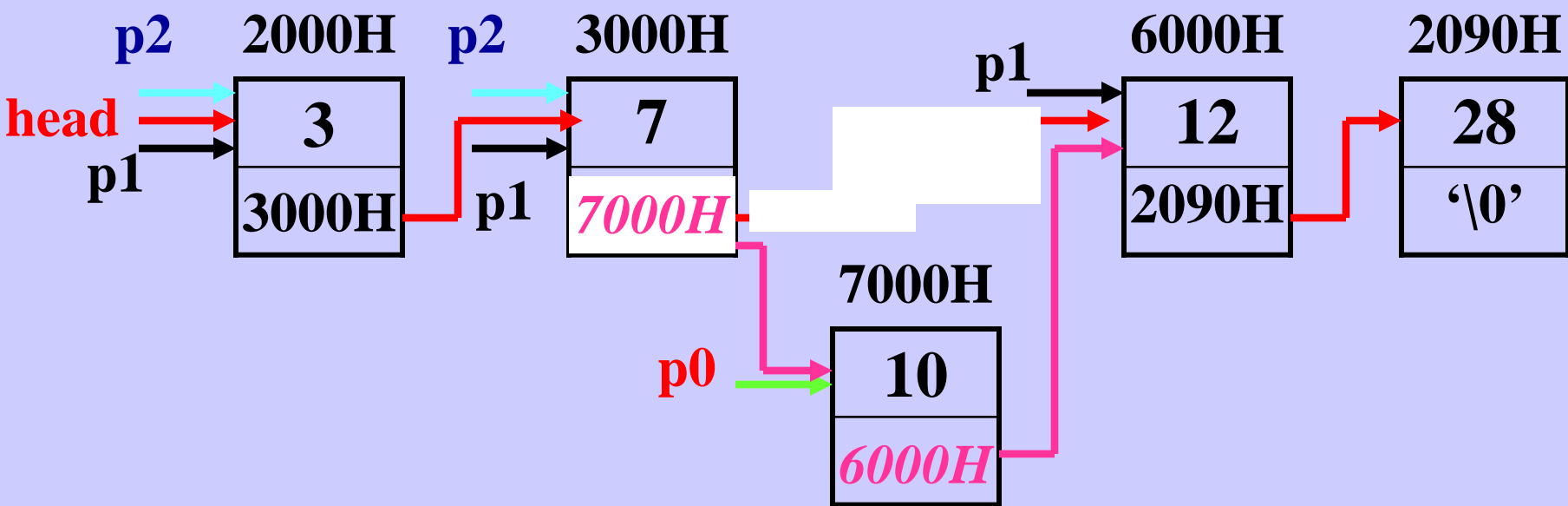
未找到结点，循环

找到结点

结点为第一个

循环结束，没有要找的结点

4、插入结点：要插入结点的链表是排序的链表。插入10。



1、定义三个结构体指针变量 `STU *p1,*p2,*p0`; `p0`指向要插入的结点。`p1=head`;

2、比较`p1->num`与`p0->num`, 若`p1->num < p0->num`, `p2=p1`;
`p1=p1->next`; 继续比较。

3、若`p1->num >= p0->num`, `p0`应插在`p1`与`p2`之间, 则`p2->next=p0`;
`p0->next=p1`;

特殊情况:

1、若链表为空链表，将插入结点作为唯一的结点，
`head=p0`;返回。

2、若插入结点中的数据最小，则插入的结点作为
头结点。

`p0->next=head;`

`head=p0;`

3、插入到链尾，插入结点为最后一个结点。

`p2->next=p0;`

`p0->next=NULL;`

STU *insert(STU * head, STU * stud)

```
{ STU *p0,*p1,*p2;
```

```
  p1=head;  p0=stud;
```

```
  if (head==NULL)
```

链表为空

```
    { head=p0; p0->next=NULL;}
```

```
  else
```

```
    while((p0->num>p1->num)&&(p1->next!=NULL))
```

```
        { p2=p1; p1=p1->next; }
```

未找到结点，循环

```
    if (p0->num<=p1->num)
```

```
        { if (head==p1) head=p0;
```

插入在第一个结点前

找到结点

```
        else p2->next=p0;
```

```
        p0->next=p1;
```

```
    }
```

```
    else {p1->next=p0; p0->next=NULL;}
```

插入在最后一个后

```
  n=n+1;    return (head);
```

```
}
```

```
void main(void) { STU *head, stu;
    int del_num;
    head=creat( );
    print(head);
    cout<<“Input the deleted number:\n”;
    cin>>del_num;
    head=del(head,del_num);
    print(head);
    cout<<“Input the inserted record:\n”;
    cin>>stu.num>>stu.score;
    head=insert(head, &stu);
    print(head);
}
```

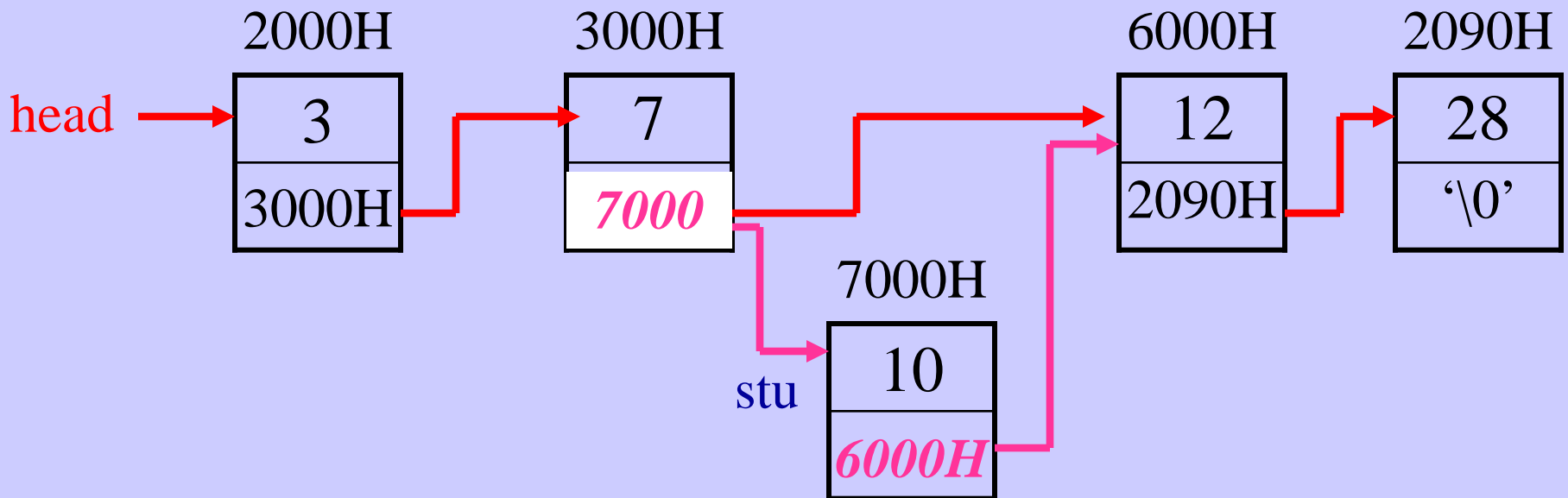
变量, 固定空间

建立链表

打印链表

删除结点

插入结点



```
void main(void)
```

```
{ STU *head,*stu;
```

```
int num;
```

指针，可以赋值

```
.....
```

```
cout<<“Input the inserted record:\n”;
```

```
stu=new student;
```

如果要插入结点，动态开辟新结点

```
cin>>stu->num>>stu->score;
```

```
while (stu->num!=0)
```

```
{ head=insert(head, stu);
```

```
print(head);
```

```
cout<<“Input the inserted record:\n”;
```

```
stu=new student;
```

重新开辟空间

```
cin>>stu->num>>stu->score;
```

```
}
```

```
}
```

用typedef定义类型

typedef定义新的类型来代替已有的类型。

typedef	已定义的类型	新的类型
---------	--------	------

```
typedef float REAL
```

REAL **x, y;** **float** **x, y;**

1、typedef可以定义类型，但不能定义变量。

2、tyoedef只能对已经存在的类型名重新定义一个类型名，而不能创建一个新的类型名。

```
typedef struct student REC x, y, *pt;
```

```
{ int i;
```

```
struct student x, y, *pt;
```

```
int *p;
```

```
} REC;
```

```
typedef char *CHARP;
```

```
CHARP p1,p2;      char *p1, *p2;
```

```
typedef char STRING[81];
```

```
STRING s1, s2, s3;  char s1[81], s2[81], s3[81];
```

1、先按定义变量的方法写出定义体 **char s[81];**

2、把变量名换成新类型名 **char STRING[81];**

3、在前面加typedef **typedef char STRING[81];**

4、再用新类型名定义变量 **STRING s;**

```
#define REAL float    编译前简单替换
```

typedef: 编译时处理，定义一个类型替代原有的类型。

面向对象的程序设计

第九章 类和对象

按钮对象：

按钮的内容、大小，按钮的字体、图案等等

针对按钮的各种操作，创建、单击、双击、拖动等

班级对象：

班级的静态特征，所属的系和专业、班级的人数，所在的教室等。这种静态特征称为属性；

班级的动态特征，如学习、开会、体育比赛等，这种动态特征称为行为。

任何一个对象都应当具有这两个要素，一是属性(attribute)；二是行为(behavior)，即能根据外界给的信息进行相应的操作。**对象是由一组属性和一组行为构成的。**

面向对象的程序设计采用了以上人们所熟悉的这种思路。使用面向对象的程序设计方法设计一个复杂的软件系统时，**首要的问题是确定该系统是由哪些对象组成的，并且设计这些对象。在C++中，每个对象都是由数据和函数(即操作代码)这两部分组成的。**

我们可以对一个对象进行封装处理，把它的一部分属性和功能对外界屏蔽，也就是说从外界是看不到的、甚至是不可知的。

使用对象的人完全可以不必知道对象内部的具体细节，只需了解其外部功能即可自如地操作对象。

把对象的内部实现和外部行为分隔开来。

传统的面向过程程序设计是围绕功能进行的，用一个函数实现一个功能。**所有的数据都是公用的**，一个函数可以使用任何一组数据，而一组数据又能被多个函数所使用。程序设计者必须考虑每一个细节，什么时候对什么数据进行操作。

面向对象程序设计采取的是另外一种思路。它面对的是一个对象。实际上，每一组数据都是有特定的用途的，是某种操作的对象。也就是说，**一组操作调用一组数据**。

程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即决定把哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。各个对象的操作完成了，整体任务也就完成了。

因此人们设想把相关的数据和操作放在一起，形成一个整体，与外界相对分隔。这就是面向对象的程序设计中的对象。

在面向过程的结构化程序设计中，人们常使用这样的公式来表述程序：

程序=算法+数据结构

面向对象的程序组成：

对象 = 算法 + 数据结构

程序=(对象+对象+对象+.....)+ 消息

消息的作用就是对对象的控制。

程序设计的关键是设计好每一个对象以及确定向这些对象发出的命令，使各对象完成相应的操作。

每一个实体都是对象。有一些对象是具有相同的结构和特性的。

每个对象都属于一个特定的类型。

在C++中对象的类型称为类(class)。类代表了某一批对象的共性和特征。类是对象的抽象，而对象是类的具体实例(instance)。

类的定义

类是一种复杂的数据**类型**，它是将**不同类型的数据**和**与这些数据相关的运算**封装在一起的集合体。

类将一些数据及与数据相关的**函数**封装在一起，使类中的数据得到很好的“保护”。在大型程序中不会被随意修改。

类的定义格式:

关键字 **class** **类名** **类名**
{
 私有 private :
 成员数据;
 成员函数;
 公有 public :
 成员数据;
 成员函数;
 保护 protected:
 成员数据;
 成员函数;
}; **分号不能少**

class Student

{ private :
 char Name[20];
 float Math;
 float Chiese;
public :
 float average;
 void SetName(char *name);
 void SetMath(float math);
 void SetChinese(float ch);
 float GetAverage(void);
};

用关键字**private**限定的成员称为**私有成员**，对私有成员**限定在该类的内部使用**，即只允许该类中的成员函数使用私有的成员数据，对于私有的成员函数，只能被**该类内的成员函数调用**；类就相当于私有成员的作用域。

用关键字**public**限定的成员称为**公有成员**，
公有成员的数据或函数不受类的限制，**可以在类内或类外自由使用**；对类而言是透明的。

而用关键字**protected**所限定的成员称为**保护成员**，只允许在类内及该类的派生类中使用保护的数据或函数。即保护成员的作用域是**该类及该类的派生类**。

	私有成员	公有成员	保护成员
类内函数	可以调用	可以调用	可以调用
类外函数	不可调用	可以调用	不可调用

	私有函数	公有函数	保护函数
类内函数	可以调用	可以调用	可以调用
类外函数	不可调用	可以调用	不可调用

每一个限制词(**private**等)在类体中可使用多次。一旦使用了限制词，该限制词一直有效，直到下一个限制词开始为止。

如果未加说明，类中成员默认访问权限是**private**，即私有的。

```
class Student
```

```
{
```

```
    char Name[20];
```

```
    float Math;
```

```
    float Chiese;
```

```
    public :
```

```
        float average;
```

```
        void SetName(char *name);
```

```
        void SetMath(float math);
```

```
        void SetChinese(float ch);
```

```
        float GetAverage(void);
```

```
};
```

均为私有权限

均为公有权限


```
class A
```

```
{  float x, y;
```

```
  public:
```

```
    void Setxy(float a,float b)
```

```
    { x=a; y=b; }
```

```
    void Print(void)
```

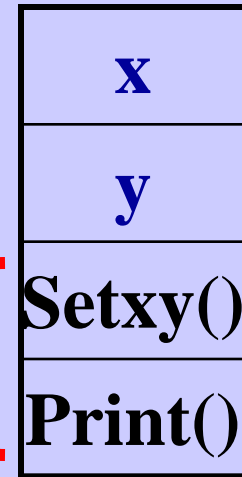
```
    { cout<<x<<'\t'<<y<<endl; }
```

```
};
```

私有数据

公有函数

A



在类外不能直接使用 **x** 或 **y** ，必须通过**Setxy()**给 **x** 或 **y** 赋值，通过**Print()**输出 **x** 或 **y** 。

成员函数与成员数据的定义不分先后，可以先说明**函数原型**，再在类体外定义函数体。

```
class A
{
    float x, y;

public:
    void Setxy(float a, float b)
    { x=a; y=b; }
    void Print(void)
    { cout<<x<<'\\t'<<y<<endl; }

};
```

在类体内定义成员函数

```
class A
```

```
{    float x, y;
```

```
public:
```

```
void Setxy(float a,float b);
```

```
void Print(void);
```

```
};
```

```
void A::Setxy(float a,float b)
```

```
{ x=a; y=b; }
```

```
void A::Print(void)
```

```
{    cout<<x<<'\t'<<y<<endl; }
```

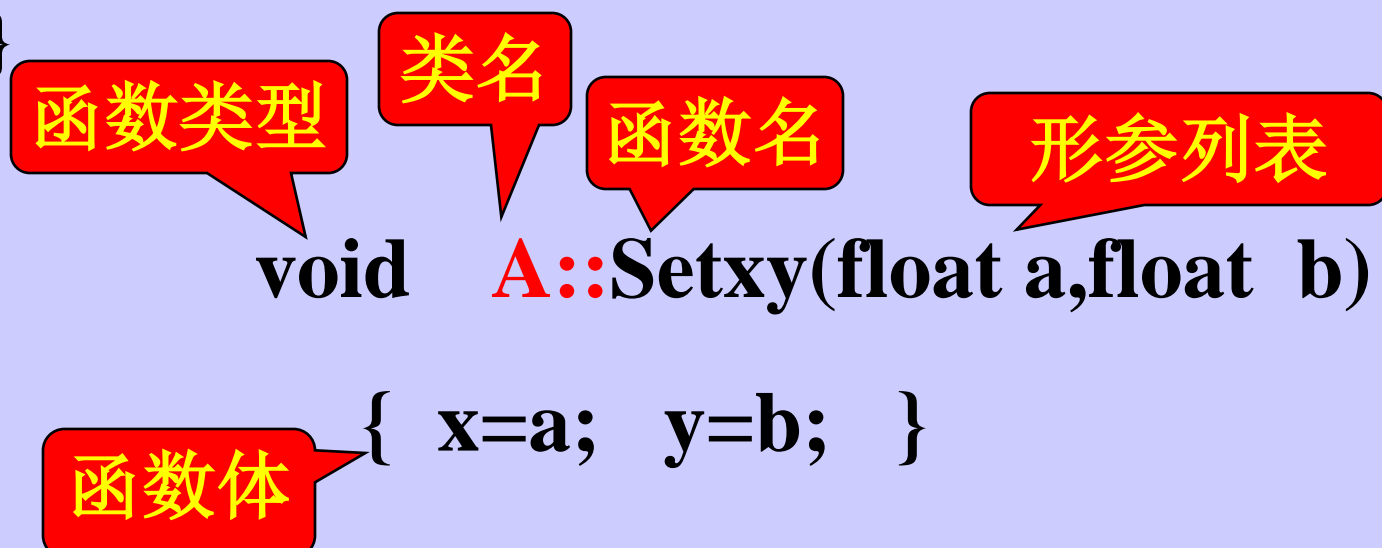
在类体内说明
成员函数原型

在类体外定
义成员函数

在类体外定义成员函数的格式:

<type> < class_name > :: < func_name > (<参数表>)

{
..... //函数体

}

void **A::Setxy**(float a, float b)

{ x=a; y=b; }

在定义一个类时，要注意如下几点：

1、类具有封装性，并且类只是定义了一种结构（样板），所以类中的任何成员数据均不能使用关键字**extern**，**auto**或**register**限定其存储类型。

2、在定义类时，只是定义了一种导出的数据类型，并不为类分配存储空间，所以，在定义类中的数据成员时，不能对其初始化。如：

```
class Test { int x=5,y=6; //是不允许的  
  
    extern float x; //是不允许的  
  
}
```

在C++语言中，结构体类型只是类的一个特例。结构体类型与类的唯一的区别在于：在类中，其成员的缺省的存取权限是私有的；而在结构体类型中，其成员的缺省的存取权限是公有的。

内联成员函数

当我们定义一个类时，可以在类中直接定义函数体。这时成员函数在编译时是作为内联函数来实现的。

同时，我们也可以在类体外定义类的内联成员函数，在类体内说明函数，在类体外定义时，在成员函数的定义前面加上关键字**inline**。

```
class A
{
    float x, y;
public:
    void Setxy(float a, float b);
    void Print(void);
};
```

说明该成员函数为内联

```
inline void A::Setxy(float a, float b)
{
    x=a; y=b;
}

inline void A::Print(void)
{
    cout<<x<<'\t'<<y<<endl;
}
```

对象

在定义类时，只是定义了一种数据类型，即说明程序中可能会出现该类型的数据，并不为类分配存储空间。

只有在定义了属于类的变量后，系统才会为类的变量分配空间。

类的变量我们称之为对象。

对象是类的实例，定义对象之前，一定要先说明该对象的类。

不同对象占据内存中的不同区域，它们所保存的数据各不相同，但对成员数据进行操作的成员函数的程序代码均是一样的。

对象的定义格式：

《存储类型》 类名 对象1，对象2 《,.....》；

Student st1, st2;



类名

对象名

在建立对象时，只为对象分配用于保存数据成员的内存空间，而成员函数的代码为该类的每一个对象所共享。

对象的定义方法同结构体定义变量的方法一样，也分三种，当类中有数据成员的访问权限为私有时，不允许对对象进行初始化。

```
class A {  
    float x,y;  
public:  
    void Setxy( float a, float b ){ x=a; y=b; }  
    void Print(void) { cout<<x<<'\t'<<y<<endl; }  
} a1,a2;  
void main(void)  
{ A a3,a4;  
}
```

定义全局对象

定义局部对象

对象的使用

一个对象的成员就是该对象的类所定义的成员，有成员数据和成员函数，引用时同结构体变量类似，用“.”运算符。

```

class A {
    float x,y;
public:
    float m,n;
    void Setxy( float a, float b ){ x=a; y=b; }
    void Print(void) { cout<<x<<'\\t'<<y<<endl; }
};

```

```

void main(void)
{
    A a1,a2; //定义对象
    a1.m=10; a1.n=20; //为公有成员数据赋值
    a1.Setxy(2.0 , 5.0); //为私有成员数据赋值
    a1.Print();
}

```

输出： 2 5

a1	
x	2.0
y	5.0
m	10
n	20
Setxy()	
Print()	

a2	
x	
y	
m	
n	
Setxy()	
Print()	

用成员选择运算符“.”只能访问对象的**公有成员**，而不能访问对象的私有成员或保护成员。若要访问对象的私有的数据成员，只能通过对象的公有成员函数来获取。

```
class A {  
    float x,y;  
public:  
    float m,n;  
    void Setxy( float a, float b ){ x=a; y=b; }  
    void Print(void) { cout<<x<<'\t'<<y<<endl; }  
};
```

必须通过类内公有函数访问私有数据成员

```
void main(void)  
{  
    A a1,a2;  
    a1.m=10; a1.n=20; //为公有成员数据赋值  
    a1.x=2; a1.y=5;  
    a1.Setxy(2.0,5.0);  
    a1.Print();  
}
```

非法，私有成员不能在类外访问

同类型的对象之间可以整体赋值，这种赋值与对象的成员的访问权限无关。

```
class A {  
    float x,y;  
public:  
    float m,n;  
    void Setxy( float a, float b ){ x=a; y=b; }  
    void Print(void) { cout<<x<<'\t'<<y<<endl; }  
};  
  
void main(void)  
{  
    A a1,a2;  
    a1.m=10; a1.n=20;    //为公有成员数据赋值  
    a1.Setxy(2.0,5.0);  
    a2=a1;   
    a1.Print(); a2.Print();  
}
```

同类型的对象之
间可以整体赋值

相当于成员数
据间相互赋值

对象可以作函数的入口参数（实参、形参），也可以作函数的出口参数。这与一般变量作为函数的参数是完全相同的。

可以定义类类型的指针，类类型的引用，对象数组，指向类类型的指针数组和指向一维或多维数组的指针变量

一个类的对象，可作为另一个类的成员

类作用域、类类型的作用域和对象的作用域

类体的区域称为**类作用域**。类的成员函数与成员数据，其作用域都是属于类的作用域，仅在该类的范围内有效，**故不能在主函数中直接通过函数名和成员名来调用函数。**

```
class A {  
    float x,y;  
public:  
    float m,n;  
    void Setxy( float a, float b ){ x=a; y=b; }  
    void Print(void) { cout<<x<<'\t'<<y<<endl; }  
};
```

```
void main(void)  
{  
    A a1,a2;  
    a1.m=20; a1.n=10;  
    a1.Setxy(2.0, 5.0);  
    a1.Print();  
}
```

用对象名调用

```
void main(void)  
{  
    A a1,a2;  
    m=20; n=10;  
    Setxy(2.0, 5.0);  
    Print();  
}
```

不能直接调用

类类型的作用域：在函数定义之外定义类，其类名的作用域为**文件作用域**；而在函数体内定义的类，其类名的作用域为**块作用域**。

对象的作用域与前面介绍的变量作用域完全相同，**全局对象、局部对象、局部静态对象**等。

```

class A {
    float x,y;
public:
    float m,n;
    void Setxy( float a, float b ){ x=a; y=b; }
    void Print(void) { cout<<x<<'\\t'<<y<<endl; }
}a3,a4;
void main(void)
{
    A a1,a2;
    class B{
        int i,j;
    public :
        void Setij(int m, int n){ i=m; j=n; }
    };
    B b1,b2;
    a1.Setxy(2.0, 5.0); b1.Setij(1,2);
}

```

类A：文件作用域，在整个文件中有效

全局对象

局部对象

类B：块作用域，在函数内有效。

类的嵌套

在定义一个类时,在其类体中又包含了一个类的完整定义,称为类的嵌套。

类是允许嵌套定义的。

```
class A {
```

```
    class B{
```

```
        int i,j;
```

类B包含在类A中，
为嵌套定义

```
    public :
```

```
        void Setij(int m, int n){ i=m; j=n; }
```

```
};
```

```
    float x,y;
```

```
public:
```

嵌套类的对象

```
    B b1,b2;
```

在类A的定义中，并不为b1,b2分配空间，只有在定义类A的对象时，才为嵌套类的对象分配空间。嵌套类的作用域在类A的定义结束时结束。

```
    void Setxy( float a, float b ){ x=a; y=b; }
```

```
    void Print(void) { cout<<x<<'\t'<<y<<endl; }
```

```
};
```

类的对象如何引用私有数据成员

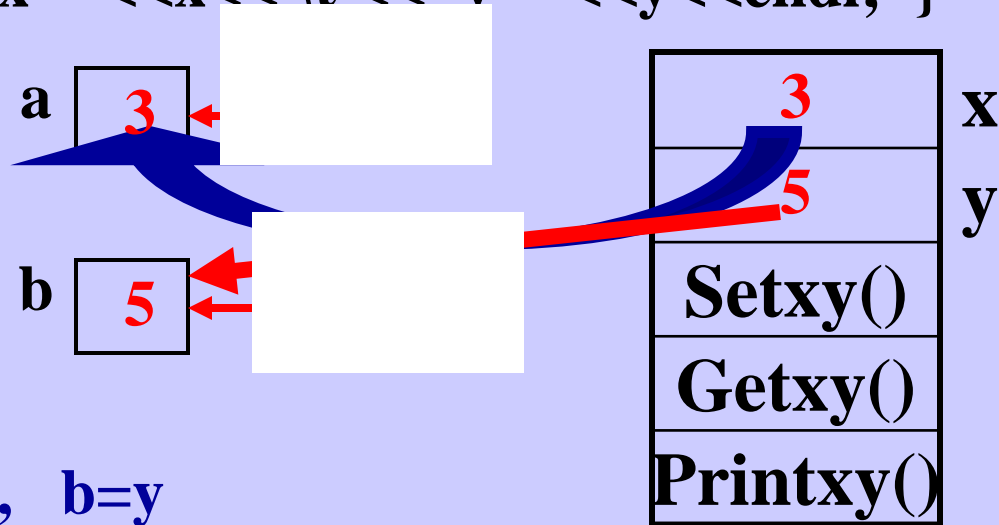
1、通过公有函数为私有成员赋值

```
class Test{  
    int x , y;  
  
public:  
    void Setxy(int a, int b){x=a;   y=b;}  
    void Printxy(void) {cout<<"x="<<x<<"\t"<<"y="<<y<<endl;}  
};  
  
void main(void)  
{  
    Test  p1,p2;  
    p1.Setxy(3, 5);  
    p1.Printxy( );  
}
```

调用公有函数为
私有对象赋值

2、利用指针访问私有数据成员

```
class Test{  
    int x,y;  
  
public:  
    void Setxy(int a, int b) {x=a; y=b;}  
    void Getxy(int *px, int *py) {*px=x;*py=y;} //提取x,y值  
    void Printxy(void){cout<<"x="<<x<<"\t"<<"v="<<y<<endl; }  
};  
  
void main(void)  
{  
    Test p1,p2;  
    p1.Setxy(3,5);  
    int a,b;  
    p1.Getxy(&a, &b);//将 a=x, b=y  
    cout<<a<<"\t"<<b<<endl;  
}
```



输出: 3 5

3、利用函数访问私有数据成员

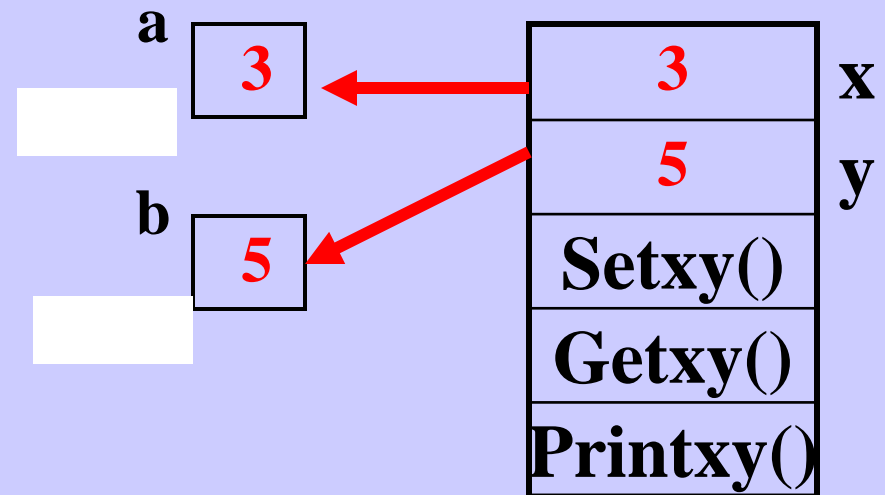
```
class Test{  
    int x,y;  
  
public:  
    void Setxy(int a, int b) {x=a; y=b;}  
    int Getx(void) { return x;} //返回x值  
    int Gety(void) { return y;} //返回y值  
    void Printxy(void){cout<<"x="<<x<<"\t"<<"y="<<y<<endl; }  
};  
  
void main(void)  
{ Test p1,p2;  
  p1.Setxy(3,5);  
  int a,b;  
  a=p1.Getx(); b=p1.Gety(); //将 a=x, b=y  
  cout<<a<<"\t"<<b<<endl;  
}
```

函数值就是私有
成员变量的值

4、利用引用访问私有数据成员

```
class Test{  
    int x,y;  
  
public:  
    void Setxy(int a, int b){ x=a; y=b;}  
    void Getxy(int &px, int &py) { px=x; py=y; } //提取x,y值  
    void Printxy(void){cout<<"x="<<x<<"\t"<<"y="<<y<<endl; }  
};
```

```
void main(void)  
{  
    Test p1,p2;  
    p1.Setxy(3,5);  
    int a,b;  
    p1.Getxy(a, b);//将 a=x, b=y  
    cout<<a<<"\t"<<b<<endl;  
}
```



输出： 3 5

类引用举例（三角形类：三角形的三边及与三边相关的运算）

```
class Triangle{  
  
private:  
  
    float a,b,c; //三边为私有成员数据  
  
public:  
  
    void Setabc(float x, float y, float z);//置三边的值  
    void Getabc(float &x, float &y, float &z);//取三边的值  
    float Perimeter(void);//计算三角形的周长  
    float Area(void);//计算三角形的面积  
    void Print(void);//打印相关信息  
  
};
```

```
void Triangle::Setabc (float x,float y,float z)
```

```
    {a =x; b=y; c=z; } //置三边的值
```

```
void Triangle::Getabc (float &x,float &y,float &z) //取三边的值
```

```
    {x=a; y=b; z=c;}
```

```
float Triangle::Perimeter (void)
```

```
    {return (a+b+c)/2;} //计算三角形的周长
```

```
float Triangle::Area (void) //计算三角形的面积
```

```
{float area, p;
```

```
    p= Perimeter();
```

```
    area=sqrt((p-a)*(p-b)*(p-c)*p);
```

```
    return area;
```

```
}
```

```
void Triangle::Print(void) //打印相关信息
```

```
{cout<<"Peri="<<Perimeter()<<"\t"<<"Area="<<Area()<<endl;
```

```
void main(void)
{
    Triangle Tri1;      //定义三角形类的一个实例（对象）
    Tri1.Setabc (4,5,6); //为三边置初值
    float x,y,z;
    Tri1.Getabc (x,y,z); //将三边的值为x,y,z赋值
    cout<<x<<"\t"<<y<<"\t"<<z<<endl;
    cout<<"s="<<Tri1.Perimeter ()<<endl;//求三角形的周长
    cout<<"Area="<<Tri1.Area ()<<endl;//求三角形的面积
    cout<<"Tri1:"<<endl;
    Tri1.Print ();//打印有关信息
}
```

类引用举例（学生类：学生的姓名成绩及相关的运算）

```
class Stu
{
    char Name[20];        //学生姓名
    float Chinese;        //语文成绩
    float Math;           //数学成绩

public:
    float Average(void); //计算平均成绩
    float Sum(void);     //计算总分
    void Show(void);     //打印信息
    void SetStudent(char*,float,float); //为对象置姓名、成绩
    void SetName(char *); //为对象置姓名
    char *GetName(void);  //取得学生姓名
};
```

```
float Stu::Average(void){ return (Chinese+Math)/2;}//平均成绩
```

```
float Stu::Sum(void){ return Chinese+Math; }//总分
```

```
void Stu::Show(void) //打印信息
```

```
{ cout<<"Name: "<<Name<<endl<<"Score: "<<Chinese<<"\t"<<
    Math<<"\t"<<"average: "<<Average()<<"\t"<<"Sum: "<<Sum()<<endl;
}
```

```
void Stu::SetStudent(char *name,float chinese,float math)
```

```
{    strcpy(Name,name); //置姓名
    Chinese=chinese;    //置语文成绩
    Math=math;          //置数学成绩
}
```

```
char * Stu::GetName(void){ return Name;}//返回姓名
```

```
void main(void)  
{    Stu p1,p2;  
  
    p1.SetStudent(“Li qing”,98,96);//对象置初值  
    p2.SetStudent("Wang Gang",90,88); //对象置初值  
    p1.Show();//打印信息  
    p2.Show();//打印信息  
    p1.SetName (“Zhao jian”);//重新置p1对象的名字  
    p1.Show ();  
  
    cout<<“p1.Name: ”<<p1.GetName ()<<endl;//打印对象的名字  
    cout<<“p1.average: ”<<p1.Average ()<<endl;//打印对象的成绩  
}
```


成员函数的重载

类中的成员函数与前面介绍的普通函数一样，成员函数可以带有缺省的参数，也可以重载成员函数。

重载时，函数的形参必须在类型或数目上不同。

```
class Test{
```

```
    int x , y;
```

```
    int m, n;
```

```
public:
```

```
    void Setxy(int a, int b){x=a; y=b;}
```

```
    void Setxy(int a,int b,int c,int d){ x=a;y=b;m=c;n=d;}
```

```
    void Printxy(int x){cout<< "m="<<m<<'\t'<<"n="<<n<<endl;}
```

```
    void Printxy(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl;}
```

```
};
```

```
void main(void)
```

```
{    Test  p1,p2;
```

```
    p1.Setxy(3, 5); p2.Setxy(10,20,30,40);//参数不同
```

```
    p1.Printxy();
```

```
    p2.Printxy();  p2.Printxy(2);//参数、类型不同
```

```
}
```

输出: x=3 y=5

x=10 y=20

m=30 n=40

```
class Stu
```

```
{    char Name[20];  
    float Chinese;  
    float Math;  
    float English;  
    float Physical;
```

```
public:
```

```
    float Average(void);//语文、数学平均成绩  
    float Average(int n);//四门课的平均成绩  
    float Sum(void);//语文、数学总分  
    float Sum(int n);//四门课总分  
    void Show(void);  
    void SetStudent(char*,float,float);//置姓名、语文、数学初值  
    void SetStudent(char *, float,float,float,float);//置姓名、成绩  
    void SetName(char *);  
    char *GetName(void);
```

```
};
```

可以有缺省参数的成员函数，若形参不完全缺省，则必须从形参的右边开始缺省。

缺省参数的成员函数

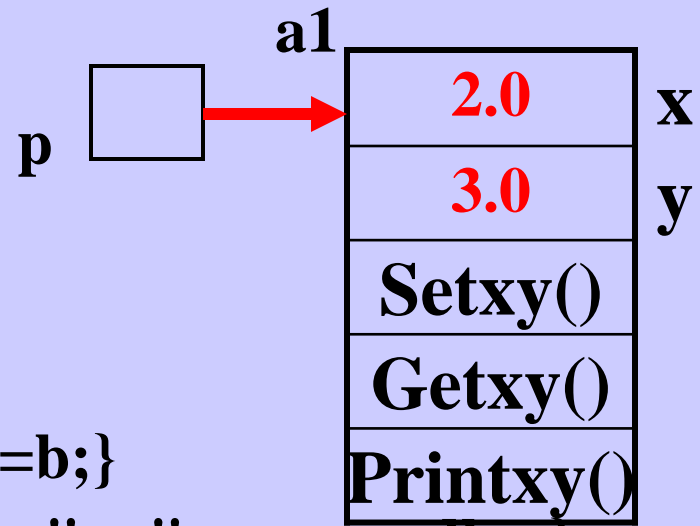
```
class A{  
    float x,y;  
  
public:  
    float Sum(void) { return x+y; }  
    void Set(float a,float b=10.0) { x=a; y=b;}  
    void Print(void) { cout<<"x="<<x<<"\t"<<"y="<<y<<endl; }  
};  
  
void main(void)  
{  
    A a1,a2;  
    a1.Set (2.0,4.0);  
    cout<<"a1: "; a1.Print ();  
    cout<<"a1.sum="<<a1.Sum ()<<endl;  
    a2.Set(20.0);  
    cout<<"a2: "; a2.Print ();  
    cout<<"a2.sum="<<a2.Sum ()<<endl;  
}
```

不缺省参数, a1.x=2, a1.y=4

缺省参数, a2.x=20, a2.y=10

定义类的指针及如何用指针来引用对象

```
class A{  
    float x,y;  
public:  
    float Sum(void) { return x+y; }  
    void Set(float a,float b) { x=a; y=b;}  
    void Print(void) { cout<<"x="<<x<<"\t"<<"y="<<y<<endl; }  
};
```



```
void main(void)  
{  
    A a1,a2;  
    A *p;           //定义类的指针  
    p=&a1;          //给指针赋值  
    p->Set(2.0, 3.0); //通过指针引用对象的成员函数  
    p->Print();  
    cout<<p->Sum()<<endl;  
    a2.Set(10.0, 20.0); a2.Print();  
}
```

定义类的数组及数组中元素的引用

```
void main(void)
```

```
{
```

```
    Stu stu[3];    //定义类的数组
```

```
    Stu *pstu;    //定义类的指针
```

```
    pstu=stu;     //为指针赋值
```

```
    int i;
```

```
    stu[0].SetStudent ("A",90,90);//通过数组元素的引用赋值
```

```
    stu[1].SetStudent ("B",80,80);
```

```
    stu[2].SetStudent ("C",70,70);
```

```
    for(i=0;i<3;i++)
```

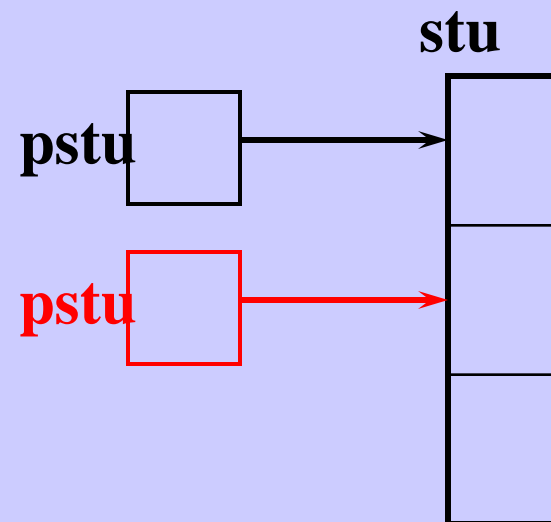
```
    {
```

```
        pstu->Show ();//指针变量指向数组元素
```

```
        pstu++;      //指针变量加一，指向下一元素
```

```
    }
```

```
}
```



返回引用类型的成员函数(可以返回私有数据成员的引用)

```
class A{
    float x,y;
public:
    float &Getx(void ) { return x; } //返回x的引用
    void Set(float a,float b) { x=a; y=b; }
    void Print(void) { cout<<x<<'\\t'<<y<<endl; }
};

void main(void)
{
    A a1,a2;
    a1.Set (3,5);
    cout<<"a1: ";    a1.Print ();
    a1.Getx()=30;    //将a1对象中的x成员赋值
    cout<<"changed a1: ";
    a1.Print ();
}
```

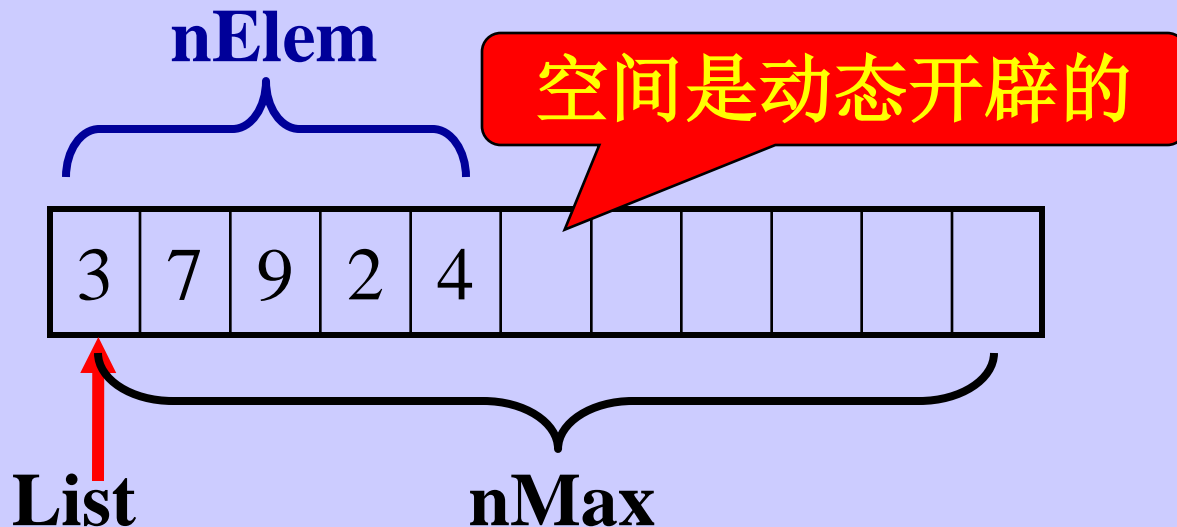

线性表的应用

线性表用来存放若干个整数，用一个指针指向其存放整数的首地址。当存放的数据大于原先开辟的空间时，线性表自动**动态开辟**空间，储存所有的整数。

线性表有**三个参数**来描述：指向线性表存储空间首地址的指针变量**List**；无符号整数**nMax**，指示表的最大长度；无符号整数**nElem**，指示表中实际所放的数据个数。

实际上，线性表相当于一个整型数组，**List**为数组的首地址，**nMax**为数组在内存开辟的空间数，**nElem**为数组中实际存放的元素个数。只不过**数组的空间是动态开辟的**。

```
class ListClass {  
    int *List;    //指向线性表的指针  
    unsigned nMax; //表的最大长度  
    unsigned nElem; //表中当前的元素个数  
    .....  
};
```



```
class ListClass {  
    int *List;    //指向线性表的指针  
    unsigned nMax; //表的最大长度  
    unsigned nElem; //表中当前的元素个数  
public:  
    void Init(int n=10); //初始化针表，最大长度的缺省值为10  
    void Elem(int);      //在线性表尾增加一个元素  
    int & Elem(unsigned n); //返回线性表中第n个元素的引用  
    unsigned Elem(void);   //返回当前线性表中元素的个数  
    unsigned Max(void); //返回线性表的长度（占用空间数）  
    void Print(void);     //输出线性表中所有的元素  
    int GetElem(int i);  //返回线性表中第i个元素的值  
    void Destroy(void); //收回线性表占用的存储空间  
};
```

```

void ListClass::Elem (int elem) //在线性表尾增加一个元素
{
    nElem=nElem+1;
    if(nElem>nMax)                //空间不够，动态开辟存储空间
    {
        int *list;
        list=new int[nMax+1]; //开辟一较大的空间
        for(int i=0;i<nMax;i++)
            list[i]=List[i];
        nMax=nMax+1;
        list[i]=elem;
        delete []List;
        List=list;
    }
    else    {
        List[nElem-1]=elem; }
}

```

重新定义一个指针list，用其开辟一较大的空间，将原有空间的数全部拷贝到新的空间，并将新的数据放在新空间内，同时将旧的空间收回，将线性表的指针指向新空间。

this 指针

不同对象占据内存中的不同区域，它们所保存的数据各不相同，但对成员数据进行操作的成员函数的程序代码均是一样的。

```
class A{
```

```
    int x,y;
```

```
    public:
```

```
    void Setxy(int a, int b)
```

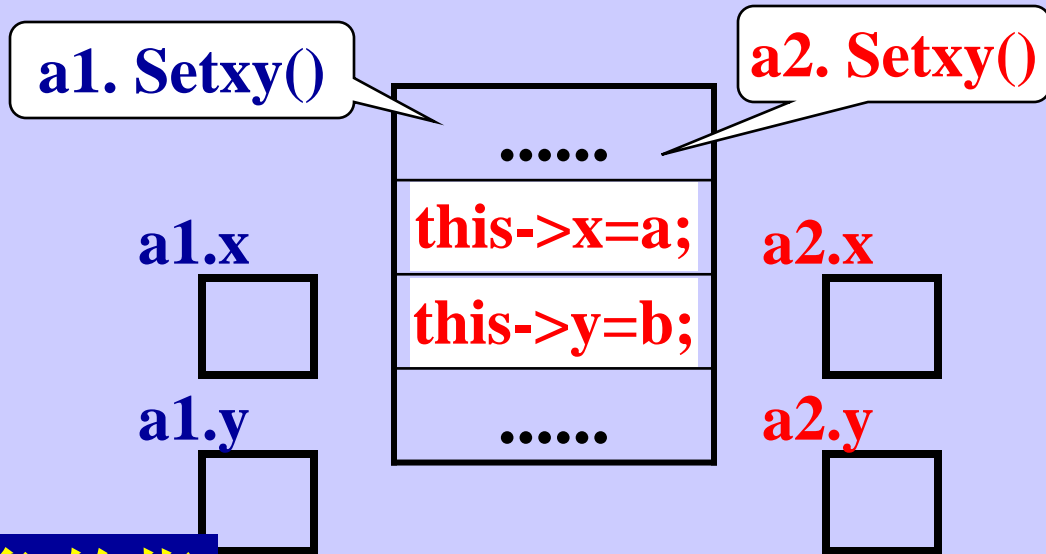
```
    { x=a; y=b;}
```

```
};
```

```
A  a1, a2;
```

```
a1.Setxy(1,2);
```

```
a2.Setxy(3,4);
```



系统自动将对象的指针带到成员函数中

当对一个对象调用成员函数时，编译程序先将对象的地址赋给this指针，然后调用成员函数，每次成员函数存取数据成员时，也隐含使用this指针。

this指针具有如下形式的缺省说明：

类名

Stu *const this;

即 this 指针里的地址是一个常量

```
class S{public :  
    char *strp;  
    int length;  
    void Ini(char *s);  
    void Print(void);  
    void Copy(S &s);  
};  
void main(void)  
{  
    S s1,s2;  
    s1.Ini("China");  
    s2.Ini("");  
    s1.Print();  
    s2.Copy(s1);  
    s2.Print();  
    s1.Copy(s1);  
}
```

```
void S::Ini(char *s)  
{  
    length=strlen(s);  
    strp=new char[length+1];  
    strcpy(strp, s);  
}  
void S::Print(void)  
{  
    cout<<strp<<endl; }  
void S::Copy(S &s)  
{  
    if(strp)  
        delete [ ]strp;  
    length= s.length;  
    strp=new char[length+1];  
    strcpy(strp, s.strp);  
}
```

```
void S::Ini(char *s)
{
    length=strlen(s);
    strp=new char[length+1];
    strcpy(strp, s);
}
```

求长度

对象动态开辟空间

空间赋值

```
void S::Copy(S &s)
{
    if(strp)
        delete [ ]strp;
    length= s.length;
    strp=new char[length+1];
    strcpy(strp, s.strp);
}
```

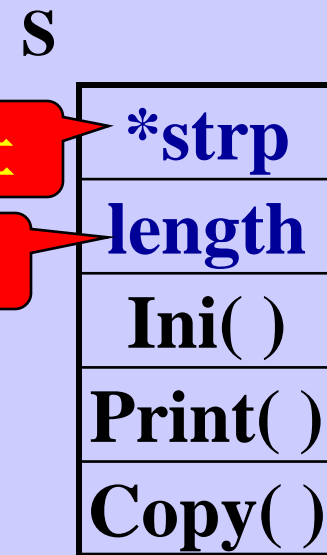
s2.Copy(s1);

s1.Ini("China");

length=5

strp → C h i n a \0

strp → C h i n a \0



s1.Copy(s1); **length=5**

strp →

随 机	随 机	随 机	随 机	随 机	随 机
--------	--------	--------	--------	--------	--------

```
void S::Copy(S &s)
```

```
{ if(strp)
```

```
    delete [ ]strp;
```

失去了原来的内容，
不能正确复制

```
    length= s.length;
```

```
    strp=new char[length+1];
```

```
    strcpy(strp, s.strp);
```

```
}
```

```
void S::Copy(S &s)
```

```
{ if(&s==this)
```

判断是否是自身复制

```
    cout<<“不能复制本身\n”;
```

```
    else
```

```
    {   if(strp)
```

```
        s2.Copy(s1);
```

```
        delete [ ]strp;
```

```
        s1.Copy(s1);
```

```
        length= s.length;
```

```
        strp=new char[length+1];
```

```
        strcpy(strp, s.strp);
```

```
    }
```

this为正在调用该函数的对象的地址

```
if(s==*this)
```

```
}
```

第十章 构造函数和析构函数

构造函数和析构函数是在类体中说明的两种特殊的成员函数。

构造函数是在创建对象时，使用给定的值来将对象初始化。

析构函数的功能正好相反，是在系统释放对象前，对对象做一些善后工作。

构造函数可以带参数、可以重载，同时没有返回值。

构造函数是类的成员函数，系统约定构造函数名必须与类名相同。构造函数提供了初始化对象的一种简单的方法。

```
class A{
    float x,y;
public:
    A(float a,float b){ x=a; y=b;}//构造函数，初始化对象
    float Sum(void) { return x+y; }
    void Set(float a,float b) { x=a; y=b;}
    Print(void) { cout<<"x="<<x<<"\t"<<"y="<<y<<endl;}
};

void main(void)
{
    A a1(2.0, 3.0);//定义时调用构造函数初始化
    A a2(1.0,2.0);
    a2.Set(10.0, 20.0); //利用成员函数重新为对象赋值
    a1.Print();
    a2.Print();
}
```

对构造函数，说明以下几点：

1. 构造函数的函数名必须与类名相同。构造函数的主要作用是完成初始化对象的数据成员以及其它的初始化工作。
2. 在定义构造函数时，不能指定函数返回值的类型，也不能指定为void类型。
3. 一个类可以定义若干个构造函数。当定义多个构造函数时，必须满足函数重载的原则。

4.构造函数可以指定参数的缺省值。

5.若定义的要说明该类的对象时，构造函数必须是**公有的成员函数**。如果定义的类型仅用于派生其它类时，则可将构造函数定义为**保护的成员函数**。

由于构造函数属于类的成员函数，它对私有数据成员、保护的数据成员和公有的数据成员**均能进行初始化**。


```

class A{
    float x,y;
public:
    A(float a, float b=10)    { x=a; y=b; }
    A()    { x=0; y=0;}
    void Print(void) {cout<<x<<'\\t'<<y<<endl; }
};

void main(void)
{
    A a1, a2(20.0), a3(3.0, 7.0);
    a1.Print();
    a2.Print();
    a3.Print();
}

```

帶缺省参数的构造函数

不帶参数的构造函数

每一个对象必须要有相应的构造函数

0	0
20	10
3	7

每一个对象必须要有相应的构造函数

若没有显式定义构造函数，
系统默认缺省的构造函数。

```
class A{  
    float x,y;
```

对象开辟了空间，但没有初始化

```
public:
```

```
    A() {}
```

隐含的缺省的构造函数

```
    void Print(void) {cout<<x<<'\t'<<y<<endl; }
```

```
};
```

只允许这样定义对象

```
A a1, a2;
```

对局部对象，静态对象，全局对象的初始化
对于局部对象，每次定义对象时，都要调用构造函数。

对于静态对象，是在首次定义对象时，调用构造函数的，且由于对象一直存在，只调用一次构造函数。

对于全局对象，是在main函数执行之前调用构造函数的。

```
class A
```

```
{ int x,y;
```

```
public:
```

```
A(int a){ x=a; cout<<“1\n”;} 
```

```
A(int a, int b) { x=a,y=b;cout<<“2\n”;} 
```

```
};
```

```
A a1(3);
```

1

```
void f(void) { A b(2,3);}
```

2

```
void main(void)
```

2

```
{ A a2(4,5);
```

2

```
f(); f();
```

```
}
```

```

class A{
    float x,y;
public:
    A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
    A(){ x=0; y=0; cout<<"初始化静态局部对象\n";}
    A(float a){ x=a; y=0; cout<<"初始化全局对象\n"; }
    void Print(void){ cout<<x<<"\t"<<y<<endl; }
};

```

A a0(100.0);//定义全局对象

初始化全局对象

void f(void)

进入main函数

```

{ cout<<" 进入f()函数\n";A a2(1,2);
    static A a3; //初始化局部静态对象
}

```

初始化自动局部对象

进入f()函数

}

初始化自动局部对象

void main(void)

初始化局部静态变量

```

{ cout<<"进入main函数\n";

```

进入f()函数

A a1(3.0, 7.0);//定义局部自动对象

初始化自动局部对象

```

f(); f(); }

```

缺省的构造函数

在定义类时，若没有定义类的构造函数，则编译器**自动**产生一个缺省的构造函数，其格式为：

```
className::className() { }
```

缺省的构造函数并不对所产生对象的数据成员赋初值；即**新产生对象的数据成员的值是不确定的。**

```

class A{
    float x,y;
public:
    A(){ }//缺省的构造函数，编译器自动产生,可以不写
    float Sum(void) { return x+y; }
    void Set(float a,float b) { x=a; y=b;}
    void Print(void) { cout<<"x="<<x<<"\t"<<"y="<<y<<endl; }
};

void main(void)
{
    A a1,a2;//产生对象时，自动调用缺省的构造函数，不赋值
    a1.Set (2.0,4.0);
    cout<<"a1: ";
    a1.Print ();
    cout<<"a1.sum="<<a1.Sum ()<<endl;
    a2.Print();//打印随机值
}

```

关于缺省的构造函数，说明以下几点：

- 1、在定义类时，只要显式定义了一个类的构造函数，则编译器就不产生缺省的构造函数
 - 2、所有的对象在定义时，必须调用构造函数
- 不存在没有构造函数的对象！


```
class A{
```

```
    float x,y;
```

显式定义了构造函数，不
产生缺省的构造函数

```
public:
```

```
    A(float a,float b)    {    x=a; y=b; }
```

```
    void Print(void){    cout<<x<<'\\t'<<y<<endl; }
```

```
};
```

```
void main(void)
```

```
{    A a1;
```

error,定义时，没有
构造函数可供调用

```
    A a2(3.0,30.0);
```

```
}
```

3、在类中，若定义了没有参数的构造函数，或各参数均有缺省值的构造函数也称为缺省的构造函数，缺省的构造函数只能有一个。

4、产生对象时，系统必定要调用构造函数。所以任一对象的构造函数必须唯一。

```
class A{  
    float x,y;  
public:  
    A(float a=10,float b=20){    x=a; y=b; }  
    A(){ }  
    void Print(void){    cout<<x<<'\t'<<y<<endl; }  
};  
void main(void)  
{  
    A a1;  
    A a2(3.0,30.0);  
}
```

两个函数均为缺省的构造函数

两个构造函数均可供调用，构造函数不唯一

构造函数与new运算符

可以使用new运算符来动态地建立对象。建立时要自动调用构造函数，以便完成初始化对象的数据成员。最后返回这个动态对象的起始地址。

用new运算符产生的动态对象，在不再使用这种对象时，必须用delete运算符来释放对象所占用的存储空间。

用new建立类的对象时，可以使用参数初始化动态空间。

```

class A{
    float x,y;
public:
    A(float a, float b)    {    x=a;y=b;    }
    A()    {    x=0; y=0;    }
    void Print(void)    {    cout<<x<<'\\t'<<y<<endl;    }
};

void main(void)
{
    A *pa1,*pa2;
    pa1=new A(3.0, 5.0);//用new动态开辟对象空间，初始化
    pa2=new A;//用new动态开辟空间，调用构造函数初始化
    pa1->Print();
    pa2->Print();
    delete pa1; //用delete释放空间
    delete pa2; //用delete释放空间
}

```

3 5

0 0

析构函数

析构函数的作用与构造函数正好相反，是在对象的生命期结束时，释放系统为对象所分配的空间，即要撤消一个对象。

析构函数也是类的成员函数，定义析构函数的格式为：

```
ClassName::~~ClassName( )
```

```
{
```

```
.....
```

```
//      函数体;
```

```
}
```

析构函数的特点如下：

- 1、析构函数是**成员函数**，函数体可写在类体内，也可写在类体外。
- 2、析构函数是一个特殊的成员函数，函数名必须与类名相同，并在其**前面加上字符“~”**，以便和构造函数名相区别。
- 3、析构函数**不能带有任何参数，不能有返回值，不指定函数类型。**

4、一个类中，只能定义一个析构函数，析构函数不允许重载。

5、析构函数是在撤消对象时由系统自动调用的。

在程序的执行过程中，当遇到某一对象的生存期结束时，系统自动调用析构函数，然后再收回为对象分配的存储空间。


```

class A{
    float x,y;
public:
    A(float a,float b)
{
    x=a;y=b;cout<<"调用非缺省的构造函数\n";}
    A() { x=0; y=0; cout<<"调用缺省的构造函数\n" ;}
    ~A() { cout<<"调用析构函数\n";}
    void Print(void) { cout<<x<<"\t"<<y<<endl; }
};

void main(void)
{
    A a1;
    A a2(3.0,30.0);
    cout<<"退出主函数\n";
}

```

调用缺省的构造函数

调用非缺省的构造函数

退出主函数

调用析构函数

调用析构函数

在程序的执行过程中，对象如果用**new**运算符开辟了空间，则在类中**应该**定义一个析构函数，并在析构函数中使用**delete**删除由**new**分配的内存空间。因为在撤消对象时，系统自动收回为对象所分配的存储空间，而不能自动收回由**new**分配的动态存储空间。

```

class Str{
    char *Sp;  int Length;
public:
    Str(char *string)
    {  if(string){      Length=strlen(string);
        Sp=new char[Length+1];
        strcpy(Sp,string);
    }
    else    Sp=0;
}
void Show(void){ cout<<Sp<<endl; }
~Str(){  if(Sp)      delete []Sp; }
};

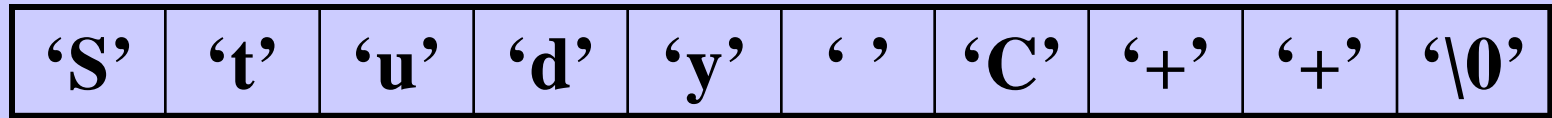
void main(void)
{
    Str s1("Study C++");
    s1.Show();
}

```

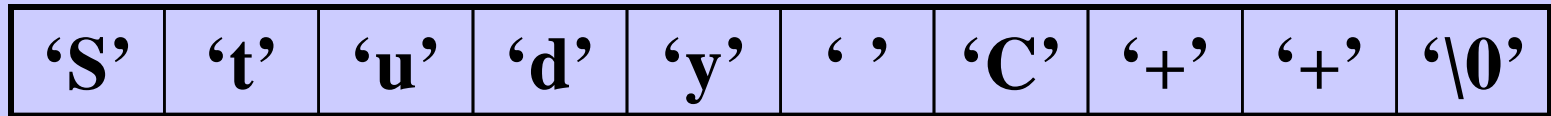
在构造函数中将成员数据指针指向动态开辟的内存

用初值为开辟的内存赋值

析构函数，当释放对象时收回用new开辟的空间



string



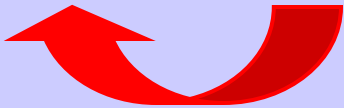
Sp

new开辟的空间

```
Length=strlen(string);
```

```
Sp=new char[Length+1];
```

```
strcpy(Sp,string);
```



可以用**new**运算符为对象分配存储空间，如：

```
A *p;
```

```
p=new A;
```

这时必须用**delete**才能释放这一空间。

```
delete p;
```

用**new**运算符为对象分配动态存储空间时，调用了构造函数，用**delete**删除这个空间时，调用了析构函数。当使用运算符**delete**删除一个由**new**动态产生的对象时，它首先调用该对象的析构函数，然后再释放这个对象占用的内存空间。

```

class A{
    float x,y;

public:
    A(float a, float b){ x=a; y=b; cout<<"调用了构造函数\n";}
    void Print(void){ cout<<x<<"\t"<<y<<endl; }
    ~A() { cout<<"调用了析构函数\n"; }
};

void main(void)
{ cout<<"进入main()函数\n";
  A *pa1;
  pa1=new A(3.0, 5.0);//调用构造函数
  pa1->Print();
  delete pa1; //调用析构函数
  cout<<"退出main()函数\n";
}

```

进入main()函数

调用了构造函数

3 5

调用了析构函数

退出main()函数

不同存储类型的对象调用构造函数及析构函数

- 1、**对于全局定义的对象**（在函数外定义的对象），在程序开始执行时，调用构造函数；到程序结束时，调用析构函数。
- 2、**对于局部定义的对象**（在函数内定义的对象）当程序执行到定义对象的地方时，调用构造函数；在退出对象的作用域时，调用析构函数。
- 3、**用static定义的局部对象**，在首次到达对象的定义时调用构造函数；到程序结束时，调用析构函数

4、对于用new运算符动态生成的对象，在产生对象时调用构造函数，只有使用delete运算符来释放对象时，才调用析构函数。若不使用delete来撤消动态生成的对象，程序结束时，对象仍存在，并占用相应的存储空间，即系统不能自动地调用析构函数来撤消动态生成的对象。


```

class A{
    float x,y;
public:
    A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
    A(){ x=0; y=0; cout<<"初始化静态局部对象\n";}
    A(float a){ x=a; y=0; cout<<"初始化全局对象\n"; }
    ~A(){ cout<<"调用析构函数" <<endl; }
};

A a0(100.0); //定义全局对象
void f(void)
{ cout<<" 进入f()函数\n";
    A ab(10.0, 20.0); //定义局部自动对象
    static A a3; //初始化局部静态对象
}

void main(void)
{ cout<<"进入main函数\n";
  f(); f(); }

```

初始化全局对象
 进入main函数
 进入f()函数
 初始化自动局部对象
 初始化静态局部对象
 调用析构函数
 进入f()函数
 初始化自动局部对象
 调用析构函数
 调用析构函数
 调用析构函数

举例：建立一个类NUM，求指定数据范围内的所有素数。

如： 定义类NUM的对象test，查找范围为100~200，正确的输出结果：

num=21

101 103 107 109 113

127 131

动态构造及析构对象数组

用**new**运算符来动态生成对象数组时，自动调用构造函数，而用**delete**运算符来**释放p1所指向的对象数组**占用的存储空间时，在指针变量的前面必须加上**[]**，才能将数组元素所占用的空间全部释放。否则，只释放第**0**个元素所占用的空间。

```
pa1=new A[3];
```

```
....
```

```
delete [ ]pa1;
```

```

class A{
    float x,y;
public:
    A(float a=0, float b=0){x=a; y=b;cout<<"调用了构造函数\n";}
    void Print(void){ cout<<x<<'\\t'<<y<<endl; }
    ~A() { cout<<"调用了析构函数\n"; }
};

void main(void)
{ cout<<"进入main()函数\n";
  A *pa1;
  pa1=new A[3];//开辟数组空间
      cout<<"\n完成开辟数组空间\n\n";
  delete [ ]pa1; //必须用[]删除开辟的空间
  cout<<"退出main()函数\n";
}

```

进入main()函数

调用了构造函数

调用了构造函数

调用了构造函数

完成开辟数组空间

调用了析构函数

调用了析构函数

调用了析构函数

退出main()函数⁹²

缺省的析构函数

若在类的定义中没有显式地定义析构函数时，则编译器自动地产生一个缺省的析构函数，其格式为：

```
ClassName::~~ClassName() { };
```

任何对象都必须有构造函数和析构函数，但在撤消对象时，要释放对象的数据成员用**new**运算符分配的动态空间时，必须显式地定义析构函数。

实现类型转换的构造函数

同类型的对象可以相互赋值，相当于类中的数据成员相互赋值；

如果直接将数据赋给对象，所赋入的数据需要强制类型转换，这种转换需要调用构造函数。

```

class A{
    float x,y;
public:
    A(float a,float b) {x=a;y=b;cout<<"调用构造函数\n";}
    ~A()      {      cout<<"调用析构函数\n";}
    void Print(void) {  cout<<x<<"\t"<<y<<endl; }
};

void main(void)
{ A  a1(1.0, 10.0);
  a1.Print();
  a1=A(3.0 , 30.0);
  a1.Print();
  cout<<"退出主函数\n";
}

```

调用构造函数
1 10
调用构造函数
调用析构函数
3 30
退出主函数
调用析构函数

产生临时对象，
初始化并赋值后
立即释放

注意：当构造函数只有一个参数时，可以用=强制赋值。

```
class B{
    float x;
public:  B(float a) {x=a; cout<<"调用构造函数\n";}
        ~B() { cout<<"调用析构函数\n";}
        void Print(void) { cout<<x<<endl; }
};

void main(void)
{B b1(1.0);b1.Print();
  B b2=100;
  b2.Print();
  b1=10;      b1.Print();
  cout<<"退出主函数\n";
}
```

单参数可以这样赋值

b1=B(10)

产生一个
临时对象

调用构造函数
1
调用构造函数
100
调用构造函数
调用析构函数
10
退出主函数
调用析构函数
调用析构函数

完成拷贝功能的构造函数

可以在定义一个对象的时候用另一个对象为其初始化，**即构造函数的参数是另一个对象的引用**，这种构造函数常为完成拷贝功能的构造函数。

完成拷贝功能的构造函数的一般格式为：

```
ClassName::ClassName(ClassName &<变量名>)
```

```
{    .....
```

```
// 函数体完成对应数据成员的赋值
```

```
}
```

```
class A{
```

```
    float x,y;
```

```
public:
```

```
    A(float a=0, float b=0){x=a; y=b;}
```

```
    A(A &a)
```

形参必须是同类型对象的引用

```
    { x=a.x; y=a.y;}
```

```
};
```

```
void main(void)
```

```
{  A  a1(1.0,2.0);
```

```
    A  a2(a1);
```

实参是同类型的对象

```
}
```

```

class A{
    float  x,y;

public:
    A(float a=0, float b=0){x=a; y=b;cout<<"调用了构造函数\n";}
    A(A &a) { x=a.x;y=a.y;cout<<"调用了完成拷贝功能的构造函数\n"; }
    void Print(void){ cout<<x<<"\t"<<y<<endl; }
    ~A() { cout<<"调用了析构函数\n"; }
};

void main(void)
{
    A  a1(1.0,2.0);
    A  a2(a1);
    a1.Print();
    a2.Print();
}

```

用已有的对象中的数据为新创建的对象赋值

调用了构造函数

调用了完成拷贝功能的构造函数

1 2

1 2

调用了析构函数

调用了析构函数

如果没有定义完成拷贝功能的构造函数，编译器自动生成一个隐含的完成拷贝功能的构造函数，依次完成类中对应数据成员的拷贝。

```
A::A(A &a)
```

```
{
```

```
    x=a.x;
```

```
    y=a.y;
```

```
}
```

隐含的构造函数

```

class A{
    float  x,y;
public:
    A(float a=0, float b=0){x=a; y=b; cout<<"调用了构造函数\n";}
    void Print(void){ cout<<x<<"\t"<<y<<endl; }
    ~A() { cout<<"调用了析构函数\n"; }
};

```

调用了构造函数

```

void main(void)

```

```

{  A  a1(1.0,2.0);

```

隐含了拷贝
的构造函数

1 2

```

    A  a2(a1);

```

1 2

```

    A  a3=a1;//可以这样赋值

```

1 2

```

    a1.Print();

```

调用了析构函数

```

    a2.Print();

```

调用了析构函数

```

    a3.Print();

```

调用了析构函数

```

}

```

由编译器为每个类产生的这种隐含的完成拷贝功能的构造函数，依次完成类中对应数据成员的拷贝。

但是，当类中的数据成员中使用new运算符，动态地申请存储空间进行赋初值时，必须在类中显式地定义一个完成拷贝功能的构造函数，以便正确实现数据成员的复制。

```

class Str{
    int Length;  char *Sp;
public:
    Str(char *string){
        if(string){Length=strlen(string);
            Sp=new char[Length+1];
            strcpy(Sp,string);      }
        else      Sp=0;
    }
    void Show(void){cout<<Sp<<endl;}
    ~Str(){if(Sp) delete []Sp;  }
};

void main(void)
{
    Str s1("Study C++");
    Str s2(s1);
    s1.Show ();   s2.Show ();
}

```

隐含的拷贝构造函数为:

```

Str::Str(Str &s)
{
    Length=s.Length;
    Sp=s.Sp;
}

```

s2.Sp



“Study C++”



s1.Sp

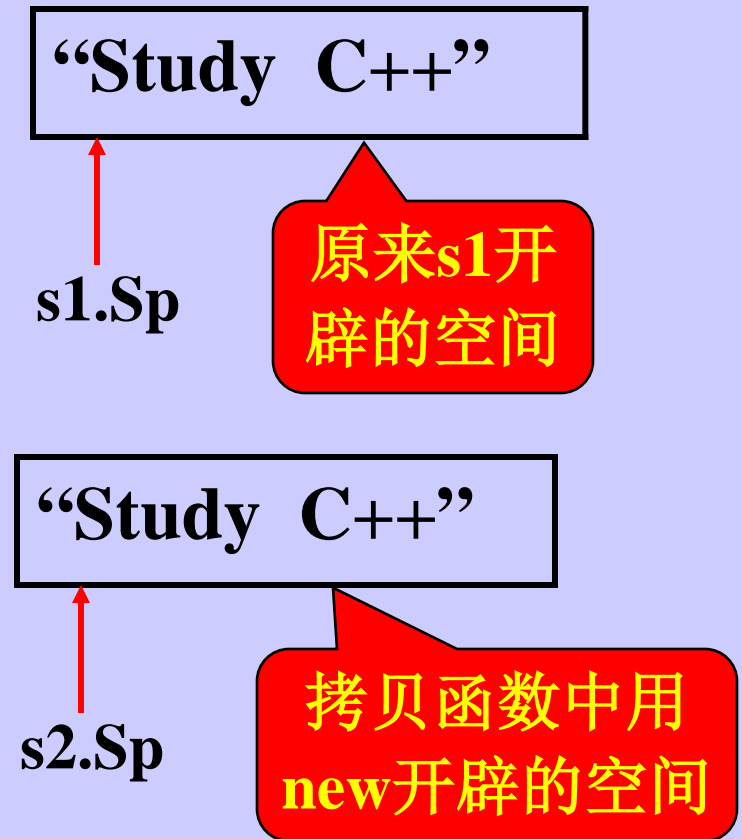
new开辟
的空间

同一空间释放两次，造成运行错误。

在这种情况下，**必须**要定义完成拷贝功能的构造函数。

```
Str::Str(Str &s){  
    if(s.Sp){  
        Length=s.Length ;  
        Sp=new char[Length+1];  
        strcpy(Sp,s.Sp);  
    }  
    else Sp=0;  
}
```

Str s2(s1);



构造函数与对象成员

```
class B{
```

```
....
```

```
};
```

```
class A{
```

```
int x , y;
```

```
B b1,b2;
```

```
};
```

在类A中包含
类B的对象

对类A的对象初始化的同时还要对其成员数据类B的对象进行初始化，所以，类A的构造函数中要调用类B的构造函数。

```

class A {
    float x,y;
public:
    A(int a,int b)    {    x=a;y=b;    }
    void Show(){ cout<< "x="<<x<<"\t"<<"y="<<y<<"\n"; }
};

class C{
    float z;
    A a1;//类C的数据成员为类A 的对象a1
public:
    C(int a,int b,int c):a1(b,c) {z=a;}//类C的对象初始化
    void Show(){cout<< "z="<<a<<"\n";    a1.Show();}
};

void main(void)
{
    C c1(1, 2, 3 );    //对类C的对象初始化
    c1.Show();
}

```

利用类A的构造函数对类A的对象初始化

在类C中调用类A的成员函数

A

x
y
A()
Show()

C

a1

z
a1.x
a1.y
a1.A()
a1.Show()
C()
Show()

a1(b, c)

```
ClassName::ClassName(args):c1(args1),...,cn(agsn)
{
..... //对其它成员的初始化
}
```

初始化对象成员的参数（实参）可以是表达式。
也可以仅对部分对象成员进行初始化。

```

class A {
    float x,y;
public: A(int a,int b) {      x=a;y=b;      }
    void Show(){ cout<< "x="<<x<<"\t"<<"y="<<y<<"\n"; }
};

class B{
    float x1,y1;
public:      B(int a, int b){  x1=a; y1=b; }
    void Show(){ cout<<"x1="<<x1<<"\t"<<"y="<<y<<"\n";}
};

class C{
    float z;  A a1; B b1;
public: C(int a,int b,int c,int d, int e):a1(a+b, c) ,b1(a,d) {z=e;}
    void Show(){cout<< "z="<<a<<"\n";  a1.Show();b1.Show();}
};

void main(void)
{
    C c1(1, 2, 3 ,4,5);    //对类C的对象初始化 }

```

对象初始化的参数可以是表达式

对对象成员的构造函数的调用顺序取决于这些对象成员在类中说明的顺序，与它们在成员初始化列表中的顺序无关。

当建立类ClassName的对象时，先调用各个对象成员的构造函数，初始化相应的对象成员，然后才执行类ClassName的构造函数，初始化类ClassName中的其它成员。析构函数的调用顺序与构造函数正好相反。

```

class A{
    float x;
public:
    A(int a){ x=a; cout<<“调用了A的构造函数\n”;}
    ~A(){cout<<“调用了A的析构函数\n”;}
};

```

```

class B{
    float y;
public:
    B(int a){ y=a; cout<<“调用了B的构造函数\n”;}
    ~B(){cout<<“调用了B的析构函数\n”;}
};

```

```

class C{
    float z; B b1; A a1;
public:
    C(int a,int b,int c): a1(a),b1(b){z=c;cout<<“调用了C的构造函数\n”;}
    ~C(){cout<<“调用了C的析构函数\n”;}
};

```

```

void main(void)
{ C c1(1,2,3); }

```

调用了B的构造函数
 调用了A的构造函数
 调用了C的构造函数
 调用了C的析构函数
 调用了A的析构函数
 调用了B的析构函数

第十一章 继承和派生类

继承性是面向对象程序设计中最重要机制。这种机制提供了无限重复利用程序资源的一种途径。通过C++语言中的继承机制，可以扩充和完善旧的程序设计以适应新的需求。这样不仅可以节省程序开发的时间和资源，并且为未来程序增添了新的资源。

```
class Student
```

```
{ int num;
```

```
char name[30];
```

```
char sex;
```

```
public:
```

```
void display( ) //对成员函数display的定义
```

```
{cout<<"num: "<<num<<endl;
```

```
cout<<"name: "<<name<<endl;
```

```
cout<<"sex: "<<sex<<endl; }
```

```
};
```

class Student1

```
{  int num;      //此行原来已有  
    char name[20];  //此行原来已有  
    char sex;      //此行原来已有
```

int age;

char addr [20] ;

public:

```
void display( ) ;    //此行原来已有
```

```
{cout<<"num: "<<num<<endl;  //此行原来已有
```

```
cout<<"name: "<<name<<endl; //此行原来已有
```

```
cout<<"sex: "<<sex<<endl;    //此行原来已有
```

cout<<"age: "<<age<<endl;

cout<<"address: "<<addr<<endl;}

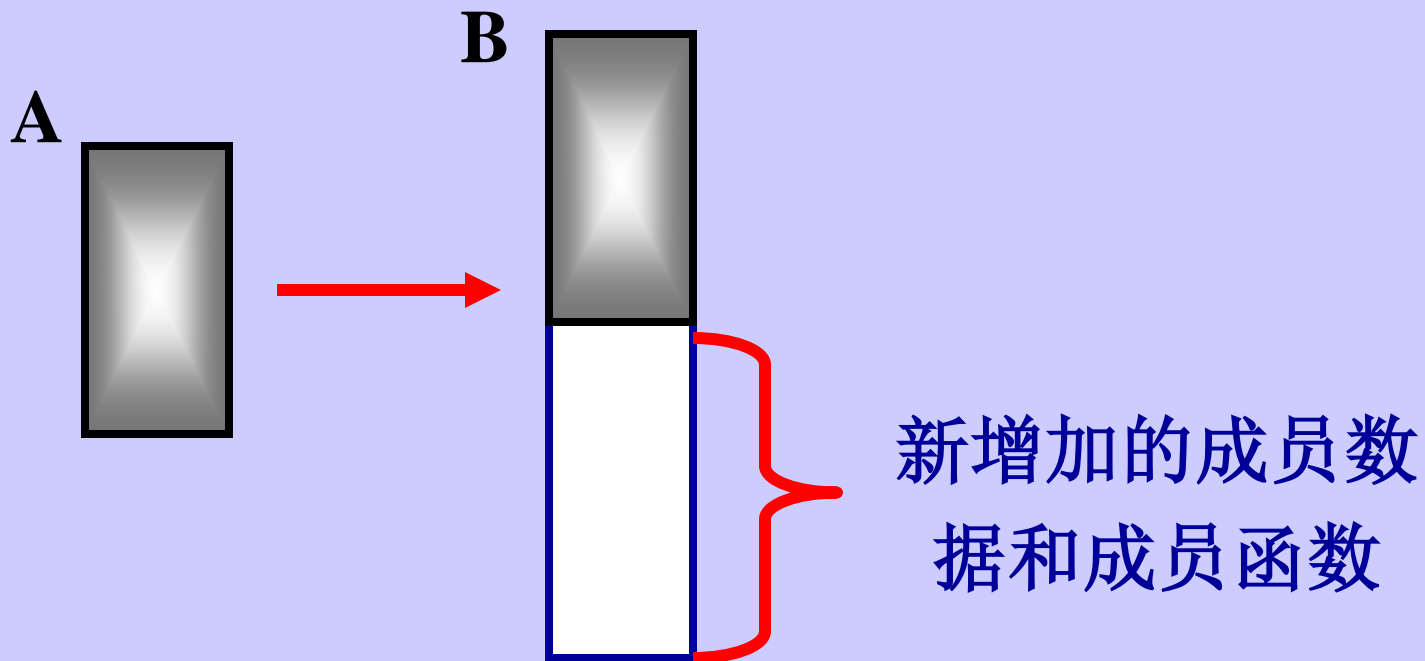
```
};
```

利用原来定义的类Student作为基础，再加上新的内容即可，以减少重复的工作量。 C++提供的继承机制就是为了解决这个问题。

在C++中所谓“继承”就是在一个已存在的类的基础上建立一个新的类。已存在的类称为“基类(base class)”或“父类(father class)”。新建立的类称为“派生类(derived class)”或“子类(son class)”。

```
class Student1: public Student//声明基类是Student
{private:
    int age; //新增加的数据成员
    string addr; //新增加的数据成员
public:
    void display_1() //新增加的成员函数
    { cout<<"age: "<<age<<endl;
      cout<<"address: "<<addr<<endl;
    }
};
```

类A派生类B：类A为基类，类B为派生类。



在C++语言中，一个派生类可以从一个基类派生，也可以从多个基类派生。从一个基类派生的继承称为单继承；从多个基类派生的继承称为多继承。

通过继承机制，可以利用已有的数据类型来定义新的数据类型。所定义的新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员。我们称已存在的用来派生新类的类为基类，又称为父类。由已存在的类派生出的新类称为派生类，又称为子类。

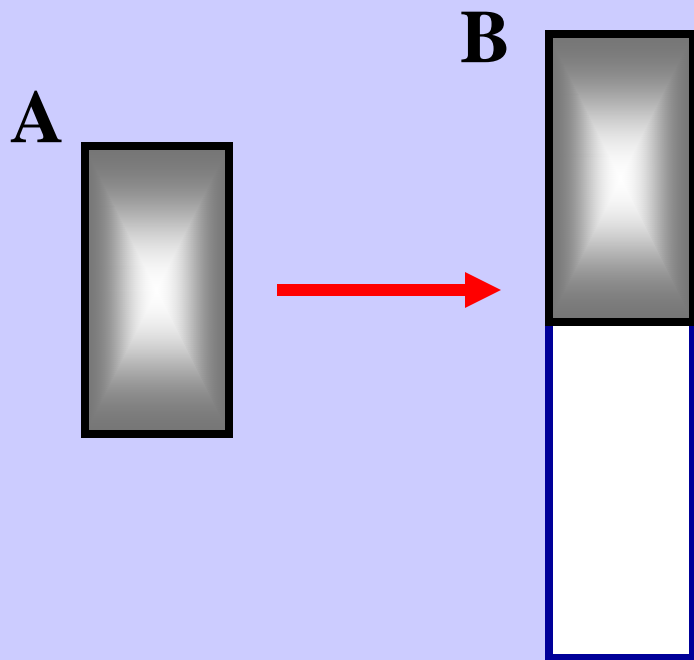
在建立派生类的过程中，基类不会做任何改变，派生类则除了继承基类的所有可引用的成员变量和成员函数外，还可另外定义本身的成员变量和处理这些变量的函数，由于派生类可继承基类的成员变量和成员函数，因此在基类中定义好的数据和函数等的程序代码可重复使用，这样可以提高程序的可靠性。

当从已有的类中派生出新的类时，可以对派生类做以下几种变化：

- 1、可以继承基类的成员数据或成员函数。
- 2、可以增加新的成员变量。
- 3、可以增加新的成员函数。
- 4、可以重新定义已有的成员函数。
- 5、可以改变现有的成员属性。

在C++中有二种继承：单一继承和多重继承。当一个派生类仅由一个基类派生时，称为单一继承；而当一个派生类由二个或更多个基类所派生时，称为多重继承。

类A派生类B：类A为基类，类B为派生类。



`class B: public A{...};`

`class B: protected A{...};`

`class B: private A{...};`

`class B: A{...};`

A为私有派生

但派生并不是简单的扩充，有可能改变基类的性质。

有三种派生方式：公有派生、保护派生、私有派生。

默认的是私有派生。

从一个基类派生一个类的一般格式为：

```
class ClassName:<Access>BaseClassName
```



```
{  
    private:  
        .....;    //私有成员说明  
    public:  
        .....;    //公有成员说明  
    protected:  
        .....;    //保护成员说明  
}
```

派生类中新增加的成员

public: 表示公有基类
private:表示私有基类(默认)
protected:表示保护基类

公有派生

公有派生，派生类中保持基类的成员特性

```
class ClassName: public BaseClassName
```

公有派生时，基类中所有成员在派生类中保持各个成员的访问权限。

基类成员属性	派生类中	派生类外
公有	可以引用	可以引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类: public: 在派生类和类外可以使用

protected: 在派生类中使用

private: 不能在派生类中使用

A

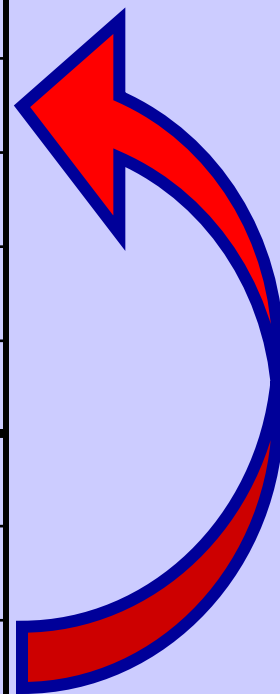
x(私有)
y(保护)
z(公有)
A()
Getx()
Gety()
ShowA()

public



B

x(私有)
y(保护)
z(公有)
A()
Getx()
Gety()
ShowA()
m(私有)
n(私有)
B()
Show()
Sum()



x在类B新增加的成员中不能直接调用

y在类B中可以调用

z在整个文件中可以调用

对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

```
class A {  int x;
protected:  int y;
public:    int z;

  A(int a,int b,int c){x=a;y=b;z=c;}//基类初始化
```

因为y是基类保护，所以在派生类中可以直接引用。而在类外不可直接引用。

```
int Getx(){return x;}    //返回x
```

```
int Gety(){return y;}    //返回y
```

```
void ShowA(){cout<< "x="<<x<<"\t"<<"y="<<y<<"\t"<<"z="<<z<<"\n";}
```

```
};
```

公有派生

```
class B:public A{
```

```
    int m,n;
```

对基类初始化

```
public:  B(int a,int b,int c,int d,int e):A(a,b,c){m=d;n=e;}
```

```
void Show(){cout<<"m="<<m<<"\t"<<"n="<<n<<"\n";
```

```
cout<<"x="<<Getx()<<"\t"<<"y="<<y<<"\t"<<"z="<<z<<"\n"; }
```

```
int Sum(){return ( Getx()+y+z+m+n);}
```

因为z是基类公有，所以在派生类中和类外均可直接引用。

```
};
```

因为x是基类私有，所以在派生类和类外中不能直接引用

```
void main(void)
```

```
{  B b1(1,2,3,4,5);
```

```
  b1.ShowA();    b1.Show();
```

```
  cout<< "Sum="<<b1.Sum()<<"\n";cout<<"x="<<b1.Getx()<<"\t";
```

```
  cout << "y=" <<b1.Gety()<<"\t";    cout << "z="<<b1.z<<"\n";}
```

私有派生

私有派生，派生类中基类公有和保护成员成为私有

```
class ClassName: private BaseClassName
```

私有派生时，基类中公有成员和保护成员在派生类中均变为私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

- 基类: public: (变为私有)在派生类中使用，类外不可使用
- protected: (变为私有) 在派生类中使用，类外不可使用
- private: 不能在派生类中和类外使用

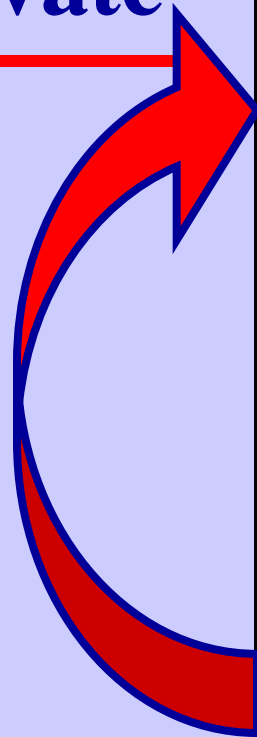
A

x(私有)
y(保护)
z(公有)
A()
Getx()
Gety()
ShowA()

B

x(私私有)
y(私有)
z(私有)
A()
Getx()
Gety()
ShowA()
m(私有)
n(私有)
B()
Show()
Sum()

private



均为私有
类B 外不
能引用

x在类B新增加的成员中
不能直接调用

y在类B中可以调用

z在类B中可以调用

对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

```
class A {  int x;
protected:  int y;
public:    int z;

A(int a,int b,int c){x=a;y=b;z=c;}//基类初始化
```

y是基类保护，所以在派生类中可以直接引用。而在类外不可直接引用。

```
int Getx(){return x;}    //返回x
```

```
int Gety(){return y;}    //返回y
```

```
void ShowA(){cout<< "x="<<x<<"\t"<<"y="<<y<<"\t"<<"z="<<z<<"\n";}
```

```
};
```

私有派生

```
class B:private A{
```

```
    int m,n;
```

对基类初始化

```
public:  B(int a,int b,int c,int d,int e):A(a,b,c){m=d;n=e;}
```

```
void Show(){cout<<"m="<<m<<"\t"<<"n="<<n<<"\n";
```

```
cout<<"x="<<Getx()<<"\t"<<"y="<<y<<"\t"<<"z="<<z<<"\n"; }
```

```
int Sum(){return ( Getx()+y+z+m+n);}
```

z是基类公有，私有派生变为私有，所以在派生类中可直接引用，而在类外不可。

```
};
```

因为x是基类私有，所以在派生类和类外中不能直接引用

```
void main(void)
```

```
{  B b1(1,2,3,4,5);  A a1(1,2,3); a1.ShowA();
```

```
  b1.ShowA();      b1.Show();
```

```
  cout<< "Sum="<<b1.Sum()<<"\n";cout<<"x="<<b1.Getx()<<"\t";
```

```
  cout << "y=" <<b1.Gety()<<"\t";      cout << "z="<<b1.z<<"\n";}
```

这些函数都是基类公有，在类外不可使用。

保护派生

保护派生，派生类中基类公有和保护成员降级使用

```
class ClassName: protected BaseClassName
```

保护派生时，基类中公有成员和保护成员在派生类中均变为保护的和私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

- 基类: public: (变为保护)在派生类中使用，类外不可使用
- protected: (变为私有) 在派生类中使用，类外不可使用
- private: 不能在派生类中和类外使用

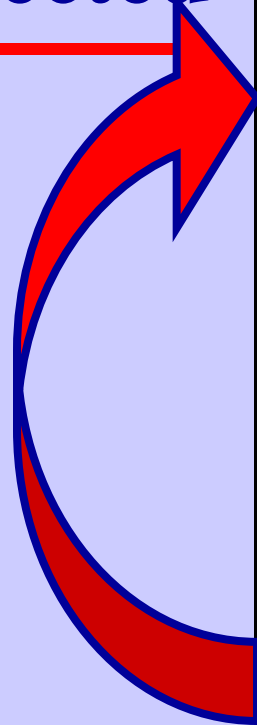
A

x(私有)
y(保护)
z(公有)
A()
Getx()
Gety()
ShowA()

B

x(私私有)
y(私有)
z(保护)
A()
Getx()
Gety()
ShowA()
m(私有)
n(私有)
B()
Show()
Sum()

protected



均为保护
类B 外不
能引用

x在类B新增加的成员
中不能直接调用

y在类B中可以调用

z在类B中可以调用

对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

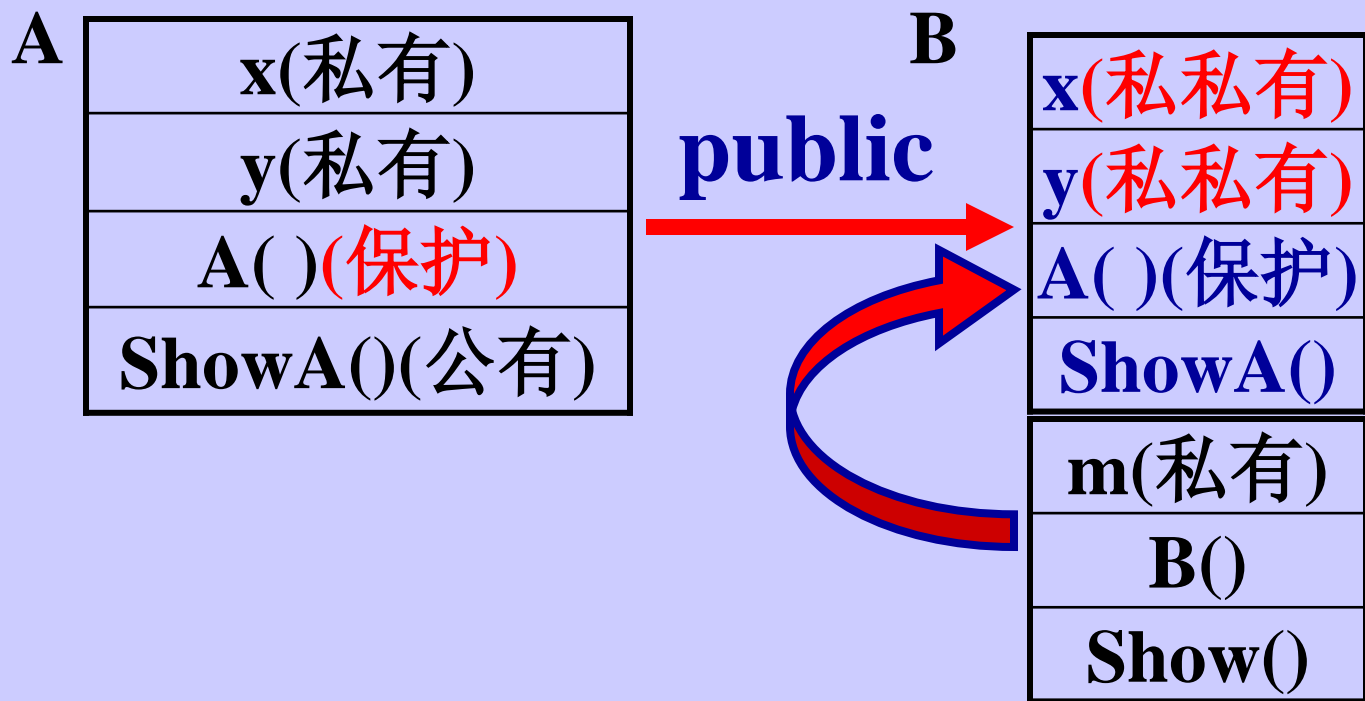
protected 成员是一种具有血缘关系内外有别的成员。它对派生类的对象而言，是公开成员，可以访问，对**血缘外部而言，与私有成员一样被隐蔽。**

抽象类与保护的成员函数

当定义了一个类，这个类只能用作基类来派生出新的类，而不能用这种类来定义对象时，称这种类为抽象类。当对某些特殊的对象要进行很好地封装时，需要定义抽象类。

将类的构造函数或析构函数的访问权限定义为保护的时，这种类为抽象类。

当把类中的构造函数或析构函数说明为**私有**的时，所定义类通常没有任何实用意义的，一般情况下，不能用它来产生对象，也不能用它来产生派生类。



在类**B**中不能定义**A**的对象

在任何时候都不能定义**A**的对象

但可以在初始化类**B**的对象时初始化**原类A**中的成员，
因为**A()**在类**B**中是可以被调用的。


```

class A { int x, y;
protected: A(int a,int b){x=a;y=b;}//基类初始化
public:
void ShowA(){cout<< "x="<<x<<"\t"<<"y="<<y<<"\n";}
};

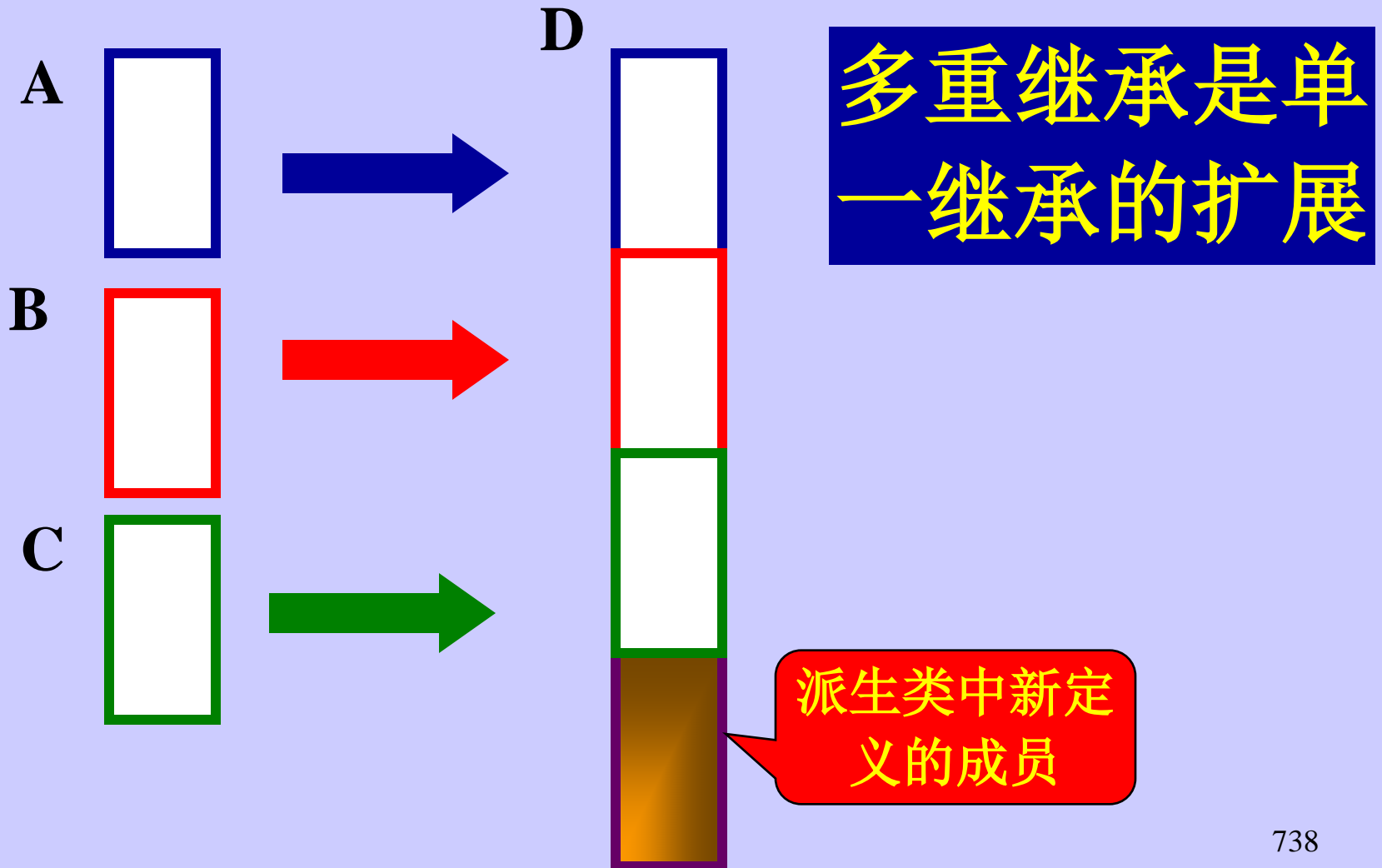
class B: public A{
    int m;
A a1; //在派生类中也不可以定义A的对象，实际上还是类外调用
public:
    B(int a,int b,int c):A(a,b)//可以在派生类中调用A的构造函数
    {m=c;}
    void Show(){    cout<<"m="<<m<<"\n"; ShowA(); }
};

void main(void)
{ B b1(1,2,3); //可以定义派生类对象
  b1.Show();
  A aa;    //不可定义A的对象
}

```

多重继承

可以用多个基类来派生一个类。



格式为:

继承方式

```
class 类名:<Access>类名1,..., <Access>类名n
{
    private:    ..... ;    //私有成员说明;
    public:     ..... ;    //公有成员说明;
    protected: ..... ;    //保护的成员说明;
};
```

```
class D: public A, protected B, private C
{    ....//派生类中新增加成员
};
```

```

class A{ int x1,y1;
public: A(int a,int b)    { x1=a; y1=b;    }
void ShowA(void){ cout<<"A.x="<<x1<<"\t"<<"A.y="<<y1<<endl; }
};

class B{int x2,y2;
public: B(int a,int b)    {x2=a; y2=b;    }
void ShowB(void){ cout<<"B.x="<<x2<<"\t"<<"B.y="<<y2<<endl; }
};

class C:public A,private B{
    int x,y;
public: C(int a,int b,int c,int d,int e,int f):A(a,b),B(c,d)    {x=e; y=f;    }
void ShowC(void){cout<<"C.x="<<x<<"\t"<<"C.y="<<y<<endl;
    ShowA();ShowB();    }
};

void main(void)
{
    C c(1,2,3,4,5,6);
    c.ShowC();
    c.ShowA ();
    c.ShowB ();
}

```

公有派生 (points to `public A`)

私有派生 (points to `private B`)

仍为公有 (points to `ShowA()`)

成为私有 (points to `ShowB()`)

非法，私有类外不可调用 (points to `c.ShowB ();`)

```

B b1(10,20);
b1.ShowB();

```

初始化基类成员

构造函数不能被继承,派生类的构造函数必须调用基类的构造函数来初始化基类成员基类子对象。

派生类构造函数的调用顺序如下:

基类的构造函数

子对象类的构造函数

派生类的构造函数

```
class B:public A{
```

```
    int y;
```

```
    A a1;
```

```
public:
```

```
    B(int a, int b):A(a),a1(3){y=b;}
```

```
    .....
```

```
};
```

基类的构造函数

子对象类的构造函数

派生类的构造函数

当撤销派生类对象时，析构函数的调用正好相反。

```
class Base1 { int x;
public: Base1(int a){x=a;cout<<"调用基类1的构造函数!\n"; }
      ~Base1( ){    cout<<"调用基类1的析构函数!\n";      }
};
```

```
class Base2 { int y;
public: Base2(int a){ y=a;cout<<"调用基类2的构造函数!\n"; }
      ~Base2( ){    cout<<"调用基类2的析构函数!\n";      }
};
```

先说明基类2

```
class Derived:public Base2, public Base1{
    int z;
public: Derived(int a,int b):Base1(a),Base2(20)
    {z=b; cout<<"调用派生类的构造函数!\n";}
    ~Derived( ){cout<<"调用派生类的析构函数!\n";}
};
```

调用基类2的构造函数

调用基类1的构造函数

调用派生类的构造函数

调用派生类的析构函数

调用基类1的析构函数

调用基类2的析构函数

```
void main(void)
{
    Derived c(100,200);
}
```

```
class Derived:public Base2, public Base1{
```

```
    int z;
```

基类子对象

```
    Base1 b1,b2;
```

```
public:
```

基类成员构造

基类子对象构造

```
    Derived(int a,int b):Base1(a),Base2(20), b1(200),b2(a+b)
```

```
    {z=b; cout<<"调用派生类的构造函数!\n";}
```

```
    ~Derived( ){cout<<"调用派生类的析构函数!\n";}
```

```
};
```

```
void main(void)
```

```
{    Derived  c(100,200);
```

```
}
```

基类成员构造用基类名，
基类子对象构造用对象名


```

class Base1 { int x;
public: Base1(int a){x=a;cout<<"调用基类1的构造函数!\n"; }
      ~Base1( ){    cout<<"调用基类1的析构函数!\n";      }
};

class Base2 { int y;
public: Base2(int a){ y=a;cout<<"调用基类2的构造函数!\n"; }
      ~Base2( ){    cout<<"调用基类2的析构函数!\n";      }
};

class Derived:public Base2, public Base1{
      int z; Base1 b1,b2;
public: Derived(int a,int b):Base1(a),Base2(20), b1(200),b2(a+b)
      {z=b; cout<<"调用派生类的构造函数!\n";}
      ~Derived( ){cout<<"调用派生类的析构函数!\n";}
};

void main(void)
{      Derived c(100,200);
}

```

调用基类2的构造函数
调用基类1的构造函数
调用基类1的构造函数
调用基类1的构造函数
调用派生类的构造函数
调用派生类的析构函数
调用基类1的析构函数
调用基类1的析构函数
调用基类1的析构函数
调用基类2的析构函数

说明基类1的对象
b1,b2

```

class Base1 {
    int x;
public: Base1(int a);
    ~Base1( );
};

class Base2 {
    int y;
public: Base2(int a);
    ~Base2( );
};

class Derived:public Base2, public Base1{
    int z; Base1 b1,b2;
public: Derived(int a,int b);
    ~Derived( );
};

```

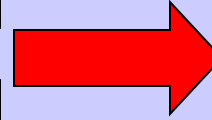
Base1

x
Base1()
~Base1()

Base2

y
Base2()
~Base2()

Derived

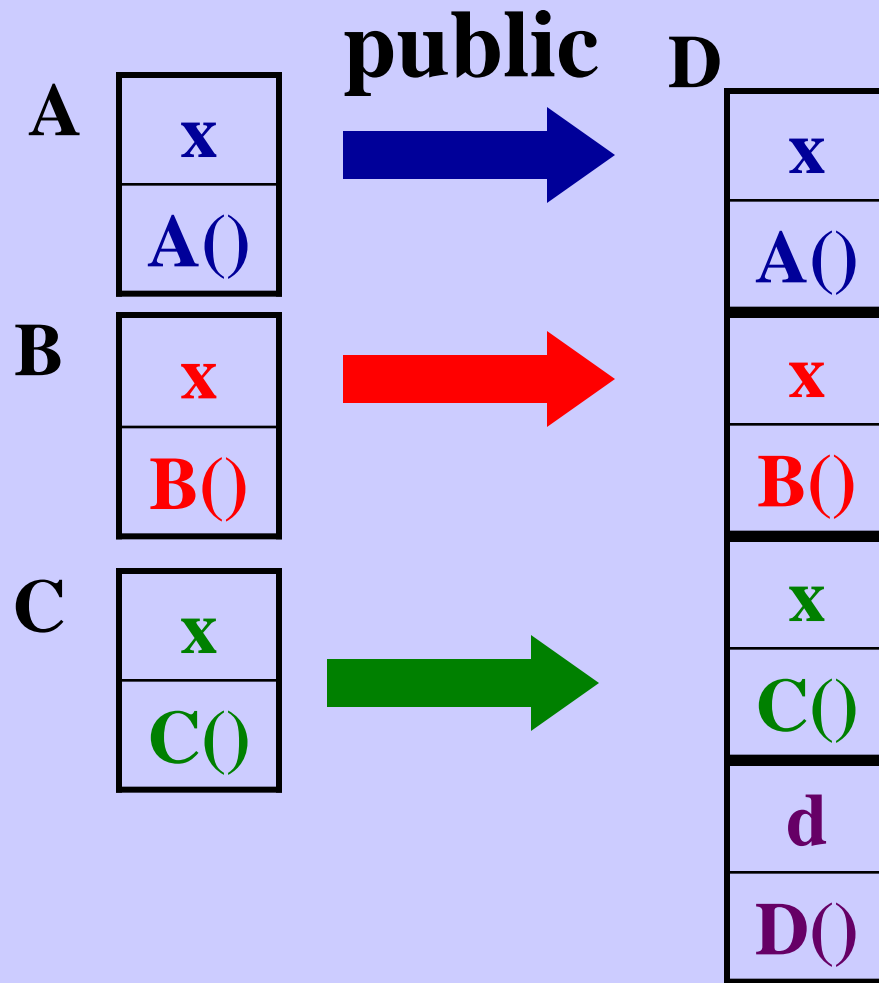


b1

b2

x
Base1()
~Base1()
y
Base2()
~Base2()
x
Base1()
~Base1()
x
Base1()
~Base1()
z
Derived()
~Derived

冲突



派生类对象

`D v;`

`v.x=5;`

产生了冲突

`A::v.x=5;`

用类作用符限定

```
class A{
public:  int x;
    void Show(){cout <<"x="<<x<<"\n";}
    A(int a=0){x=a;}
};
```

```
class B{
public:  int x;
    void Show(){cout <<"x="<<x<<"\n";}
    B(int a=0){x=a;}
};
```

```
class C:public A,public B{  int y;
public: void Setx(int a){ x=a;} //c1对象中有两个x成员
        void Sety(int b){y=b;}
        int Gety() {return y;}
};
```

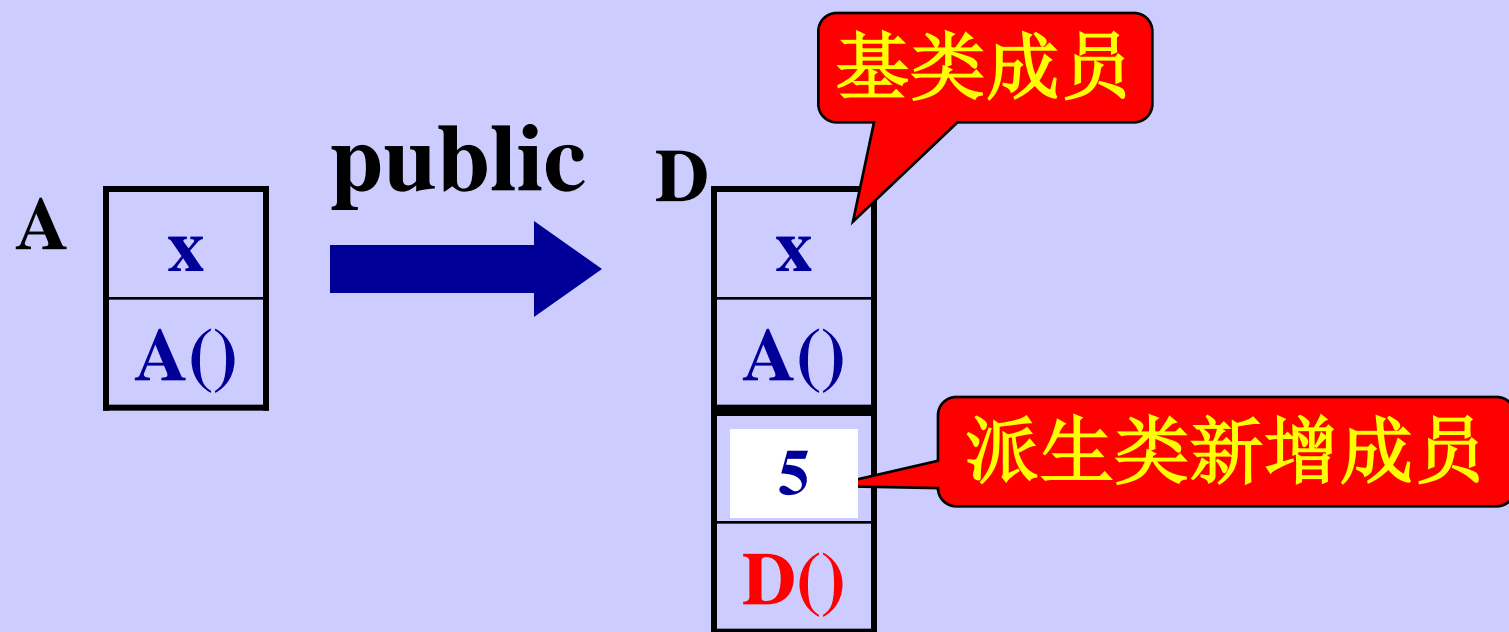
```
void main(void)
{
    C c1;  c1.Show(); //c1对象中有两个Show()函数
}
```

这时，可以利用类作用域符::来指明数据或函数的来源。

如: **A::x=a;**

c1.B::Show();

支配规则



`D v;`

`v.x=5;`

当派生类中新增加的数据或函数与基类中原有的同名时，若不加限制，则**优先调用派生类中的成员**。

```

class A{
public: int x;
        void Show(){cout <<"x="<<x<<"\n";}
};
class B{
public: int y;
        void Show(){cout <<"y="<<y<<"\n";}
};
class C:public A,public B{
public: int y; //类B和类C均有y的成员
};
void main(void)
{ C c1;      c1.x=100;
  c1.y=200;   //给派生类中的y赋值
  c1.B::y=300; //给基类B中的y赋值
  c1.A::Show();
  c1.B::Show(); //用作用域运算符限定调用的函数
  cout <<"y="<<c1.y<<"\n";      //输出派生类中的y值
  cout <<"y="<<c1.B::y<<"\n";  //输出基类B中的y值
}

```

当派生类中新增加的数据或函数与基类中原有的同名时，若不加限制，则**优先调用派生类中的成员**。

```

class A{
public: int x;
      A(int a=0) { x=a; }
};
class B{
public: int y;
      B(int a=0) { y=a; }
};
class C:public A{
      int z;   B  b1; A  a1;
public: C(int a,int b,int m):A(a),b1(b),a1(a+b){ z=m; }
      void Show() {
cout<<"x="<<x<<"\t";
cout<<"y="<<b1.y<<"\t";
cout<<"x="<<z<<"\n";
cout<<"a1.x="<<a1.x<<endl;    }
};
void main(void)
{
    C c1(100,200,500);
    c1.Show();}

```

从基类A中继承

x

在类C中新说明

z

在类C中新说明

Show()

在类C中新说明

a1.x

在类C中新说明

b1.y

C的对象所占空间

从基类A中继承

新说明类A对象中的x

基类与对象成员

任一基类在派生类中只能继承一次，否则，会造成成员名的冲突

若在派生类中，确实要有二个以上基类的成员，**则可用基类的二个对象作为派生类的成员。**

把一个类作为派生类的基类或把一个类的对象作为一个类的成员，在使用上是有区别的：在派生类中可直接使用基类的成员（访问权限允许的话），但要**使用对象成员的成员时，必须在对象名后加上成员运算符“.”和成员名。**

在平面上作两个点，连一直线，求直线的长度和直线中点的坐标。

基类为**Dot**，有两个公有数据成员，即平面上的坐标 (x,y) ，同时有构造函数及打印函数。

派生类为**Line**，有两个基类**Dot**对象，分别存放两点的坐标，同时，从基类继承了一个**Dot**数据，存放直线中点的坐标。

x
y
Dot(x,y)(构造)
Dot(&dot)(拷贝)
Show()

Dot的对象空间

**Line
对象
空间**

x(中点)	
y(中点)	
Dot(x,y)	
Dot(&dot)	
Show()	
d1	x
	y
	Dot(x,y)
	Dot(&dot)
	Show()
d2	x
	y
	Dot(x,y)
	Dot(&dot)
	Show()
Line(dot1,dot2)	
Showl()	

**从基类
继承**

**基类
对象**

```
class Dot{
```

```
public: float x,y;
```

```
Dot(float a=0,float b=0){ x=a; y=b;}
```

```
void Show(void){cout<<"x="<<x<<"\t"<<"y="<<y<<endl;}
```

```
};
```

```
class Line:public Dot{
```

```
Dot d1,d2;
```

```
public: Line(Dot dot1,Dot dot2):d1(dot1),d2(dot2)
```

```
{ x=(d1.x+d2.x)/2; y=(d1.x+d2.y)/2;}
```

```
void Showl(void){ cout<<"Dot1: "; d1.Show(); cout<<"Dot2: "; d2.Show();  
cout<<"Length="<<sqrt((d1.x-d2.x)*(d1.x-d2.x)+(d1.y-d2.y)*(d1.y-d2.y))<<endl;  
cout<<"Center: " <<"x="<<x<<"\t"<<"y="<<y<<endl; }
```

```
};
```

```
void main(void)
```

```
{ float a,b;
```

```
cout<<"Input Dot1: \n"; cin>>a>>b;
```

```
Dot dot1(a,b);//调用Dot的构造函数
```

```
cout<<"Input Dot2: \n"; cin>>a>>b;
```

```
Dot dot2(a,b); Line line(dot1,dot2); line.Showl();}
```

用坐标初始化Dot对象

打印坐标

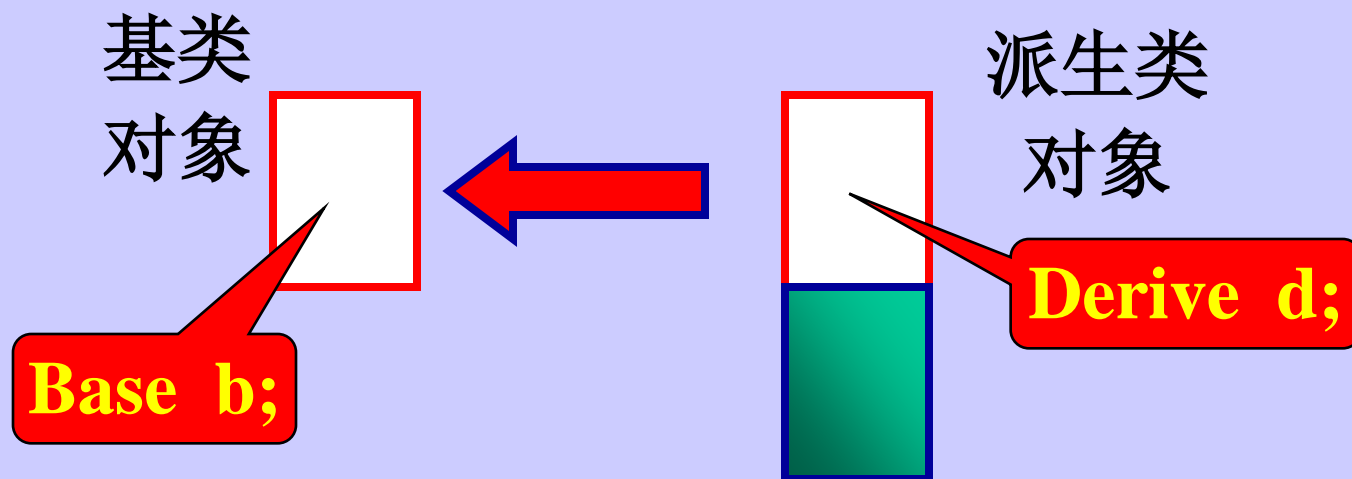
在Line中新说明的成员

对成员初始化

x, y是继承基类的成员

赋值兼容规则

相互之间能否赋值？



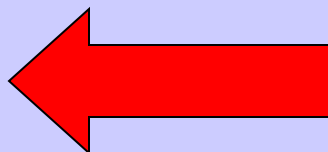
可以将派生类对象的值赋给基类对象。反之不行

$b = d;$

只是将从基类继承来的成员赋值。

x
y
Dot(x,y)
Dot(&dot)
Show()

Dot的对象空间



Line的对象空间

```
Dot dot;
Line line;
dot=line;
line=dot;
```

非法

x	
y	
Dot(x,y)	
Dot(&dot)	
Show()	
d1	x
	y
	Dot(x,y)
	Dot(&dot)
	Show()
d2	x
	y
	Dot(x,y)
	Dot(&dot)
	Show()
Line()	
Showl()	

从基类继承

基类对象

可以将一个派生类对象的地址赋给基类的指针变量。

基类指针

`Base *basep;`

`basep=&b;`

基类对象

基类对象

`basep`

`Base b;`

派生类对象

`basep = &d`

`basep`只能引用
从基类继承来
的成员。

派生类对象

`basep`

`Derive d;`

派生类对象可以初始化基类的引用。

Derive d;

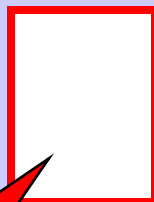
派生类对象

Base basei=&d;

基类引用

basei只能引用
从基类继承来
的成员。

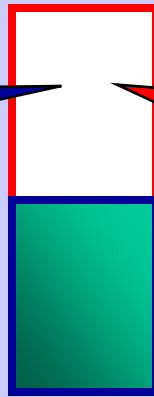
基类对象



Base b;

派生类对象

别名basei



Derive d;


```

class A{
public: int x;
      A(int a=0) { x=a;}
};

class B{
public: int y;
      B(int a=0) { y=a;}
};

class C:public A,public B{ int z;
public: C(int a,int b,int m): A(a), B(b) { z=m; }
      void Show() { cout<<"x="<<x<<"\t"; cout<<"y="<<y<<"\t";
                    cout<<"z="<<z<<"\n"; }
};

void main(void)
{
    A a1(100);    B b1(200);    C c1(10,20,50);
    cout<<"a1.x="<<a1.x<<endl;  cout<<"b1.y="<<b1.y<<endl;
    c1.Show();    a1=c1; b1=c1; //派生类对象向基类对象赋值
    cout<<"a1.x="<<a1.x<<endl;  cout<<"b1.y="<<b1.y<<endl;
    A *p;    p=&c1; //基类指针指向派生类对象    p->Show();
}

```

a1

10

A()

b1

20

B()

c1

x	10
A()	
y	20
B()	
z	50
C()	
Show()	

错误!

虚基类

类B 是类A的派生类

类C 是类A的派生类

类D 是类B和类C的派生类

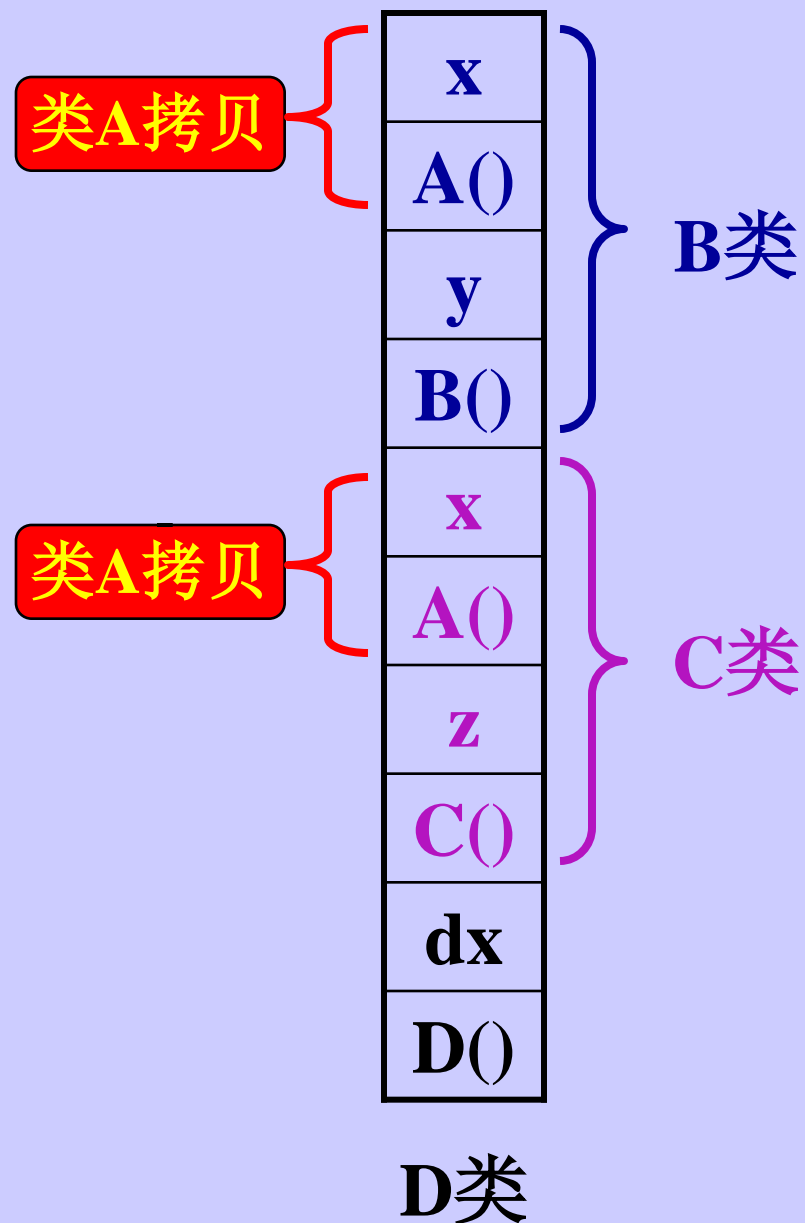
这样，类D中就有两份类A的拷贝



这种同一个公共的基类在派生类中产生多个拷贝，不仅多占用了存储空间，而且可能会造成多个拷贝中的数据不一致和模糊的引用。

D d;

d.x=10; //模糊引用



```

class A{
public: int x;
      A(int a=0) { x=a;}
};

class B:public A{
public: int y;
      B(int a=0,int b=0):A(a) { y=b;}
};

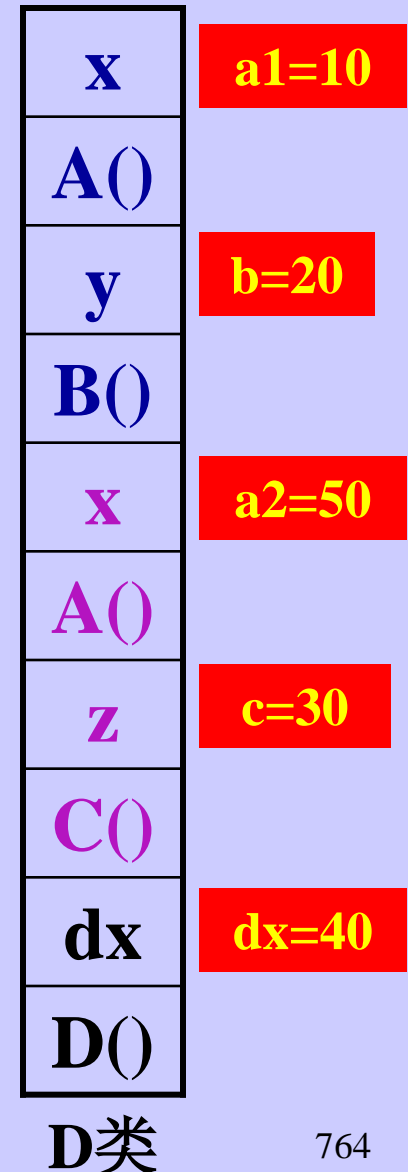
class C:public A{
public: int z;
      C(int a,int c):A(a){ z=c; }
};

class D:public B,public C{
public: int dx;
      D(int a1,int b,int c,int d,int a2):B(a1,b),C(a2,c)
      {      dx=d;}
};

void main(void)
{      D d1(10,20,30,40,50);      cout<<d1.x<<endl; }

```

模糊引用，错误！



在多重派生的过程中，若使公共基类在派生类中只有一个拷贝，则可将这种基类说明为**虚基类**。

在派生类的定义中，只要在基类的类名前加上关键字**virtual**，就可以将基类说明为虚基类。

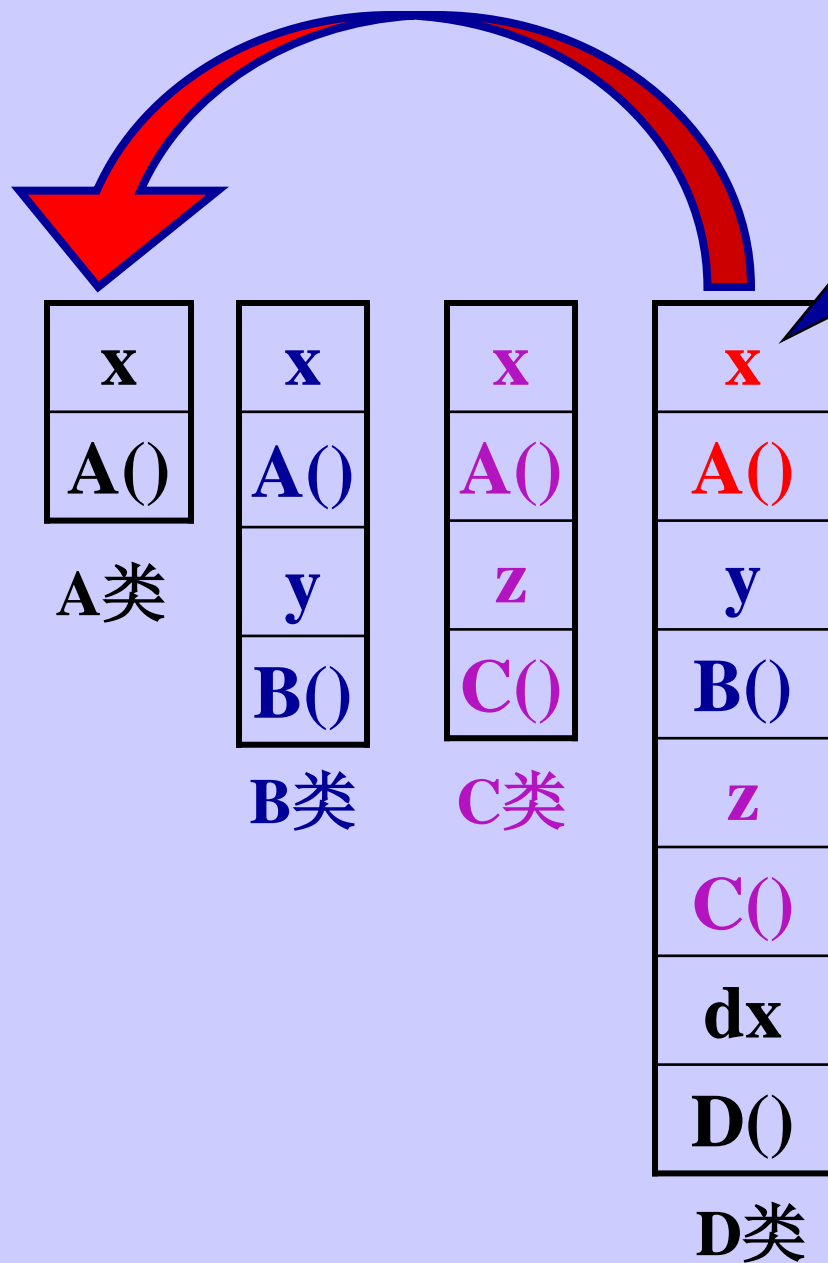
```
class B:public virtual A{
```

```
public:
```

```
    int y;
```

```
    B(int a=0, int b=0 ):A(b) { y=a;}
```

```
};
```



一份拷贝，在D()的构造函数中直接调用A()

由虚基类派生出的对象初始化时，**直接调用**虚基类的构造函数。因此，若将一个类定义为虚基类，**则一定有正确的构造函数**可供所有派生类调用。

```

class A{
public: int x;
      A(int a=0) { x=a;}
};

class B:public virtual A{
public: int y;
      B(int a=0,int b=0): A(a) {y=b;}
};

class C:public virtual A{
public: int z;
      C(int a=0,int c=0):A(a){z=c; }
};

class D:public B,public C{
public: int dx;
      D(int a1,int b,int c,int d,int a2):B(a1,b),C(a2,c) , A(a2)
      {
          dx=d;}
};

```

```

void main(void)
{
    D d1(10,20,30,40,50);
    cout<<d1.x<<endl;
    d1.x=400;
    cout<<d1.x<<endl;
    cout<<d1.y<<endl;
}

```

50	0
400	400
20	20

直接在派生类中调用虚基类的构造函数

没有对虚基类构造函数的调用，用缺省的构造函数

再次强调，用虚基类进行多重派生时，若虚基类没有缺省的构造函数，则在每一个派生类的构造函数中都必须有对虚基类构造函数的调用（且首先调用）。

第十二章 类的其它特性

友元函数

类中私有和保护的成员在类外不能被访问。

友元函数是一种定义在类外部的普通函数，其特点是能够访问类中私有成员和保护成员，即类的访问权限的限制对其不起作用。

友元函数需要在**类体内**进行说明，在前面加上关键字**friend**。

一般格式为：

friend <type> FuncName(<args>);

函数名

friend float Volume(A &a);

关键字

返回值类型

函数参数

友元函数不是成员函数，用法也与普通的函数完全一致，只不过它能访问类中所有的数据。友元函数破坏了类的封装性和隐蔽性，使得非成员函数可以访问类的私有成员。

一个类的友元可以自由地用该类中的所有成员。

```
class A{
```

```
    float x,y;
```

```
public:
```

```
A(float a, float b){ x=a; y=b;}
```

```
float Sum(){ return x+y; }
```

友元函数只能用对象名引用类中的数据。

成员函数

```
friend float Sum(A &a){ return a.x+a.y; }
```

友元函数

私有数据

```
void main(void)
```

```
{ A t1(4,5),t2(10,20);
```

成员函数的调用，利用对象名调用

```
    cout<<t1.Sum()<<endl;
```

```
    cout<<Sum(t2)<<endl;
```

友元函数的调用，直接调用

有关友元函数的使用，说明如下：

友元函数不是类的成员函数

友元函数近似于普通的函数，它不带有**this**指针，因此必须将对象名或对象的引用作为友元函数的参数，这样才能访问到对象的成员。

友元函数与一般函数的不同点在于：

1. 友元函数必须在类的定义中说明，其函数体可在类内定义，也可在类外定义；
2. 它可以访问该类中的所有成员（公有的、私有的和保护了的），而一般函数只能访问类中的公有成员。

```
class A{  
    float x,y;  
public:  
    A(float a, float b){ x=a; y=b;}  
    float Getx(){ return x; }  
    float Gety(){ return y; }  
    float Sum(){ return x+y; }  
    friend float Sum(A &);  
};  
float Sumxy(A &a){ return a.Getx()+a.Gety(); }  
float Sum(A &a){ return a.x+a.y; }  
void main(void)  
{ A t1(1,2),t2(10,20), t3(100,200);  
  cout<<t1.Sum()<<endl;  
  cout<<Sum(t2)<<endl;  
  cout<<Sumxy(t3)<<endl;  
}
```

成员函数

友元函数

普通函数，必须通过公有函数访问私有成员

友元函数,可以直接调用类中私有成员

对象调用成员函数

调用友元函数

调用一般函数

友元函数不受类中访问权限关键字的限制，可以把它放在类的私有部分，放在类的公有部分或放在类的保护部分，其作用都是一样的。换言之，**在类中对友元函数指定访问权限是不起作用的。**

友元函数的作用域与一般函数的作用域相同。

谨慎使用友元函数

通常使用友元函数来**取**对象中的数据成员值，而**不修改**对象中的成员值，则肯定是安全的。

大多数情况是友元函数是某个类的成员函数，即A类中的某个成员函数是B类中的友元函数，这个成员函数可以直接访问B类中的私有数据。这就实现了类与类之间的沟通。

```
class A{
```

```
...
```

```
void fun( B &);
```

```
};
```

既是类A的成员函数

```
class B{
```

```
...
```

```
friend void fun( B &);
```

```
};
```

又是类B的友元函数

注意：一个类的成员函数作为另一个类的友元函数时，应先定义友元函数所在的类。

class B ; //先定义类A，则首先对类B作引用性说明

```
class A{  
    ..... //类A的成员定义  
    public:  
    void fun( B & );//函数的原型说明  
};  
class B{    .....  
    friend void A::fun( B & );//定义友元函数  
};  
void A::fun ( B &b) //函数的完整定义  
{  
    ..... //函数体的定义  
}
```

类A中的成员函数
fun()是类B的友元函数。即在**fun()**中可以直接引用类B的私有成员。

```
class B;           //必须在此进行引用性说明,
```

```
class A{
```

```
    float x,y;
```

```
public:
```

```
    A(float a, float b){ x=a; y=b;}
```

```
    float Sum(B &); //说明友元函数的函数原型，是类A的一成员函数
```

```
};
```

```
class B{
```

```
    float m,n;
```

```
public:
```

```
    B(float a,float b){ m=a;n=b; }
```

```
    friend float A::Sum(B &); //说明类A的成员函数是类B的友元函数
```

```
}
```

```
float A::Sum( B &b)
```

//定义该友元函数

```
{ x=b.m+b.n; y=b.m-b.n; }
```

```
void main(void)
```

```
{ A a1(3,5);
```

```
  B b1(10,20);
```

```
  a1.Sum(b1); //调用该函数，因是类A的成员函数，故用类A的对象调用
```

```
}
```

类A中有一个函数可以直接引用类B的私有成员

直接引用类B的私有成员

a1.x=30

a1.y=-10

友元类

```
class A{
```

```
.....
```

```
friend class B;
```

```
}
```

类B是类A的友元

```
class B{
```

```
.....
```

```
}
```

类B可以自由使用
类A中的成员

对于类B而言，类A是透明的

类B必须通过类A的对象使用类A的成员

```

const float PI =3.1415926;
class A{
    float r ;
    float h;
public: A(float a,float b){r=a; h=b;}
    float Getr(){return r;}
    float Geth(){return h;}
    friend class B;//定义类B为类A的友元
};

```

类B中的任何函数都能使用类A中的所有私有成员。

```

class B
{ int number;
public: B(int n=1)      {number=n;}
    void Show(A &a)
    { cout<<PI*a.r*a.r*a.h*number<<endl; }//求类A的某个对象*n的体积
};

```

直接引用类A的私有成员

```

void main(void)
{
    A a1(25,40),a2(10,40);
    B b1(2);
    b1.Show (a1);    b1.Show (a2);
}

```

不管是按哪一种方式派生，基类的私有成员在派生类中都是不可见的。

如果在一个派生类中要访问基类中的私有成员，可以将这个派生类声明为基类的友元。

```
class Base {  
    friend class Derive;  
    ....  
}
```

```
class Derive {  
    ....  
}
```



直接使用Base中的私有成员

```

#include<iostream.h>
class M
{ friend class N; //N为M的友元，可以直接使用M中的私有成员
private:
    int i , j;
    void show(void){cout<<"i="<<i<<"\t"<<"j="<<j<<"\t";}
public:
    M(int a=0,int b=0){ i=a; j=b;}
};
class N :public M{ //N为M的派生类
public:  N(int a=0,int b=0):M(a,b){  }
void Print(void){ show();  cout<<"i+j="<<i+j<<endl;  }
};
void main(void)
{ N n1(10,20);
  M m1(100,200);
// m1.show();  //私有成员函数，在类外不可调用
  n1.Print();
}

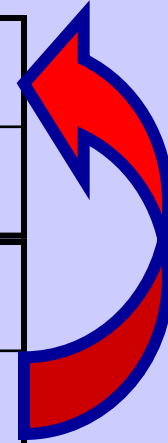
```

直接引用类M的私有成员函数和私有成员

基类对象 M



派生类对象 N



```
showy( ){ show(); cout<<i<<endl; }
```

当派生类是基类的友元时，上式可以调用

```
N n;    n.showy( );    n.show( );
```

错误,类外不可调用

虚函数

多态性是面向对象的程序设计的关键技术。

多态性：调用同一个函数名，可以根据需要
但实现不同的功能。

多态性 { 编译时的多态性（函数重载）
运行时的多态性（虚函数）

运行时的多态性是指在程序执行之前，根据函数名和参数
无法确定应该调用哪一个函数，必须在程序的执行过程中，
根据具体的执行情况来动态地确定

可以将一个派生类对象的地址赋给基类的指针变量。

基类指针

```
Base *basep;
```

基类对象

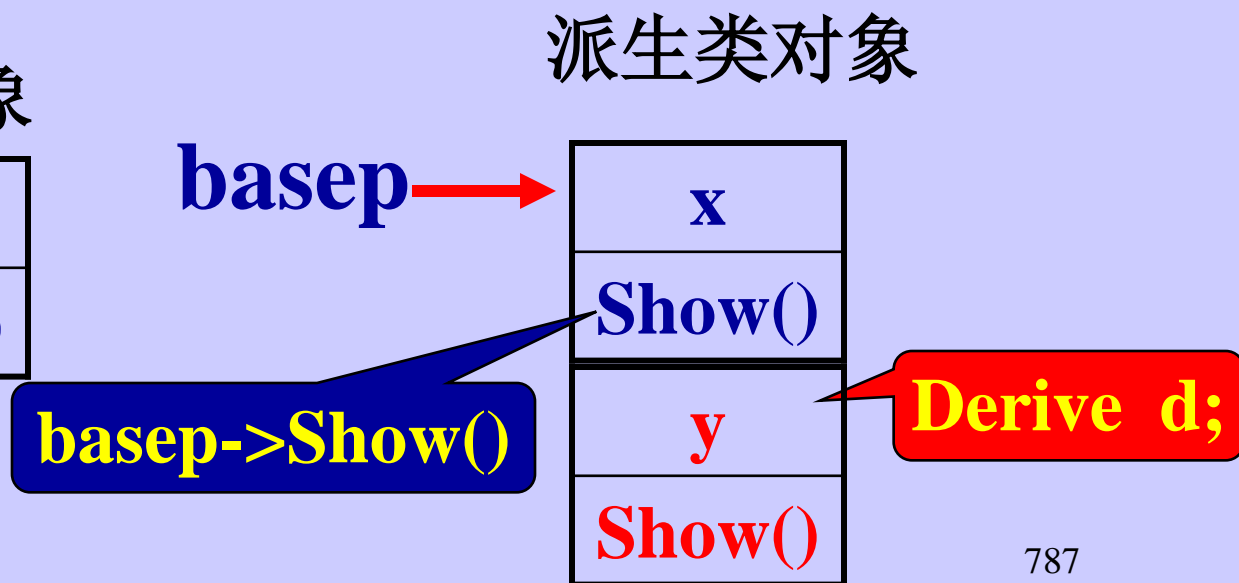
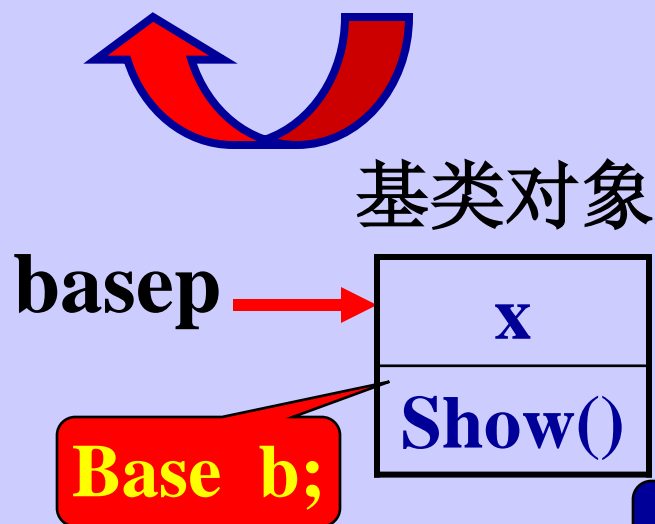
```
basep=&b;
```

派生类对象

```
basep = &d;
```

```
basep ->Show();
```

basep只能引用从基类继承来的成员。



```

class Point{
    float x,y;
public: Point(){
    Point(float i,float j){    x=i;    y=j;    }
    float area(void) {    return 0.0;    }
};

const float Pi=3.14159;

class Circle:public Point{    //类Point的派生类
    float radius;
public: Circle(float r){    radius=r;    }
    float area(void) {    return Pi*radius*radius;    }
};

void main(void)
{    Point *pp;    //基类指针， 可以将派生类对象的地址赋给基类指针
    Circle c(5.4321);
    pp=&c;
    cout<<pp->area ()<<endl;    //调用的是基类中有的公有函数
}

```

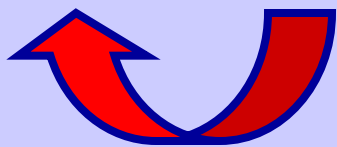
在基类和派生类中具有相同的公有函数area()。

在这种情况下，使用基类的指针时，只能访问从相应基类中继承来的成员，而不允许访问在派生类中增加的成员。输出为 0

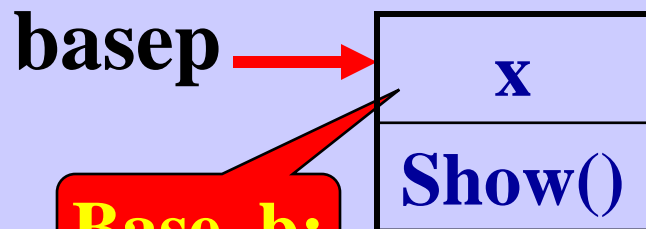
Base *basep;

basep=&b;

basep = &d;

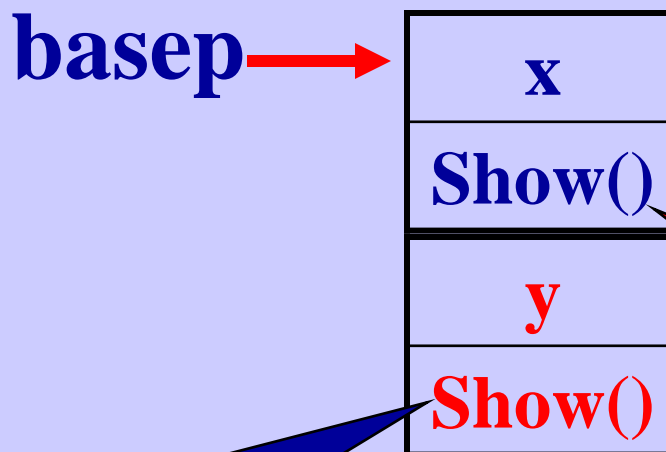


基类对象



Base b;

派生类对象



Derive d;

basep->Show()

basep ->Show();

即指向派生类新增的成员函数

需要将基类中的Show()
说明为虚函数

若要访问派生类中相同名字的函数，必须将基类中的同名函数定义为虚函数，这样，将不同的派生类对象的地址赋给基类的指针变量后，就可以动态地根据这种赋值语句调用不同类中的函数。

```
class Point{ float x,y;
public: Point(){
    Point(float i,float j){    x=i;    y=j;    }
    virtual float area(void) { return 0.0; }
};
```

声明为虚函数

```
const float Pi=3.14159;
```

输出: 92.7011

```
class Circle:public Point{           //类Point的派生类
    float radius;
public: Circle(float r){ radius=r;    }
    float area(void) { return Pi*radius*radius;}
};
```

虚函数再定义

```
void main(void)
{   Point *pp;           //基类指针, 可以将派生类对象的地址赋给基类指针
    Circle c(5.4321);
    pp=&c;
    cout<<pp->area ()<<endl; //调用虚函数
}
```

调用虚函数

将area()声明为虚函数, 编译器对其进行动态聚束, 按照实际对象c调用了Circle中的函数area()。使Point类中的area()与Circle类中的area()有一个统一的接口。

虚函数的定义和使用

可以在程序运行时通过调用相同的函数名而实现不同功能的函数称为虚函数。定义格式为：

virtual <type> FuncName(<ArgList>);

一旦把基类的成员函数定义为虚函数，由基类所派生出来的所有派生类中，该函数均保持虚函数的特性。

在派生类中重新定义基类中的虚函数时，可以不用关键字**virtual**来修饰这个成员函数。

虚函数是用关键字**virtual**修饰的某基类中的**protected**或**public**成员函数。它可以在派生类中重新定义，以形成不同版本。只有在程序的执行过程中，依据指针具体指向哪个类对象，或依据引用哪个类对象，才能确定激活哪一个版本，实现动态聚束。

```

class A{
protected:      int x;
public:  A(){x=1000;}
        virtual void print(){      cout <<"x="<<x<<"\t";    }//虚函数
};

class B:public A{      int y;
public:  B() { y=2000;}
        void print(){      cout <<"y="<<y<<"\t";    }//派生虚函数
};

class C:public A{      int z;
public:  C(){z=3000;}
        void print(){      cout <<"z="<<z<<"\n";    }//派生虚函数
};

void main(void )
{  A  a, *pa;
   B  b; C  c;
   a.print();  b.print();  c.print(); //静态调用
pa=&a;  pa->print();//调用类A的虚函数
pa=&b;  pa->print();//调用类B的虚函数
pa=&c;  pa->print();}//调用类C的虚函数

```

```
class Base {  
public :  
virtual int Set(int a, int b)  
{ ..... }  
....  
};
```

int Set(int ,int)是虚函数

```
class Derive:public Base{  
public :  
int Set(int x, int y)  
{ ..... }  
....  
};
```

```
class Base {  
public :  
virtual int Set(int a, int b)  
{ ..... }  
....  
};
```

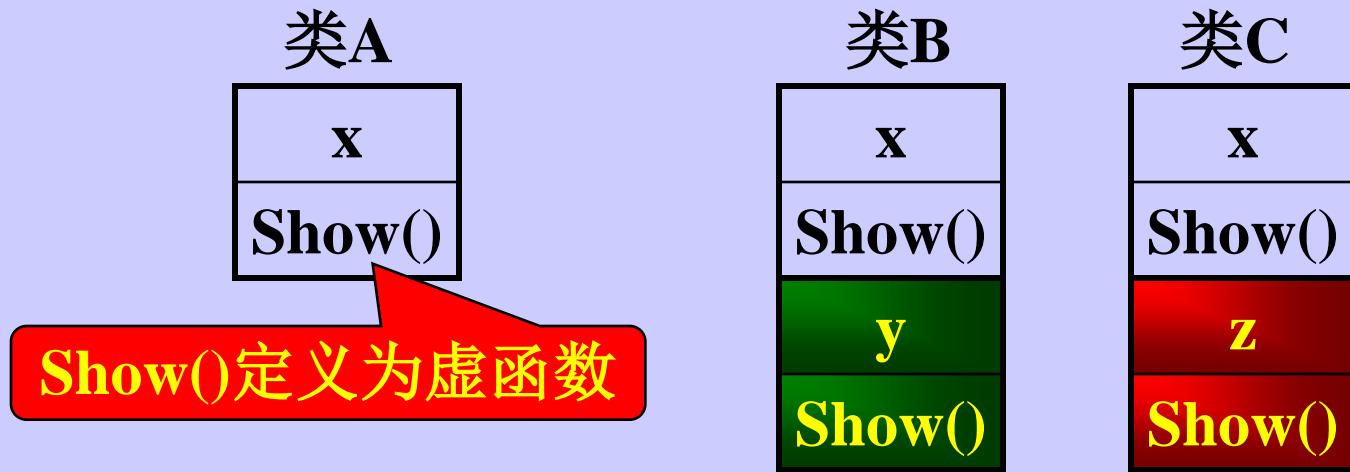
两个Set()函数参数
不一致，是重载，
不是虚函数

```
class Derive:public Base{  
public :  
int Set(int x, int y=0)  
{ ..... }  
....  
};
```

关于虚函数，说明以下几点：

1、当在基类中把成员函数定义为虚函数后，在其派生类中定义的虚函数必须与基类中的虚函数同名，参数的类型、顺序、参数的个数必须一一对应，函数的返回的类型也相同。若函数名相同，但参数的个数不同或者参数的类型不同时，则属于函数的重载，而不是虚函数。若函数名不同，显然这是不同的成员函数。

2、实现这种动态的多态性时，必须使用**基类类型**的**指针变量**，并使该指针**指向不同的派生类对象**，并通过调用指针所指向的虚函数才能实现动态的多态性。



类B与类C均为类A的公有派生。

```
A  *p;  
B   b;  
C   c;  
  
p=&b ;  
p->Show();  
  
p=&c ;  
p->Show();
```

即在程序运行时，
通过赋值语句实现多态性

3、虚函数必须是类的一个成员函数，不能是友元函数，也不能是静态的成员函数。

4、在派生类中**没有重新定义虚函数**时，与一般的成员函数一样，当调用这种派生类对象的虚函数时，**则调用其基类中的虚函数**。

5、可把析构函数定义为虚函数，但是，不能将构造函数定义为虚函数。

6、虚函数与一般的成员函数相比较，调用时的执行速度要慢一些。为了实现多态性，在每一个派生类中均要保存相应虚函数的入口地址表，函数的调用机制也是间接实现的。因此，除了要编写一些通用的程序，并一定要使用虚函数才能完成其功能要求外，通常不必使用虚函数。

7、一个函数如果被定义成虚函数，则不管经历多少次派生，仍将保持其虚特性，以实现“一个接口，多个形态”。

虚函数的访问

用基指针访问与用对象名访问

用基指针访问虚函数时，指向其实际派生类对象重新定义的函数。实现动态聚束。

通过一个对象名访问时，只能静态聚束。即由编译器在编译的时候决定调用哪个函数。


```

class Point{ float x,y;
public: Point(){
    Point(float i,float j){      x=i;    y=j;    }
    virtual float area(void) { return 0.0; }//声明为虚函数
};

const float Pi=3.14159;
class Circle:public Point{      //类Point的派生类
    float radius;
public: Circle(float r){ radius=r;    }
    float area(void) { return Pi*radius*radius;}//虚函数再定义
};

void main(void)
{   Point *pp;      //基类指针，可以将派生类对象的地址赋给基类指针
    Circle c(5.4321);
    cout<<c.area()<<endl;
    cout<<c.Point::area()<<endl;
    cout<<c.Circle::area ()<<endl;
}

```

输出: 92.7011
0
92.7011

用对象名调用area()

可见，利用对象名进行调用与一般非虚函数没有区别。

```
class base0{
public: void v(void){    cout<<"base0\n";    }
};
```

base0

base0

```
class base1:public base0{
public: virtual void v(void){ cout<<"base1\n"; }
};
```

```
class A1:public base1{
public: void v(){    cout<<"A1\n";  }
};
```

```
class A2:public A1{
public: void v(void){    cout<<"A2\n";  }
};
```

```
class B1:private base1{
public: void v(void){    cout<<"B1\n";  }
};
```

```
class B2:public B1{
public: void v(void){    cout<<"B2\n";  }
};
```

```
void main(void)
{  base0 *pb;
  A1 a1;
  (pb=&a1)->v();
  A2 a2;
  (pb=&a2)->v();
  B1 b1;
  (pb=&b1)->v();
  B2 b2;
  (pb=&b2)->v();
}
```

私有派生，在类外
不能调用基类函数

```

class base0{
public: void v(void){    cout<<"base0\n";    }
};

class base1:public base0{
public: virtual void v(void){ cout<<"base1\n"; }
};

class A1:public base1{
public: void v(){    cout<<"A1\n"; }
};

class A2:public A1{
public: void v(void){    cout<<"A2\n"; }
};

class B1:private base1{
public: void v(void){    cout<<"B1\n"; }
};

class B2:public B1{
public: void v(void){    cout<<"B2\n"; }
};

```

A1

A2

```

void main(void)
{
    base1 *pb;
    A1 a1;
    (pb=&a1)->v();
    A2 a2;
    (pb=&a2)->v();
}

```

纯虚函数

在基类中不对虚函数给出有意义的实现，它只是在派生类中有具体的意义。这时基类中的虚函数只是一个入口，具体的目的地由不同的派生类中的对象决定。这个虚函数称为**纯虚函数**。

```
class <基类名>
{
    virtual <类型><函数名>(<参数表>)=0;
    .....
};
```

抽象类

```
class A{
protected:    int x;
public:  A(){x =1000;}
        virtual void print()=0; //定义纯虚函数
};

class B:public A{ //派生类
private:  int y;
public:  B(){ y=2000;}
        void print(){cout <<"y="<<y<<"\n";} //重新定义纯虚函数
};

class C:public A{ //派生类
        int z;
public:  C(){z=3000;}
        void print(){cout <<"z="<<z<<"\n";} //重新定义纯虚函数
};

void main(void )
{  A *pa;      B b;   C c;
   pa=&b;  pa->print(); pa=&c;  pa->print();
   A a;  pa=&a;  pa->print();
}
```

y=2000

z=3000

不能定义抽象类的对象

1、在定义纯虚函数时，不能定义虚函数的实现部分。

2、把函数名赋于0，本质上是将指向函数体的指针值赋为初值0。与定义空函数不一样，空函数的函数体为空，即调用该函数时，不执行任何动作。在没有重新定义这种纯虚函数之前，是不能调用这种函数的。

3、把至少包含一个纯虚函数的类，称为抽象类。这种类只能作为派生类的基类，不能用来说明这种类的对象。

其理由是明显的：因为虚函数没有实现部分，所以不能产生对象。但可以定义指向抽象类的指针，即指向这种基类的指针。当用这种基类指针指向其派生类的对象时，**必须在派生类中重载纯虚函数**，否则会产生程序的运行错误。

4、在以抽象类作为基类的派生类中必须有纯虚函数的实现部分，即必须有重载纯虚函数的函数体。否则，这样的派生类也是不能产生对象的。

综上所述，可把纯虚函数归结为：抽象类的唯一用途是为派生类提供基类，纯虚函数的作用是作为派生类中的成员函数的基础，并实现动态多态性。

虚基类

多基派生中的多条路径具有公共基类时，在这条路径的汇合处就会因对公共基类产生多个拷贝而产生同名函数调用的二义性。

解决这个问题的办法就是把**公共基类定义为虚基类**，使由它派生的多条路径的汇聚处只产生一个拷贝。

```
class Base{ };
```

```
class A : public Base{ };
```

```
class B: public Base{ };
```

```
class C: public A, public B{ };
```

类C中继承了两个类Base，
即有两个类Base的实现部分，
在调用时产生了二义性。

由虚基类派生出的对象初始化时，**直接调用**虚基类的构造函数。因此，若将一个类定义为虚基类，则一定有正确的构造函数可供所有派生类调用。

用虚基类进行多重派生时，**若虚基类没有缺省的构造函数**，则在每一个派生类的构造函数中**都必须有对虚基类构造函数的调用**（且首先调用）。

```

class base{
public:
virtual void a(){ cout<<"a() in base\n";}
virtual void b(){ cout<<"b() in base\n";}
virtual void c(){ cout<<"c() in base\n";}
virtual void d(){ cout<<"d() in base\n";}
virtual void e(){ cout<<"e() in base\n";}
virtual void f(){ cout<<"f() in base\n";}
};

class A:public base{
public:
virtual void a(){ cout<<"a() in A\n";}
virtual void b(){ cout<<"b() in A\n";}
virtual void f(){ cout<<"f() in A\n";}
};

class B:public base{
public:
virtual void a(){ cout<<"a() in B\n";}
virtual void b(){ cout<<"b() in B\n";}
virtual void c(){ cout<<"c() in B\n";}
};

```

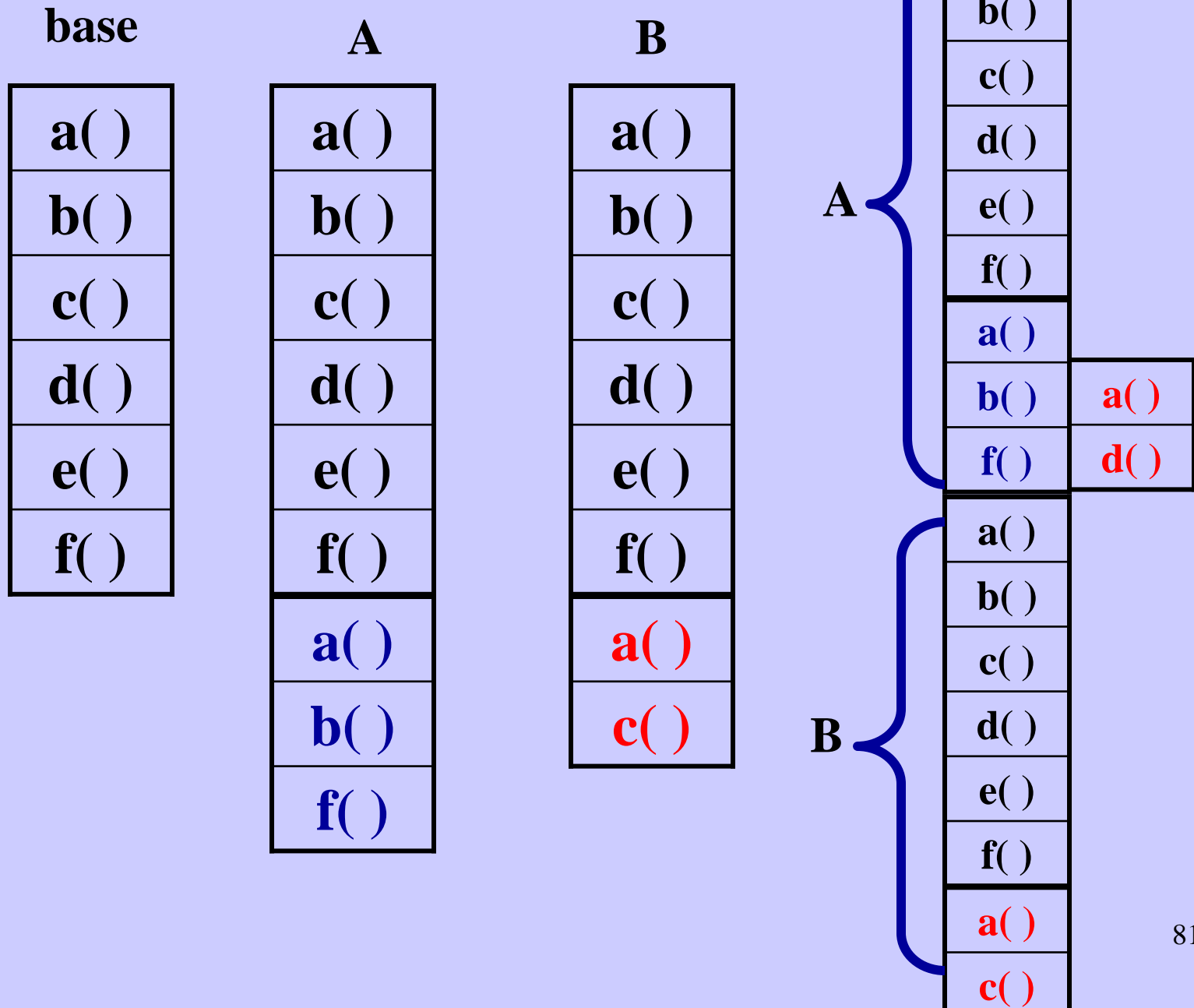
```

class C:public A,public B{
public:
virtual void a(){ cout<<"a() in C\n";}
virtual void d(){ cout<<"d() in C\n";}
};

void main(void)
{
    C cc;
    base *pbase=&cc;//错误
    A *pa=&cc;
    pa->a();
    pa->b();
    pa->c();
    pa->d();
    pa->e();
    pa->f();
}

```

将类C的地址赋值时产生歧义



```

class base{
public:
virtual void a(){ cout<<"a() in base\n";}
virtual void b(){ cout<<"b() in base\n";}
virtual void c(){ cout<<"c() in base\n";}
virtual void d(){ cout<<"d() in base\n";}
virtual void e(){ cout<<"e() in base\n";}
virtual void f(){ cout<<"f() in base\n";}
};

class A:public base{
public:
virtual void a(){ cout<<"a() in A\n";}
virtual void b(){ cout<<"b() in A\n";}
virtual void f(){ cout<<"f() in A\n";}
};

class B:public base{
public:
virtual void a(){ cout<<"a() in B\n";}
virtual void b(){ cout<<"b() in B\n";}
virtual void c(){ cout<<"c() in B\n";}
};

```

```

class C:public A,public B{
public:
virtual void a(){ cout<<"a() in C\n";}
virtual void d(){ cout<<"d() in C\n";}
};

void main(void)
{
    C cc;
    base *pbase=&cc;//错误
    A *pa=&cc;
    pa->a();           a() in C
    pa->b();           b() in A
    pa->c();           c() in base
    pa->d();           d() in C
    pa->e();           e() in base
    pa->f();           f() in A
}

```

将类C的地址赋值时产生歧义

类C中有两个base，只有一个A
为避免这种情况，将base定义为虚基类。

```

class base{
public:
virtual void a(){ cout<<"a() in base\n";}
virtual void b(){ cout<<"b() in base\n";}
virtual void c(){ cout<<"c() in base\n";}
virtual void d(){ cout<<"d() in base\n";}
virtual void e(){ cout<<"e() in base\n";}
virtual void f(){ cout<<"f() in base\n";}
};

class A:virtual public base{
public:
virtual void a(){ cout<<"a() in A\n";}
virtual void b(){ cout<<"b() in A\n";}
virtual void f(){ cout<<"f() in A\n";}
};

class B:virtual public base{
public:
virtual void a(){ cout<<"a() in B\n";}
virtual void c(){ cout<<"c() in B\n";}
};

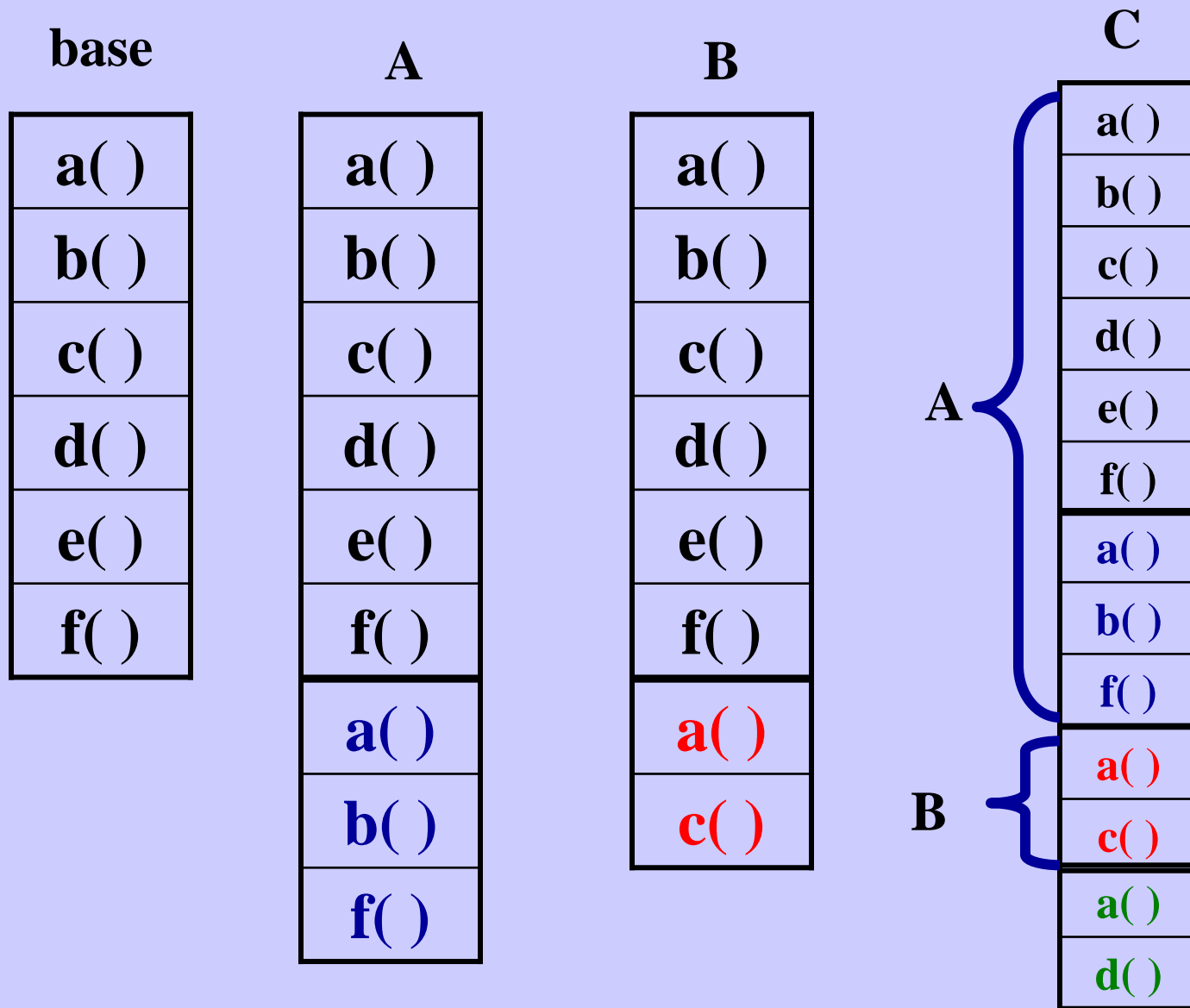
```

```

class C:public A,public B{
public:
virtual void a(){ cout<<"a() in C\n";}
virtual void d(){ cout<<"d() in C\n";}
};

void main(void)
{
    C cc;
    base *pa=&cc;
    pa->a();
    pa->b();
    pa->c();
    pa->d();
    pa->e();
    pa->f();
}

```



```

class base{
public:
virtual void a(){ cout<<"a() in base\n";}
virtual void b(){ cout<<"b() in base\n";}
virtual void c(){ cout<<"c() in base\n";}
virtual void d(){ cout<<"d() in base\n";}
virtual void e(){ cout<<"e() in base\n";}
virtual void f(){ cout<<"f() in base\n";}
};
class A:virtual public base{
public:
virtual void a(){ cout<<"a() in A\n";}
virtual void b(){ cout<<"b() in A\n";}
virtual void f(){ cout<<"f() in A\n";}
};
class B:virtual public base{
public:
virtual void a(){ cout<<"a() in B\n";}
virtual void c(){ cout<<"c() in B\n";}
};

```

```

class C:public A,public B{
public:
virtual void a(){ cout<<"a() in C\n";}
virtual void d(){ cout<<"d() in C\n";}
};
void main(void)
{
    C cc;
    base *pa=&cc;
    pa->a();          a() in C
    pa->b();          b() in A
    pa->c();          c() in B
    pa->d();          d() in C
    pa->e();          e() in base
    pa->f();          f() in A
}

```

类C中只有一个base


```

class base{
public:
void a(){cout<<"a() in base\n";}
void b(){cout<<"b() in base\n";}
void c(){cout<<"c() in base\n";}
void d(){cout<<"d() in base\n";}
void e(){cout<<"e() in base\n";}
void f(){ cout<<"f() in base\n";}
};

class A:virtual public base{
public:
void a(){cout<<"a() in A\n";}
void b(){cout<<"b() in A\n";}
void f(){ cout<<"f() in A\n";}
};

class B:virtual public base{
public:
void a(){cout<<"a() in B\n";}
void c(){cout<<"c() in B\n";}
};

```

```

class C:public A,public B{
public:
void a(){                cout<<"a() in C\n";}
void d(){                cout<<"d() in C\n";}
};

void main(void)
{
    C cc;
    base *pa=&cc;
    pa->a();                a() in base
    pa->b();                b() in base
    pa->c();                c() in base
    pa->d();                d() in base
    pa->e();                e() in base
    pa->f();                f() in base
}

```

类C中只有一个base

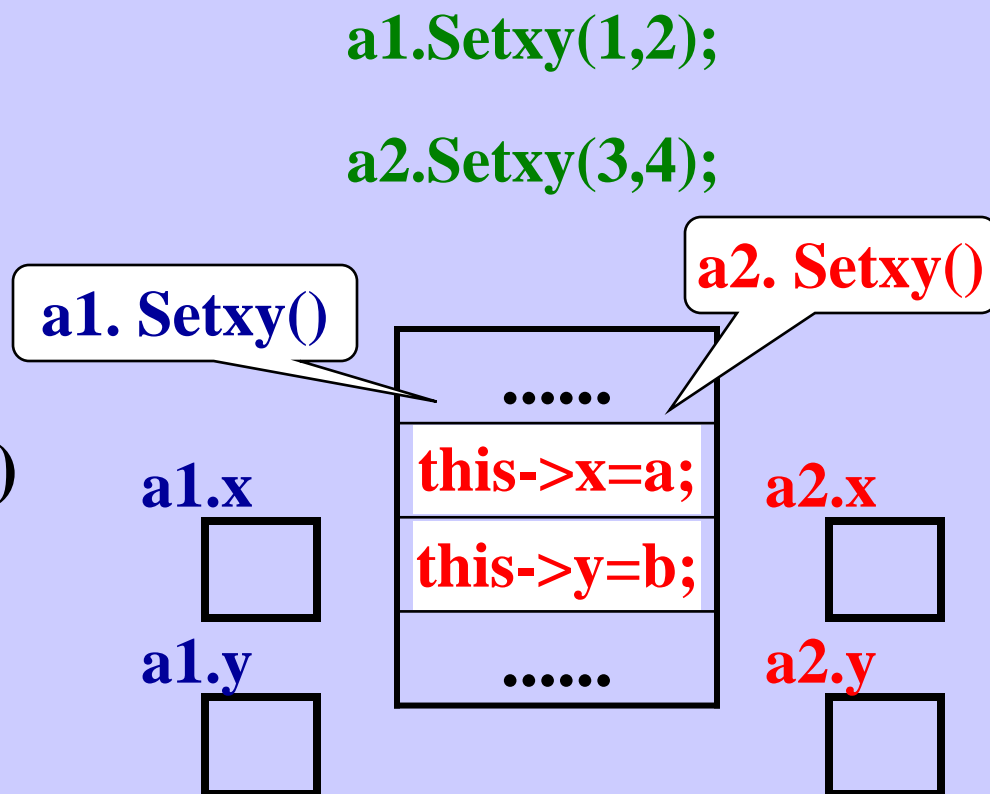
下面程序的输出是_____。

```
class A{
protected:int x;
public:A(){x =1000;}
virtual void p(){cout <<"x="<<x<<"\n"; p2();}
virtual void p2(){cout<<"A::p2()"<<endl;}
};
class C:public A{
    int z;
public:C(){z=3000;}
    void p(){cout <<"z="<<z<<"\n"; p2();}
virtual void p2(){cout<<"C::p2()"<<endl;}
};
void main(void ){
C c;
A a,*pa=&a;
pa->p();
pa=&c;pa->p();}
```

静态成员

通常，每当说明一个对象时，把该类中的有关成员数据拷贝到该对象中，即同一类的不同对象，其成员数据之间是互相独立的。

```
class A{  
    int x,y;  
public:  
    void Setxy(int a, int b)  
    { x=a; y=b;}  
};  
A a1, a2;
```

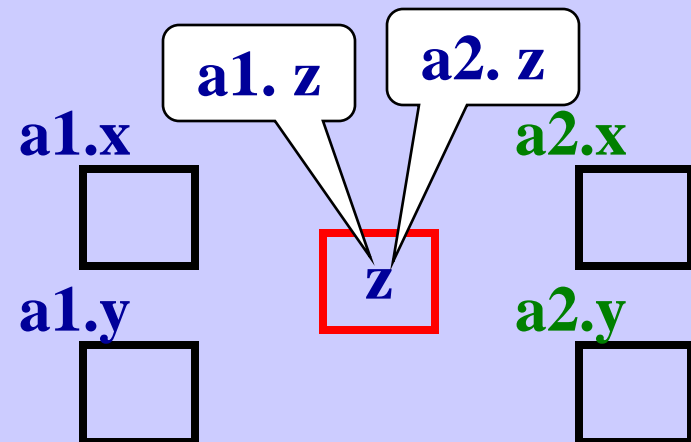


当我们将类的某一个数据成员的存储类型指定为静态类型时，则由该类所产生的所有对象，其静态成员均共享一个存储空间，这个空间是在编译的时候分配的。换言之，在说明对象时，并不为静态类型的成员分配空间。

在类定义中，用关键字static修饰的数据成员称为静态数据成员。

不同对象，同一空间

```
class A{  
    int x,y; static int z;  
    public:  
    void Setxy(int a, int b)  
    { x=a; y=b;}  
};  
A a1, a2;
```



有关静态数据成员的使用，说明以下几点：

1、类的静态数据成员是静态分配存储空间的，而其它成员是动态分配存储空间的（全局变量除外）。当类中没有定义静态数据成员时，在程序执行期间遇到说明类的对象时，才为对象的所有成员依次分配存储空间，这种存储空间的分配是动态的；而当类中定义了静态数据成员时，在编译时，就要为类的静态数据成员分配存储空间。

2、必须在文件作用域中，对静态数据成员作一次且只能作一次定义性说明。因为静态数据成员在定义性说明时已分配了存储空间，所以通过静态数据成员名前加上类名和作用域运算符，可直接引用静态数据成员。在C++中，静态变量缺省的初值为0，所以静态数据成员总有唯一的初值。当然，在对静态数据成员作定义性的说明时，也可以指定一个初值。

```

class A{
    int i,j;
    static int x,y;//定义静态成员
public: A(int a=0,int b=0,int c=0, int d=0){        i=a;j=b;x=c;y=d;        }
    void Show(){cout << "i="<<i<<"\t"<<"j="<<j<<"\t";
                cout << "x="<<x<<"\t"<<"y="<<y<<"\n";
                }
};

```

int A::x=0; //必须对静态成员作一次定义性说明

int A::y=0;

```

void main(void )

```

```

{    A  a(2,3,4,5);

```

```

    a.Show();

```

```

    A  b(100,200,300,400);

```

```

    b.Show();

```

```

    a.Show();

```

```

}

```

a.x 和b.x在内存中占据一个空间

a.y 和b.y在内存中占据一个空间

i=2 j=3 x=4 y=5

i=100 j=200 x=300 y=400

i=2 j=3 x=300 y=400

```

class A{      int i,j;
public:  static int x;
public:   A(int a=0,int b=0,int c=0){ i=a ; j=b ; x=c;      }
        void Show(){
            cout << "i="<<i<<"\t"<<"j="<<j<<"\t";
            cout << "x="<<x<<"\n";
        }
};

```

```

int A::x=500; //int A::x

```

在类外重新定义

```

void main(void )
{
    A a(20,40,10),b(30,50,100);
    a.Show ();
    b.Show ();
    cout <<"A::x="<<A::x<<"\n"; //可以直接用类名引用
}

```


3、静态数据成员具有全局变量和局部变量的一些特性。静态数据成员与全局变量一样都是静态分配存储空间的，但全局变量在程序中的任何位置都可以访问它，而静态数据成员受到访问权限的约束。必须是public权限时，才可能在类外进行访问。

4、为了保持静态数据成员取值的一致性，通常在构造函数中不给静态数据成员置初值，而是在对静态数据成员的定义性说明时指定初值。

```

class A{      int i;
    static int count;
public:
    A(int a=0)
    {   i=a; count++;
        cout <<"Number of Objects="<<count<<"\n";   }
    ~A()
    { count--; cout <<"Number of Objects="<<count<<"\n"; }
    void Show()
    {   cout << "i="<<i<<"\n";   cout << "count="<<count<<"\n";}
};
int A::count;
void main(void )
{
    A a1(100);
    A b[2];
    a1.Show();
}

```

Number of Objects=1

Number of Objects=2

Number of Objects=3

i=100

count=3

Number of Objects=2

Number of Objects=1

Number of Objects=0

静态成员函数

可以将类的成员函数定义为静态的成员函数。即使用关键字**static**来修饰成员函数。

```
class A  
  
{ float x, y;  
  
public :  
  
    A() { }  
  
    static void sum(void) { ..... }  
  
};
```

对静态成员函数的用法说明以下几点：

- 1、与静态数据成员一样，在类外的程序代码中，通过类名加上作用域操作符，可直接调用静态成员函数。
- 2、静态成员函数只能直接使用本类的静态数据成员或静态成员函数，但不能直接使用非静态的数据成员（可以引用使用）。这是因为静态成员函数可被其它程序代码直接调用，所以，它不包含对象地址的this指针。

```

class Tc {
private:int A;

    static int B;//静态数据成员

public:Tc(int a){A=a; B+=a;}

    static void display(Tc c)//Tc的对象为形参
    {
        cout<<"A="<<c.A<<",B="<<B<<endl;
    }
};

int Tc::B=2;

void main(void)
{
    Tc a(2),b(4);
    Tc::display (a);
    Tc::display (b);
}

```

非静态成员，用
对象名来引用

静态成员，
直接引用

直接用类名来调用
静态成员函数

A=2,B=8

A=4,B=8

- 3、静态成员函数的实现部分在类定义之外定义时,其前面不能加修饰词**static**。这是由于关键字**static**不是数据类型的组成部分,因此,在类外定义静态成员函数的实现部分时,不能使用这个关键字
- 4、**不能把静态成员函数定义为虚函数**。静态成员函数也是在编译时分配存储空间,所以在程序的执行过程中不能提供多态性。
- 5、可将静态成员函数定义为内联的 (**inline**), 其定义方法与非静态成员函数完全相同。

```
class Tc {  
private:int A;  
        static int B;//静态数据成员  
public:Tc(int a){A=a; B+=a;}  
        static void display(Tc c); //Tc的对象为形参  
};  
void Tc::display(Tc c)//不用static修饰  
{    cout<<"A="<<c.A<<"B="<<c.B<<endl; }  
int Tc::B=2;  
void main(void)  
{    Tc a(2),b(4);  
        Tc::display (a);  
        Tc::display (b);  
}
```

函数原型

类外定义

9. 下述程序的输出结果是_____。

```
#include<iostream.h>

class Sample{
public:
    Sample(){ cout<<"Constructor"<<endl;  }
    ~Sample(){ cout<<"Destructor"<<endl;  }
};

void fun(int i)
{
    static Sample c;
    cout<<"i="<<i<<endl;
}

void main(void)
{
    fun(10);
    fun(20);
}
```


const、volatile对象和成员函数

用const修饰的对象，只能访问该类中用const修饰的成员函数，而其它的成员函数是不能访问的。用volatile修饰的对象，只能访问该类中用volatile修饰的成员函数，不能访问其它的成员函数。

当希望成员函数只能引用成员数据的值，而不允许成员函数修改数据成员的值时，可用关键词const修饰成员函数。一旦在用const修饰的成员函数中出现修改成员数据的值时，将导致编译错误。

const和volatile成员函数

在成员函数的前面加上关键字const，则表示这函数返回一个常量，其值不可改变。

const成员函数则是指将const放在参数表之后，函数体之前，其一般格式为：

```
<type> FuncName(<args>) const ;
```

其语义是指明这函数的this指针所指向的对象是一个常量，即规定了const成员函数不能修改对象的数据成员，在函数体内只能调用const成员函数，不能调用其它的成员函数。

用volatile修饰一个成员函数时，其一般格式为：

```
<type> FuncName(<args>) volatile;
```

其语义是指明成员函数具有一个易变的this指针，调用这个函数时，编译程序把属于此类的所有的数据成员都看作是易变的变量，编译器不要对这函数作优化工作。

由于关键字const和volatile是属于数据类型的组成部分，因此，若在类定义之外定义const成员函数或volatile成员函数时，则必须用这二个关键字修饰，否则编译器认为是重载函数，而不是定义const成员函数或volatile成员函数。

指向类成员的指针

在C++中可以定义一种特殊的指针，它指向类中的成员函数或类中的数据成员。并可通过这样的指针来使用类中的数据成员或调用类中的成员函数。

指向类中数据成员的指针变量

定义一个指向类中数据成员的指针变量的一般格式为：

```
<type> ClassName:: *PointName;
```

其中type是指针PointName所指向数据的类型，它必须是类ClassName中某一数据成员的类型

- 1、指向类中数据成员的指针变量不是类中的成员，这种指针变量应在类外定义。
- 2、与指向类中数据成员的指针变量同类型的任一数据成员，可将其地址赋给这种指针变量，赋值的一般格式为：

PointName = &ClassName::member;

这种赋值，是取该成员相对于该类的所在对象中的偏移量，即相对地址（距离开始位置的字节数）

如：mptr = &S::y;

表示将数据成员y的相对起始地址赋给指针变量mptr。

- 3、用这种指针访问数据成员时，必须指明是使用那一个对象的数据成员。当与对象结合使用时，其用法为：

ObjectName.* PointName

- 4、由于这种指针变量并不是类的成员，所以使用它只能访问对象的公有数据成员。若要访问对象的私有数据成员，必须通过成员函数来实现。

指向类中成员函数的指针变量

定义一个指向类中成员函数的指针变量的一般格式为：

```
<type> (ClassName:: *PointName) (<ArgsList>) ;
```

其中PointName是指向类中成员函数的指针变量；ClassName是已定义类名；type是通过函数指针PointName调用类中的成员函数时所返回值的数据类型，它必须与类ClassName中某一成员函数的返回值的类型相一致；<ArgsList>是函数的形式参数表。

在使用这种指向成员函数的指针前，应先对其赋值

```
PointName= ClassName::FuncName;
```

同样地，只是将指定成员函数的相对地址赋给指向成员函数的指针。

在调用时，用(对象名.指针)()的形式。

比较： `int max(int a,int b)`

```
{      return (a>b?a:b);      }
```

若有： `int (*f)(int, int);` `f=max;`

则调用时 `(*f)(x,y);`

所以： `(s1.*mptr1)();` `(s1.*mptr2)(100);`

或： `(ps->*mptr1)();` `(ps-*mptr2)(100);`

对指向成员函数的指针变量的使用方法说明以下几点：

- 1、指向类中成员函数的指针变量不是类中的成员，这种指针变量应在类外定义。
- 2、不能将任一成员函数的地址赋给指向成员函数的指针变量，只有成员函数的参数个数、参数类型、参数的顺序和函数的类型均与这种指针变量相同时，才能将成员函数的指针赋给这种变量。
- 3、使用这种指针变量来调用成员函数时，必须指明调用那一个对象的成员函数，这种指针变量是不能单独使用的。用对象名引用。
- 4、由于这种指针变量不是类的成员，所以用它只能调用公有的成员函数。若要访问类中的私有成员函数，必须通过类中的其它的公有成员函数。

5、当一个成员函数的指针指向一个虚函数，且通过指向对象的基类指针或对象的引用来访问该成员函数指针时，同样地产生运行时的多态性。

6、当用这种指针指向静态的成员函数时，可直接使用类名而不要列举对象名。这是由静态成员函数的特性所确定的。

第十三章 运算符重载

函数的重载

所谓函数的重载是指完成不同功能的函数可以具有**相同的函数名**。

C++的编译器是根据**函数的实参**来确定应该调用哪一个函数的。

```
int fun(int a, int b)
```

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

```
int fun (int a)
```

```
{ return a*a; }
```

```
void main(void)
```

```
{ cout<<fun(3,5)<<endl;
```

```
cout<<fun(5)<<endl;
```

```
}
```

8

25

1、定义的重载函数必须具有不同的参数个数，或不同的参数类型。只有这样编译系统才有可能根据不同的参数去调用不同的重载函数。

2、仅返回值不同时，不能定义为重载函数。

```
int sum,a=3,b=2;
```

```
sum=a+b;      (int)=(int) + (int)
```

```
float add, x=3.2, y=2.5;
```

```
add=x+y;      (float)=(float) + (float)
```

系统自动
识别数据
类型

```
char str[4], c1[2]="a", c2[2]="b";
```

```
str=c1+c2;    (char *)=(char *) + (char *)
```

编译系统中的运算符“+”本身不能做这种运算，
若使上式可以运算，必须重新定义“+”运算符，
这种重新定义的过程成为运算符的重载。

```
class A
{ float x,y;
public:
    A(float a=0, float b=0){ x=a; y=b; }
}

void main(void)
{ A a(2,3), b(3,4), c;
  c=a+b;
}
```

两对象不能使用+，必须重新定义+

运算符重载就是赋予已有的运算符多重含义。C++通过重新定义运算符，使它能够用于特定类的对象执行特定的功能。

运算符的重载从另一个方面体现了OOP技术的多态性，且同一运算符根据不同的运算对象可以完成不同的操作。

为了重载运算符，必须定义一个函数，并告诉编译器，遇到这个重载运算符就调用该函数，由这个函数来完成该运算符应该完成的操作。这种函数称为运算符重载函数，它通常是类的成员函数或者是友元函数。运算符的操作数通常也应该是类的对象。

重载为类的成员函数

格式如下：

关键字

运算的对象

<类名> **operator**<运算符>(<参数表>)

{ 函数体 }

返回类型

函数名

运算的对象

A **operator +** (A &); //重载了类A的“+”运算符

其中：**operator**是定义运算符重载函数的关键字，
与其后的运算符一起构成函数名。

没有重载运算符的例子

```
class A
{
    int i;
public:
    A(int a=0) { i=a; }
    void Show(void){ cout<<"i="<<i<<endl; }
    void AddA(A &a, A &b) //利用函数进行类之间的运算
    {
        i=a.i+b.i;
    }
};

void main(void)
{
    A a1(10),a2(20),a3;
    a1.Show ();
    a2.Show ();
    // a3=a1+a2; //不可直接运算
    a3.AddA(a1,a2); //调用专门的功能函数
    a3.Show ();
}
```

利用函数完成了加法运算

用和作对象调用函数

```

class A
{
    int i;
public:
    A(int a=0){ i=a; }
    void Show(void){ cout<<"i="<<i<<endl; }
    void AddA(A &a, A &b) //利用函数进行类之间的运算
    {
        i=a.i+b.i;
    }
    A operator +(A &a) //重载运算符+
    {
        A t; t.i=i+a.i; return t;
    }
};

void main(void)
{
    A a1(10),a2(20),a3;
    a1.Show ();
    a2.Show ();
    a3=a1+a2; //重新解释了加法，可以直接进行类的运算
    a3.AddA(a1,a2); //调用专门的功能函数
    a3.Show ();
}

```

相当于a3=a1.operator+(a2)

重载运算符与一般函数的比较:

相同: 1) 均为类的成员函数; 2) 实现同一功能

返回值 **函数名** **形参列表**

```
void AddA(A &a, A &b)
{   i=a.i+b.i; }
```

函数调用:

```
a3.AddA(a1,a2);
```

由对象a3调用

返回值 **函数名**

```
A operator +(A &a)
{   A t;
    t.i=i+a.i;
    return t;
}
```

形参

函数调用:

```
a3=a1+a2;
a3=a1.operator+(a2);
```

由对象a1调用

返回值

函数名

A operator +(A &a)

{ A t;

t.i=i+a.i;

return t;

}

形参

函数调用:

a3=a1+a2;

a3=a1.operator+(a2);

由对象a1调用

总结:

重新定义运算符，由左操作符调用右操作符。
最后将函数返回值赋给运算结果的对象。

```

class A
{
    int i;
public:
    A(int a=0){ i=a; }
    void Show(void){ cout<<"i="<<i<<endl; }
    void AddA(A &a, A &b) //利用函数进行类之间的运算
    {
        i=a.i+b.i;
    }
    A operator +(A &a) //重载运算符+
    {
        A t; t.i=i+a.i; return t;
    }
};

void main(void)
{
    A a1(10),a2(20),a3;
    a1.Show ();
    a2.Show ();
    a3=a1+a2; //重新解释了加法，可以直接进行类的运算
    a3.AddA(a1,a2); //调用专门的功能函数
    a3.Show ();
}

```

相当于a3=a1.operator+(a2)

当用成员函数实现运算符的重载时，运算符重载函数的参数只能有二种情况：没有参数或带有一个参数。对于只有一个操作数的运算符(如++)，在重载这种运算符时，通常不能有参数；而对于有二个操作数的运算符，只能带有一个参数。这参数可以是对象，对象的引用，或其它类型的参数。在C++中不允许重载有三个操作数的运算符

- 2、在C++中,允许重载的运算符列于表13.1中。
- 3、在C++中不允许重载的运算符列于表13.2。
- 4、只能对C++中已定义了的运算符进行重载，而且，当重载一个运算符时，该运算符的优先级和结合律是不能改变的。

```

class room{
    float Length;
    float Wide;
public:
    room(float a=0.0,float b=0.0){ Length=a; Wide=b; }
    void Show(void){cout<<"Length="<<Length<<"\t"<<"Wide="<<Wide<<endl;}
    void ShowArea(void){ cout<<"Area="<<Length*Wide<<endl; }
    room operator+(room &);//重载运算符+, 函数原型
};

room room::operator + (room &r) //重载运算符, 函数定义
{
    room rr;
    rr.Length =Length+r.Length;
    rr.Wide =Wide+r.Wide ;
    return rr;
}

void main(void)
{
    room r1(3,2),r2(1,4), r3,r4;
    r1.Show ();   r2.Show ();
    r3=r1+r2;      r3.Show ();
    r4=r1+r2+r3;   r4.Show ();}

```

$r4=r1+r2+r3;$

$(r1+r2); (r1+r2)+r3;$

$r4=r1+(r2+r3);$

$(r2+r3); r1+(r2+r3);$

运算符的优先级和结合律是不能改变的


```

class A
{
    int i;
public:
    A(int a=0){ i=a; }
    void Show(void){ cout<<"i="<<i<<endl; }
    A operator +(A &a) //重载运算符+
    {
        A t; t.i=i+a.i; return t; }
    void operator+=(A &a)
    {
        i=i+a.i; }
};

void main(void)
{
    A a1(10),a2(20),a3;
    a1.Show ();
    a2.Show ();
    a3=a1+a2;
    a1+=a2;
    a3.Show ();
}

```

由左操作符调用右操作符，没有返回值，故函数类型为**void**。

相当于a3=a1.operator+(a2)

相当于a1.operator+=(a2)

单目运算符的重载

只具有一个操作数的运算符为单目运算符，最常用的为++及--。

A a;

A a, b;

++a;

b=++a;

a++;

b=a++;

可以看出，虽然运算后对象a的值一致，但先自加或后自加的重载运算符函数的返回值不一致，必须在重载时予以区分。

++为前置运算时，它的运算符重载函数的一般格式为：

```
<type> operator ++( )
```

```
{  
    .....;  
}
```

++为后置运算时，它的运算符重载函数的一般格式为：

```
<type> operator ++(int)
```

```
{  
    .....;  
}
```

A a, b;

b=++a; A operator ++() { ... }

b=a++; A operator ++(int**) { ... }**

```
class A
```

```
{ float x, y;
```

```
public:
```

```
    A(float a=0, float b=0){ x=a; y=b; }
```

```
    A operator ++( ){A t; t.x=++ x; t.y=++y; return t;}
```

```
    A operator ++(int) { A t; t.x=x++; t.y=y++; return t;}
```

```
};
```

```
void main(void)
```

```
{ A a(2,3), b;
```

```
    b=++a;
```

```
    b=a++;
```

```
}
```

返回值

函数名

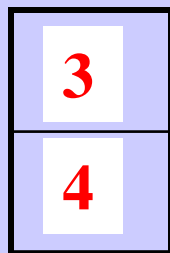
A operator ++()

```
{ A t;  
  t.x=++ x;  
  t.y=++y;  
  return t;  
}
```

A operator ++()

```
{ ++ x;  
  ++y;  
  return *this;  
}
```

a



b=++a;

b=a.operator++();

t作为函数值返回赋给b

将对象本身作为函数值返回赋给b

返回值

函数名

A operator ++(int)

```
{ A t;
```

```
  t.x=x++;
```

```
  t.y=y++;
```

```
  return t;
```

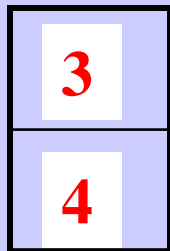
```
}
```

```
  b=a++;
```

```
  b=a.operator++(3);
```

t作为函数值返回赋给b

a



```

class incount
{
    int c1,c2;
public:
    incount(int a=0,int b=0){  c1=a; c2=b;  }
    void Show(void){cout<<"c1="<<c1<<"\t"<<"c2="<<c2<<endl;}
    incount operator++(){ c1++; c2++;    return *this;    }//前置++
    incount operator ++(int)//后置++
    { incount c;  c.c1=c1;          c.c2=c2;          c1++;  c2++;  return c;
    }
};

```

因为结果要赋值，所以函数要有返回值

```

void main(void)
{
    incount ic1(10,20),ic2(100,200), ic3 , ic4;
    ic1.Show ();
    ic2.Show ();
    ic3=++ic1;//ic3=ic1.operator++()
    ic4=ic2++;//ic4=ic4.operator++(3)
    ic3.Show ();
    ic4.Show ();
}

```

ic1.c1=11	ic3.c2=21
ic3.c1=11	ic3.c2=21
ic2.c1=20	ic2.c2=200
ic4.c1=100	ic4.c2=200

用成员函数实现运算符的重载时，**运算符的左操作数为当前对象**，并且要用到隐含的**this**指针。运算符重载函数不能定义为静态的成员函数，因为静态的成员函数中没有**this**指针。

运算符重载为友元函数

运算符重载为**成员函数**时，是由一个操作数调用另一个操作数。

A a ,b , c;

c=a+b; 实际上是**c=a.operator+(b);**

c=++a; 实际上是**c=a.operator++();**

c+=a; 实际上是**c.operator+=(a);**

重载+=

即函数的实参只有一个或没有。

友元函数是在类外的普通函数，与一般函数的区别是可以调用类中的私有或保护数据。

将运算符的重载函数定义为友元函数，参与运算的对象全部成为函数参数。

A a ,b , c;

c=a+b; 实际上是 **c=operator+(a, b);**

c=++a; 实际上是 **c=operator++(a);**

c+=a; 实际上是 **operator+=(c, a);**

对双目运算符，友元函数有2个参数，对单目运算符，友元函数有一个参数。有些运算符不能重载为友元函数，它们是：**=**，**()**，**[]**，**->****等**

格式为：

```
friend <类型说明> operator<运算符>(<参数表>)  
{.....}
```

```
c=a+b; // c=operator+( a, b)
```

```
friend A operator + (A &a, A &b)
```

```
{.....}
```

```

class A
{
    int i;
public:
    A(int a=0)    { i=a; }
    void Show(void)    {    cout<<"i="<<i<<endl;    }
    friend A operator +(A &,A &);//友元函数， 两个参数， 为引用
};

A operator +(A &a , A &b)
    {A t;  t.i=a.i+b.i;    return t; }

void main(void)
{
    A a1(10),a2(20),a3;
    a1.Show ();
    a2.Show ();
    a3=a1+a2;    //重新解释了加法， 可以直接进行类的运算
    a3.Show ();
}

```

相当于a3=operator+(a1,a2)

++为前置运算时，它的运算符重载函数的一般格式为：

```
A operator ++(A &a)
```

```
{  
    .....;  
}
```

++为后置运算时，它的运算符重载函数的一般格式为：

```
A operator ++(A &a, int)
```

```
{  
    .....;  
}
```

```
A   a, b;
```

```
b=++a;      A operator ++( A a ){ .... }
```

```
b=a++;      A operator ++(A a, int){ .... }
```

```
class A
```

```
{    int i;
```

```
public:public:
```

```
    A(int a=0)    { i=a; }
```

```
    void Show(void)    {    cout<<"i="<<i<<endl;    }
```

```
    friend A operator++(A &a){ a.i++; retrurn a;}
```

```
    friend A operator++(A &a, int n)
```

```
    { A t;        t.i=a.i;  a.i++;        return t;}
```

```
};
```

```
void main(void)
```

```
{    A a1(10),a2,a3;
```

```
    a2=++a1;  相当于a2=operator++(a1)
```

```
    a3=a1++;  相当于a3=operator++(a1,int)
```

```
    a2.Show();    a3.Show ();
```

```
}
```

```

class incount
{
    int c1,c2;
public:
    incount(int a=0,int b=0){  c1=a; c2=b;  }
    void Show(void){cout<<"c1="<<c1<<"\t"<<"c2="<<c2<<endl;}
    friend incount operator ++(incount &);//前置
    friend incount operator ++(incount &,int);//后置
};

incount operator++(incount &c)
{
    c.c1++; c.c2++; return c;}

incount operator ++(incount &c,int)
{
    incount cc;    cc=c;    c.c1++; c.c2++; return cc;}

void main(void)
{
    incount ic1(10,20),ic2(100,200), ic3 , ic4;
    ic1.Show ();
    ic2.Show ();
    ic3=++ic1;//ic3=operator(ic1)
    ic4=ic2++;//ic4=operator(ic2,  n)
    ic3.Show ();
    ic4.Show ();}

```

```

class ThreeD{
    float x,y,z;
public: ThreeD(float a=0,float b=0, float c=0){x=a;y=b;z=c;}
    friend ThreeD & operator ++(ThreeD &);//前置
    friend ThreeD  operator ++(ThreeD &,int);//后置
    void Show(){cout << "x="<<x<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n';}
};

ThreeD & operator ++(ThreeD & t)
{    t.x++;  t.y++;  t.z++;  return t;    }

ThreeD  operator ++(ThreeD &t ,int )
{    ThreeD  temp=t;        t.x++;  t.y++;  t.z++;  return temp;}

void main(void )
{
    ThreeD m1(25,50, 100),m2(1,2,3),m3;
    m1.Show();
    m3=++m1;
    m1.Show();        m3.Show();
    m3=m2++;
    m2.Show();        m3.Show();
}

```


对双目运算符，重载为成员函数时，仅一个参数，另一个被隐含；重载为友元函数时，有两个参数，没有隐含参数。

一般来说，单目运算符最好被重载为成员函数；对双目运算符最好被重载友元函数。


转换函数

转换函数就是在类中定义一个成员函数，其作用是将类转换为某种数据类型。


```
class A
{   float x, y;
    public:
        A(float a, float b){ x=a; y=b; }
};

void main(void)
{   A  a(2,3);
    cout<<a<<endl;
}
```

利用转换函数将
类A的对象a转换
成某种数据类型

A  **float**

错误！类的对象不能直接输出

A  **float**

1. 转换函数必须是类的成员函数。

格式为:

类名

关键字

欲转换类型

ClassName :: operator <type>()

{.....}

具体的转换算法

2. 转换函数的调用是隐含的，没有参数。

A :: operator float ()

{ return x+y; }

转换算法自己定义

```
class A
{
    int i;
public:
    A(int a=0) { i=a; }
    void Show(void)
    {
        cout<<"i="<<i<<endl;
    }
    operator int( ){ return i; }
};

void main(void)
{
    A a1(10),a2(20);
    cout<<a1<<endl;
    cout<<a2<<endl;
}
```

```

class Complex{
    float Real,Image;
public:
    Complex(float real=0,float image=0)
        {      Real=real;    Image=image;      }
    void Show(void)
        {cout<<"Real="<<Real<<"\t"<<"Image="<<Image<<endl; }
    operator float();    //成员函数，定义类转换 Complex—>float
};

Complex::operator float ()
{      return Real*Real+Image*Image;}

void main(void)
{      Complex c(10,20);
    c.Show ();
    cout<<c<<endl;//可以直接输出c，因为已经进行类型转换
}

```

注意，转换函数只能是成员函数，不能是友元函数。转换函数的操作数是对象。转换函数可以被派生类继承，也可以被说明为虚函数。

赋值运算符与赋值运算符重载 “=”

同类型的对象间可以相互赋值，等同于对象的各个成员的一一赋值。

```
A    a(2, 3), b;
```

```
b=a;
```

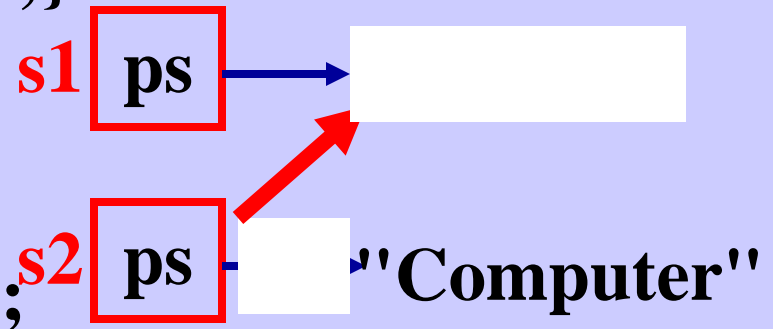
但当对象的成员中使用了动态的数据类型时(用 **new** 开辟空间)，就不能直接相互赋值，否则在程序的执行期间会出现运行错误。

```

class A{
    char *ps;
public:
    A( ){ ps=0;}
    A(char *s ){ps =new char [strlen(s)+1]; strcpy(ps,s);}
    ~A( ){ if (ps) delete ps;}
    void Show(void) { cout<<ps<<endl;}
};

void main(void )
{ A s1("China!"),s2("Computer!");
  s1.Show(); s2.Show();
  s2=s1; //相当于 s2.ps=s1.ps.
  s1.Show(); s2.Show();
}

```



s2.ps=s1.ps

首先析构s2

接着析构s1出错

这时，利用编译系统的默认赋值无法正确运行程序，必须重载赋值运算符“=”，即重新定义“=”。

格式为：

<函数类型> <ClassName>::operator=(<参数表>)

赋值运算符必须重载成员函数

成员函数作用域

函数名

函数参数

A A:: operator =(A &a)

函数返回值类型

b.operator=(a);

b=a;

左操作符调用
右操作符

```
class Sample{
    int x;
public: Sample(int i=0){x=i;}
    void disp(void){ cout<<"x="<<x<<endl;}
    void operator=(Sample &p)  { x=p.x; }
};

void main(void)
{ Sample A(20),B;
  Sample C(A);//使用缺省的拷贝构造函数
  B=A;        //使用赋值运算符重载
  B.disp();
  A.disp();
}
```

```
class A{
    char *ps;
public:    A( ){ ps=0;}
A(char *s ){ps =new char [strlen(s)+1]; strcpy(ps,s);}
~A( ){ if (ps) delete ps;}
void Show(void) { cout<<ps<<endl;}
A& operator=(A &b);
};
void main(void )
{ A s1("China!"),s2("Computer!");
  s1.Show(); s2.Show();
  s2=s1;
  s1.Show(); s2.Show();
}
```

必须重新定义 “=”

A &A::operator = (A &b)//重载赋值运算符

{ if (ps) delete [] ps;

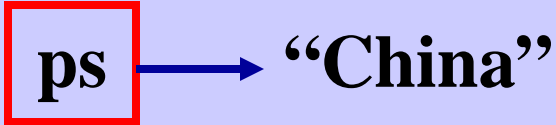
s2=s1;

if (b.ps)

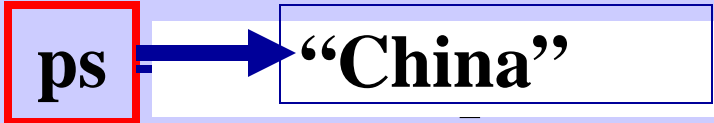
s2.operator=(s1);

{ ps = new char [strlen(b.ps)+1];

strcpy(ps, b.ps);

s1 

}

s2 

else ps =0;

return *this;

返回同种类型的引用适合于连等。

s3=s2=s1;

}

```

class A{
    char *ps;
public: A( ){ ps=0;}
    A(char *s){      ps =new char [strlen(s)+1];      strcpy(ps,s);      }
    ~A( ){ if (ps) delete ps;}
    char *GetS( ) {return ps;}
    A & operator = ( A &b);//重载赋值运算符
};

A &A::operator = ( A &b)//重载赋值运算符
{
    if ( ps ) delete [ ] ps;
    if ( b.ps) {      ps = new char [ strlen(b.ps)+1];      strcpy( ps, b.ps); }
    else          ps =0;
    return *this;
}

void main(void )
{
    A s1("China!"),s2("Computer!");
    s2=s1;
    cout <<"s1= " << s1.GetS()<<"\t";
    cout <<"s2= " << s2.GetS()<<"\n";
}

```

重新开辟内存

s2.ps重新开辟内存，存放“China”

一个字符串类

在C++中，系统提供的字符串处理能力比较弱，都是通过字符处理函数来实现的，并且不能直接对字符串进行加法、减法，字符串的拼接，字符串之间的相互赋值等操作。可以通过应用C++提供的运算符重载机制，可以提供字符串的直接操作能力，使得字符串的操作与一般的数据一样方便。

```
class String
```

```
{    int Length;//字符串长度
```

```
    char *Sp; //字符串在内存中的首地址
```

```
public:
```

```
    ....
```

```
}
```

可见，字符串类只定义了指针，并没有开辟具体的空间以存放字符串的内容，所以，无论是构造、析构还是加减等，**均需要考虑动态开辟空间的问题**，这也是字符串类的难点。

```

class String{
    int Length;                //字符串的长度
    char *Sp;                  //指向字符串的指针
public: String(){Sp=0;Length=0;} //缺省的构造函数
    String( char *s)           //以一个字符串常量作为参数
    {
        Length = strlen(s);
        Sp=new char[Length+1];
        strcpy(Sp,s);    }
    ~String(){ if(Sp) delete [ ] Sp; }
    friend String operator +(String &,String &);//友元函数重载+
    String & operator =(String &);//成员函数重载赋值=
    String (String &s); //拷贝的构造函数(必须有)
};

void main(void)
{
    String str1("China");
    String str2("CCTV");
    String str3;
    str3=str1+str2;           str2=str1;
    cout<<str3<<endl;
}

```


String & String:: operator =(String &str)

{ if (Sp)

delete []Sp;

重新定义字符串赋值B=A

Length=str.Length ;

Sp =new char[Length +1];

strcpy(Sp,str.Sp);

return *this;

}

String operator +(String &str1,String &str2)

{

String str;

str.Length=str1.Length+str2.Length;

str.Sp=new char[str.Length +1];

strcpy(str.Sp,str1.Sp);

strcat(str.Sp,str2.Sp);

return str;

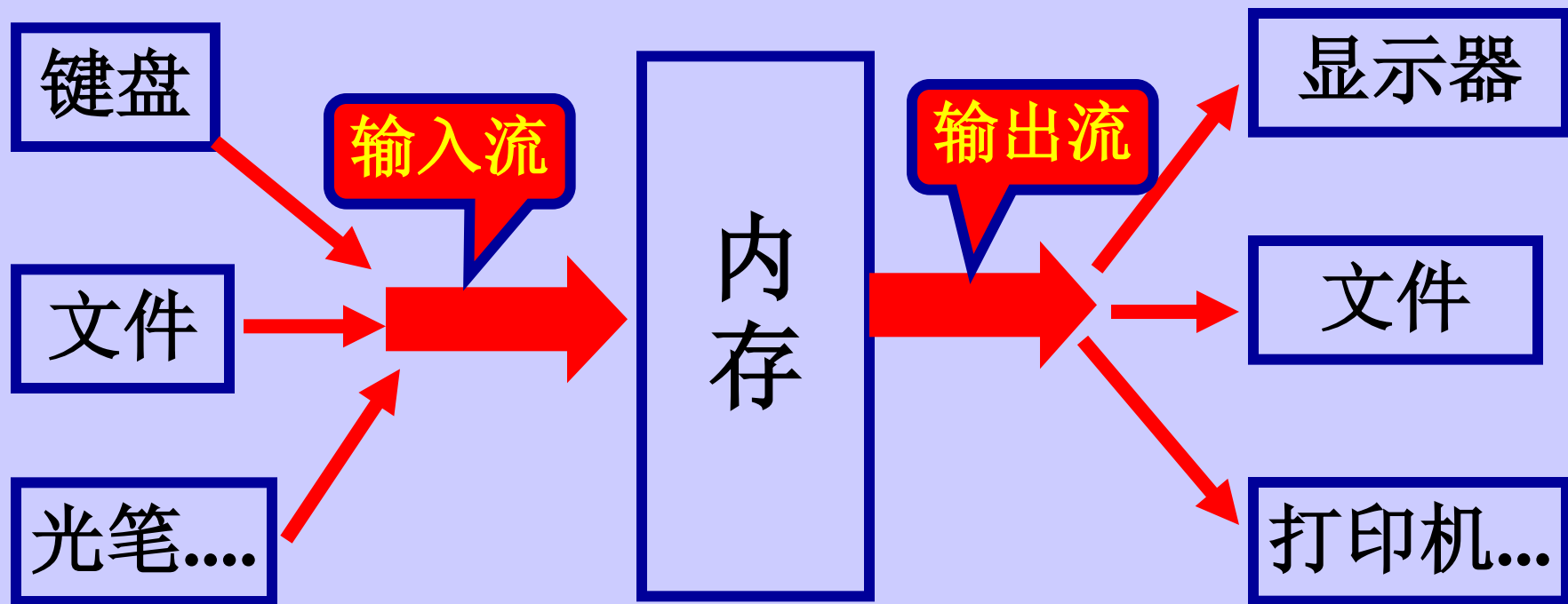
重新定义字符串连接C=A+B

}

若不定义字符串的析构函数，则可以不用定义它的拷贝的构造及赋值函数，若定义了析构函数，**必须重新定义这两个成员函数。**

原则：每个对象都有自己的独立空间。

第十四章 输入/输出流类库



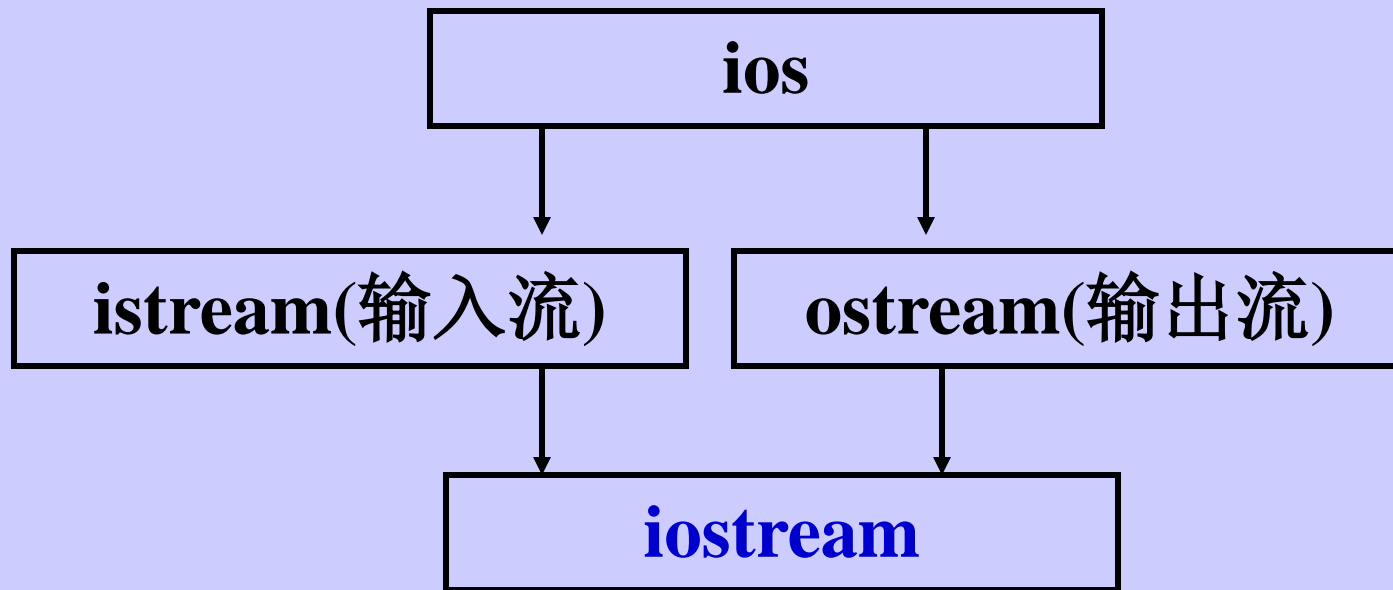
编译系统已经以运算符或函数的形式做好了对标准外设（键盘、屏幕、打印机、文件）的接口，使用时只需按照要求的格式调用即可。

`cin>>x; cout<<x;`

`cin.get(ch);`

输入输出流 (I/O Stream)

C++语言的I/O系统向用户提供一个统一的接口，使得程序的设计尽量与所访问的具体设备无关，在用户与设备之间提供了一个抽象的界面：**输入输出流**。



在“**iostream.h**”中说明

用标准流进行输入/输出时，系统自动地完成数据类型的转换。对于输入流，要将输入的字符序列形式的数据变换成计算机内部形式的数据（二进制或ASCII）后，再赋给变量，变换后的格式由变量的类型确定。对于输出流，将要输出的数据变换成字符串形式后，送到输出流（文件）中。

重载输入(提取)和输出(插入)运算符

cin>>a;

cout<<a;

class A

{ float x, y;

public:

A(float a=0, float b=0){ x=a; y=b; }

void Set(float a, float b){ x=a; y=b; }

void Show(void){ cout<<x<<'\t'<<y<<endl; }

};

void main(void)

{ A a(2,3);

a.Set(20, 30);

a.Show();

}

对象不能直接输入输出

输入对象数据

输出对象数据

在C++中允许用户重载运算符“<<”和“>>”，**实现对象的输入和输出**。重载这二个运算符时，在对象所在的类中，将重载这二个运算符的函数说明该类的**友元函数**。

重载提取运算符的一般格式为：

返回值类型 **函数名** **左操作数** **右操作数**

friend **istream & operator >>**(istream &, ClassName &);

友元函数

cin>>a;

operator>>(cin, a)

返回值类型

函数名

左操作数

右操作数

friend istream & operator >>(istream &, ClassName &);

友元函数

istream & operator >>(istream &is, ClassName &f);

cin>>a; **operator>>(cin, a)**

返回值类型：类istream的引用，cin中可以连续使用运算符“>>”。

cin>>a>>b;

第一个参数：是“>>”的左操作数cin类型，类istream的引用

第二个参数：是“>>”的右操作数，即欲输入的对象引用。

```
class A
```

```
{ float x, y;
```

```
public:
```

在类中原型说明

```
.....  
friend istream & operator >>(istream &, A &);
```

```
};
```

在类外定义函数

```
.....  
A a;  
cin>>a;  
....
```

```
istream & operator >>(istream &is, A &a)
```

```
{ cout<<" Input a:"<<endl;
```

```
is>>a.x>>a.y;
```

重新定义输入流

```
return is;
```

返回输入流

```
}
```

```

class incount{
    int c1,c2;
public:incount(int a=0,int b=0)    {    c1=a; c2=b; }
    void show(void){cout<<"c1="<<c1<<"\t"<<"c2="<<c2<<endl;}
    friend istream & operator>>(istream &,incount &);
};

```

重载输入函数原型说明

```

istream & operator>>(istream &is, incount &cc)
{
    is>>cc.c1>>cc.c2;    return is;    }

```

```

void main(void)
{
    incount x1,x2;
    x1.show ();
    x2.show ();
    cin>>x1;
    cin>>x2;
    x1.show ();
    x2.show ();
}

```

重载输入函数定义

重载输出（插入）运算符的一般格式为：

返回值类型

函数名

左操作数

右操作数

friend ostream & operator <<(ostream &, ClassName &);

友元函数

cout<<a; **operator<<**(cout, a)

与输入（提取）运算符比较：

friend istream & operator >>(istream &, ClassName &);

将输入流改为输出流。

class A

{ float x, y;

public:

在类中原型说明

.....
friend ostream & operator << (ostream &, A &);

};

在类外定义函数

.....
A a(2,3);
cout<<a;
....

ostream & operator <<(ostream &os, A &a)

{ cout<<“ The object is:”<<endl;

os<<a.x<<‘\t’<<a.y<<endl;

return os;

重新定义输出流

}

返回输出流

```

class incount{
    int c1,c2;
public: incount(int a=0,int b=0) {      c1=a;   c2=b;   }
    void show(void) {cout<<"c1="<<c1<<"\t"<<"c2="<<c2<<endl;      }
    friend istream & operator>>(istream &,incount &);
    friend ostream & operator<<(ostream &,incount &);
};

istream & operator>>(istream &is,incount &cc)
{      is>>cc.c1>>cc.c2;      return is;}

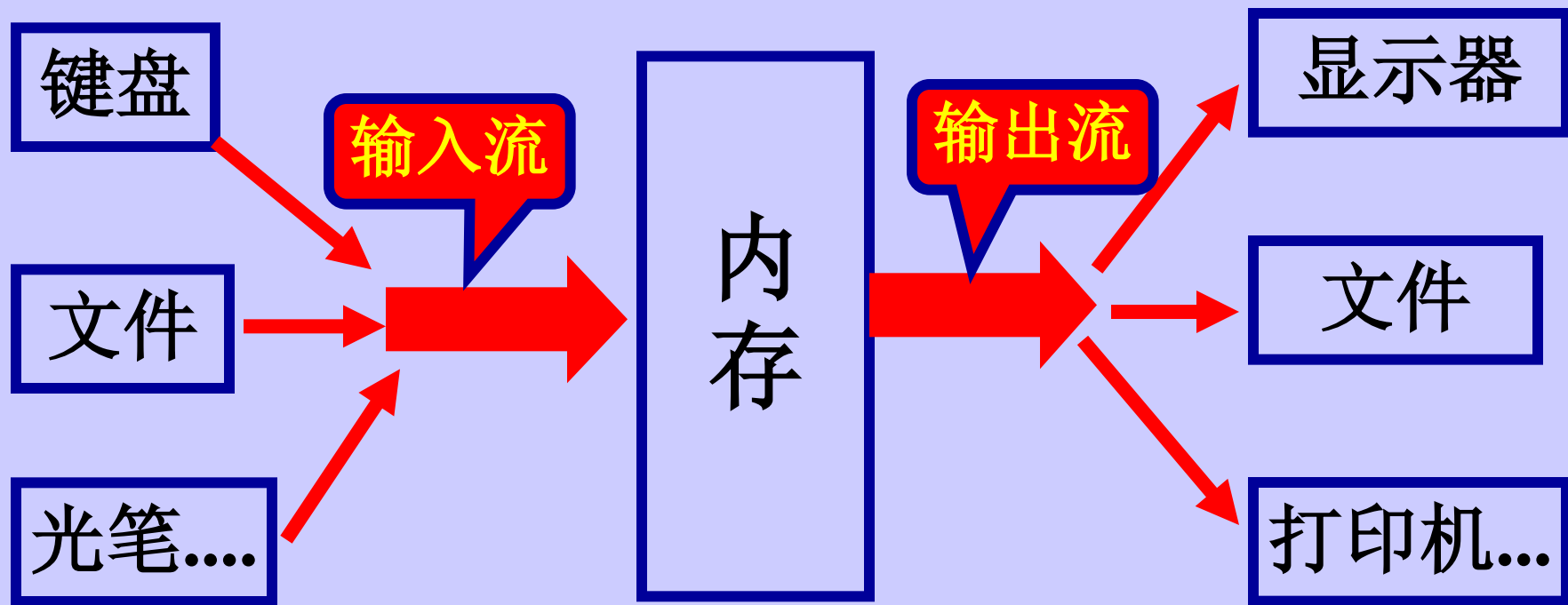
ostream &operator<<(ostream &os,incount &cc) //重载cout<<
{os<<"c1="<<cc.c1<<"\t"<<"c2="<<cc.c2<<endl; return os;}

void main(void)
{
    incount x1,x2;
    cout<<x1<<x2;//调用输出函数
    cin>>x1;      //调用输入函数
    cin>>x2;
    cout<<x1<<x2;
}

```

重载输出函数原型说明

重载输出函数定义



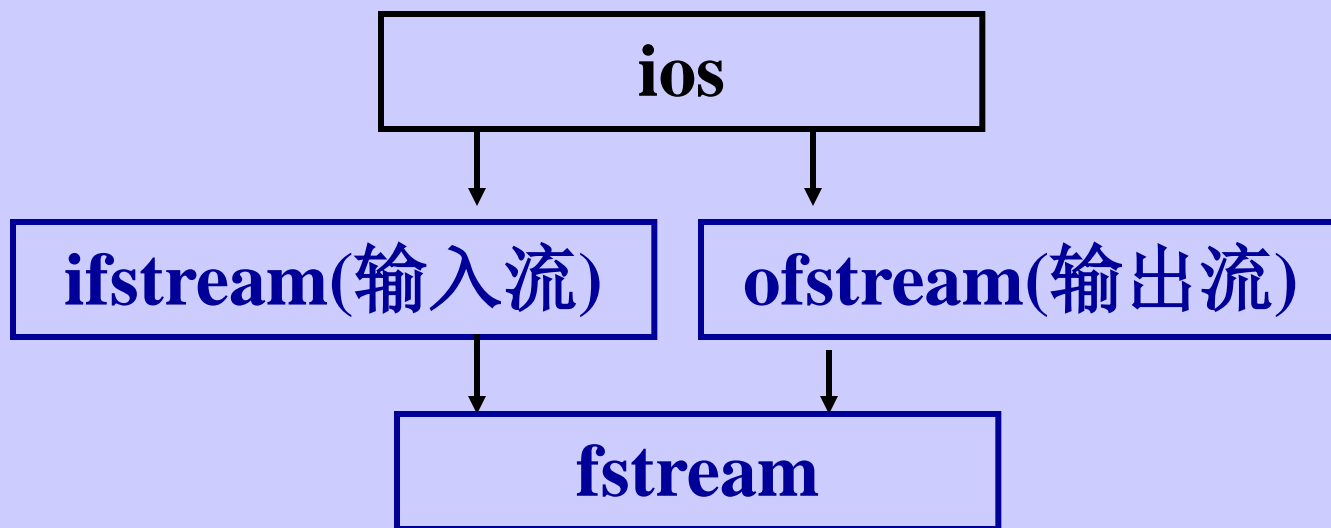
编译系统已经以运算符或函数的形式做好了对标准外设（键盘、屏幕、打印机、文件）的接口，使用时只需按照要求的格式调用即可。

`cin>>x; cout<<x;`

`cin.get(ch);`

文件流

C++在头文件**fstream.h**中定义了C++的文件流类体系,当程序中使用文件时, **要包含头文件fstream.h**。

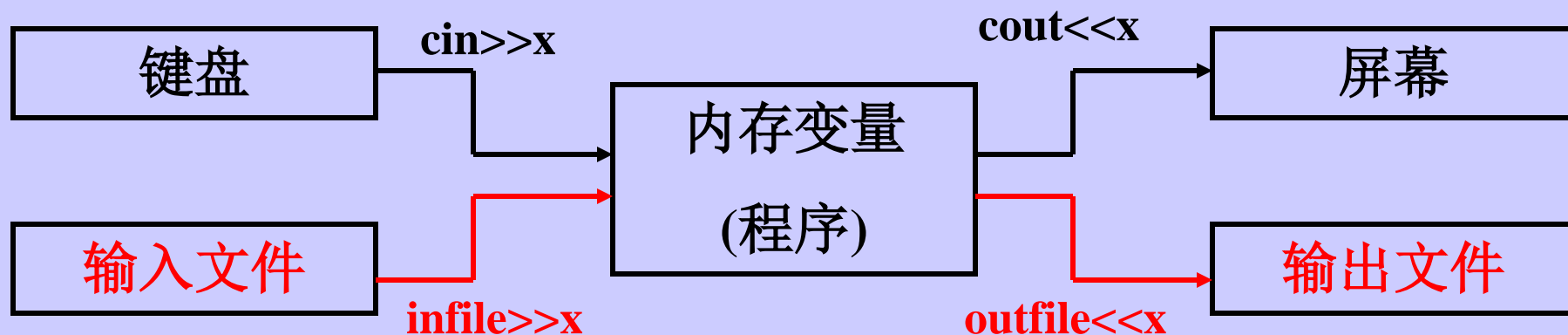


在“**fstream.h**”中说明

当使用文件时, 在程序头有: **#include<fstream.h>**

其中定义了各种文件操作运算符及函数。

程序对文本文件的操作与对键盘、显示器的操作比较：



在涉及**文本文件**的操作时，将输入文件看成键盘，将输出文件看成显示器，格式不变。**只需在程序中增加打开与关闭文件的语句。**

文件的操作

文件 { 文本文件：
以ASCII表示的文件：记事本，*.cpp等
二进制文件：
用二进制形式表示的文件：可执行程序*.EXE等

56: ASCII表示为 00110101 00110110, 占两字节

56: 二进制表示为 111000, 占六个二进制位

不同的文件操作的函数、格式不同

文本文件的打开与关闭

在文件操作前，需要将程序与被操作的文件联系起来，使程序可以“引用”文件。

在程序内定义一个文件类的对象，由该对象与文件发生联系，程序内所有的与文件的操作都是对该对象的操作。

`fstream infile, outfile;`

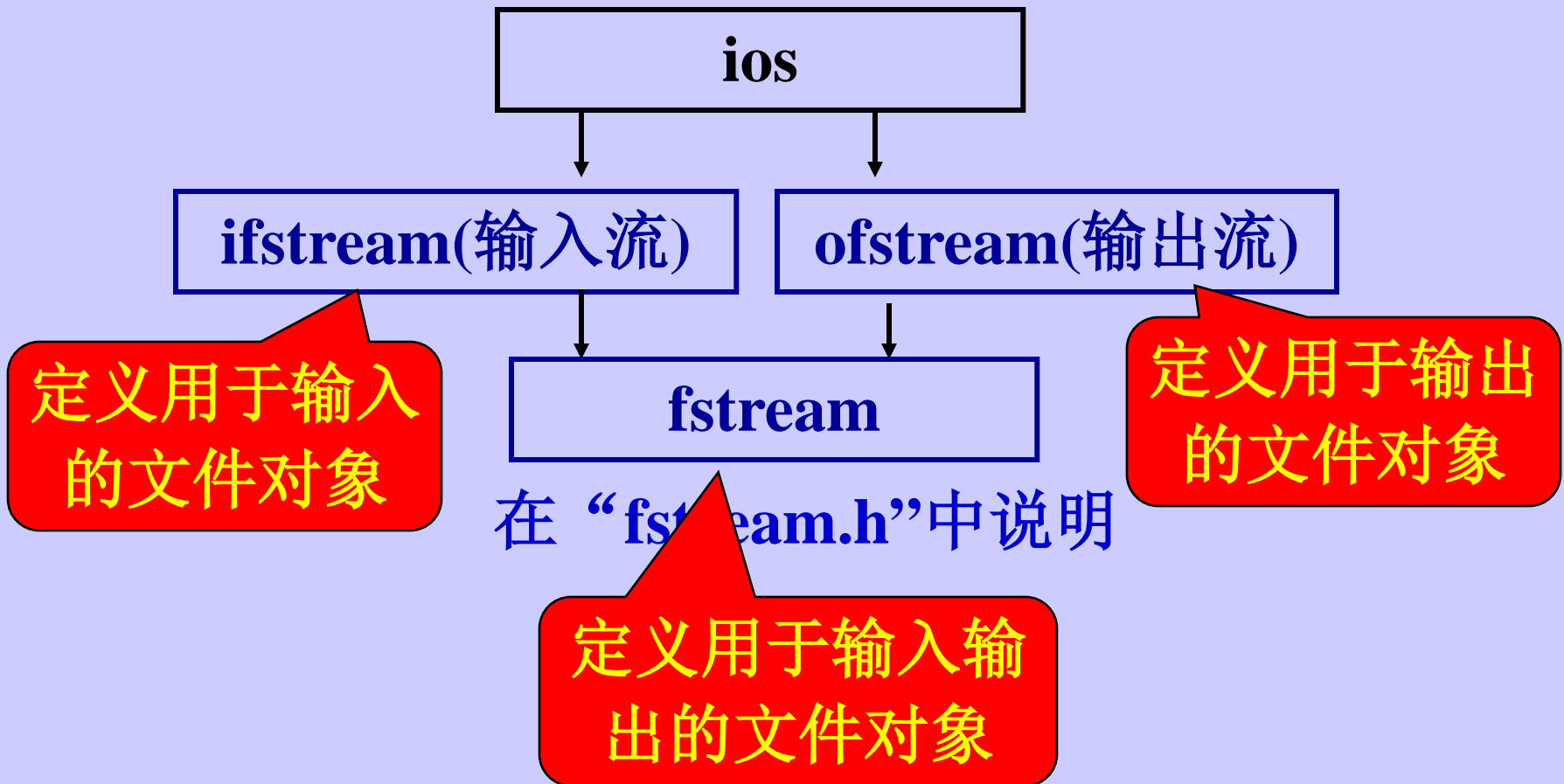
两个对象，可以联系
两个输入输出文件

`ifstream infile;`

对象只能联系输入文件

`ofstream outfile;`

对象只能联系输出文件



如何使文件类的对象与欲操作的文件发生联系？

用对象打开文件：

ifstream infile; //定义输入文件类对象

infile.open(“myfile1.txt”);//利用函数打开某一文件

打开文件的作用是，使文件流对象与要使用的文件名之间建立联系。

ofstream outfile; //定义输出文件类对象

outfile.open(“myfile1.txt”);//打开某一文件供输出

```
infile.open(“myfile1.txt”);
```

打开文件 “myfile1.txt”用于输入，并将这个文件与输入文件类对象infile建立联系，今后，在程序中，用到这个文件 “myfile1.txt”的地方就用infile来代替。

```
outfile.open(“myfile2.txt”);
```

打开文件 “myfile2.txt”用于输出，并将这个文件与输出文件类对象outfile建立联系，今后，在程序中，用到这个文件 “myfile2.txt”的地方就用outfile来代替。

如何从文件中输入输出数据？

将文件类对象看成键盘和显示器即可。

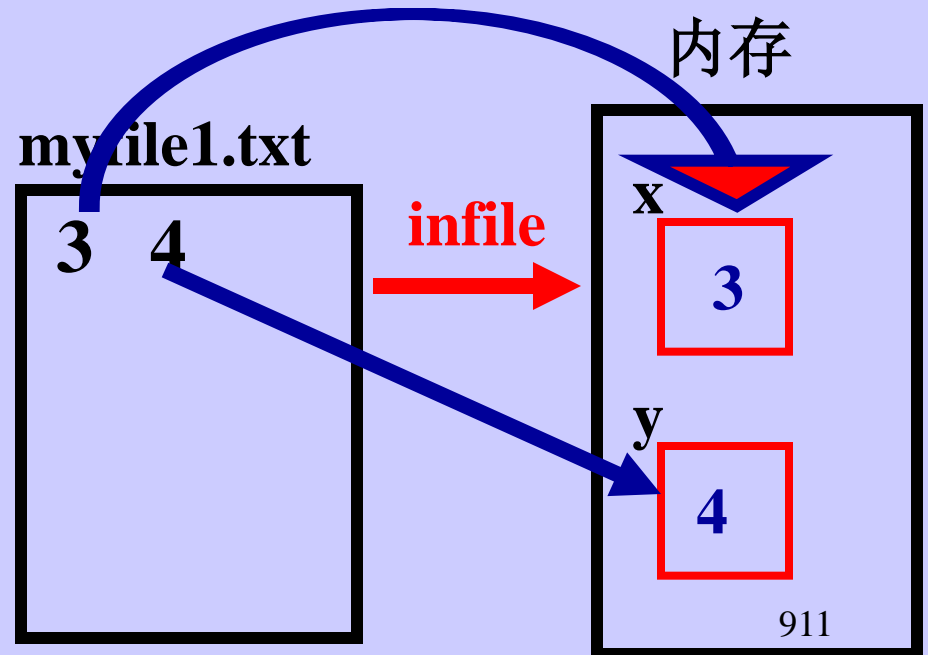
```
ifstream infile; //定义输入文件类对象
```

```
infile.open("myfile1.txt");//利用函数打开某一文件
```

```
float x, y;
```

```
infile>>x>>y;
```

用 `infile` 代替 `myfile1.txt` 进行操作。



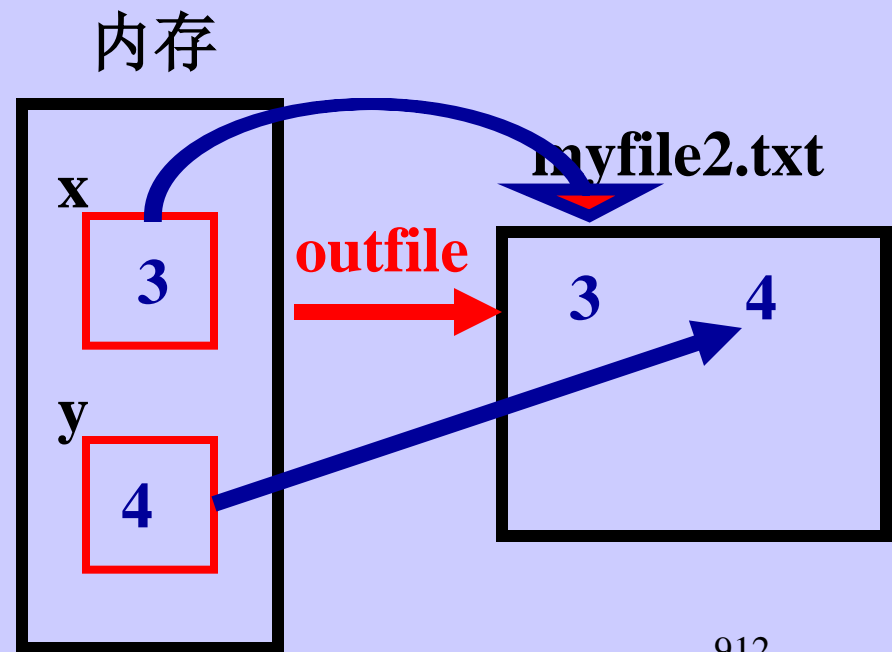
ifstream outfile; //定义输出文件类对象

infile.open("myfile2.txt");//利用函数打开某一文件

float x=3 , y=4;

outfile<<x<<"\t"<<y<<endl;

用outfile代替myfile2.txt
进行操作。



4) 用完文件后,使用成员函数关闭文件.如:

```
ifstream infile;
```

```
infile.close();
```

```
ofstream outfile
```

```
outfile.close();
```

```
infile.open("myfile1.txt");
```

```
outfile.open("myfile2.txt");
```

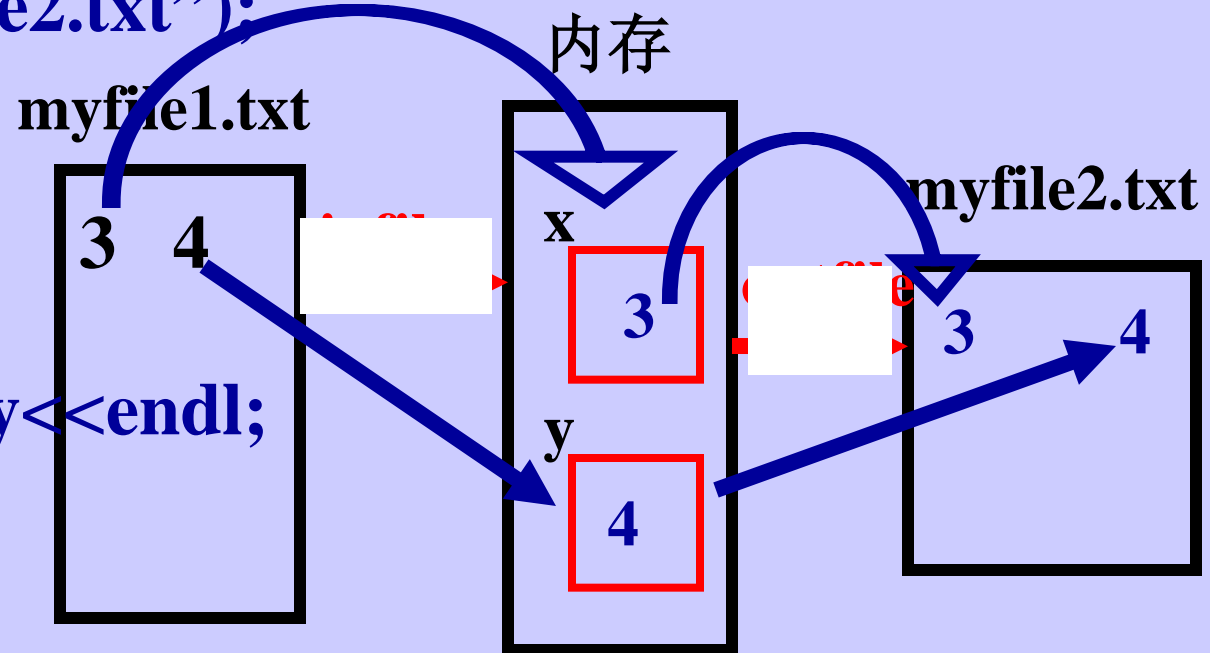
```
float x,y;
```

```
infile>>x>>y;
```

```
outfile<<x<<'\t'<<y<<endl;
```

```
infile.close();
```

```
outfile.close();
```



```
void main(void)
```

```
{ int a[10];
```

```
    ifstream infile;    //定义输入文件类
```

```
    ofstream outfile;    //定义输出文件类
```

```
    infile.open("file1.txt");    //打开一个输入文件 "file1.txt"
```

```
    outfile.open("file2.txt");    //打开一个输出文件 "file2.out"
```

```
    for(int i=0;i<10;i++)
```

```
        infile>>a[i];//将 "file1.txt"中的十个整型数输入到a[i]中
```

```
    for(i=0;i<10;i++)
```

```
        outfile<<a[i]<<"\t";//将a[i]中的十个数输出到文件 "file2.txt"中
```

```
    outfile<<endl;
```

```
    infile.close();//关闭输入文件
```

```
    outfile.close();//关闭输出文件
```

```
}
```

输入文件 “file1.txt”必须
存在，且在其中有十个整
数

当用类**fstream**定义文件对象时，该对象即能定义输入文件对象，又能定义输出文件对象，所以打开文件时，必须在成员函数**open()**中的参数中**给出打开方式（读或写）**。

fstream pfile1,pfile2;//定义了两个文件类的对象

pfile1.open(“file1.txt”, **ios::in**);//pfile1联系到 “file1.txt”,用于输入

pfile2.open(“file2.txt”, **ios::out**);//pfile2联系到 “file2.txt”,用于输出

char ch;

pfile1>>ch; //输入

pfile2<<ch; //输出

pfile1.close();

pfile2.close();

在打开文件后，都要判断打开是否成功。若打开成功，则文件流对象值为非零值；若打开不成功，则其值为0。

```
ifstream pfile1,pfile2;//定义了两个文件类的对象
```

```
pfile1.open("file1.txt", ios::in);
```

```
pfile2.open("file2.txt", ios::out);
```

```
if (!pfile1)
```

若为0，打开文件操作失败

```
{cout <<"不能打开输入文件: file1.txt"<<"\n"; exit(1);}
```

```
if (!pfile2)
```

```
{cout <<"不能打开输出文件: file2.txt"<<"\n"; exit(1);}
```

打开输入文件时，文件必须存在。

打开输出文件时，若文件不存在，则建立文件；若文件存在，则删除原文件的内容，使其成为一个空文件。

涉及到字符串的文件读写

`char ch, str[300];`

`ifstream infile("myfile1.txt");`

`ofstream outfile("myfiel2.txt");`

从键盘输入一个字符: `cin.get(ch);`

从文件输入一个字符: `infile.get(ch);`

向显示器输出一个字符: `cout.put(ch);`

向文件输出一个字符: `outfile.put(ch);`

从键盘输入一行字符: `cin.getline(str,300);`

从文件输入一行字符: `infile.getline(ch,300);`

从文件输入一字符或一行字符，当输入至文件尾时，函数返回值为0，可以据此来判断循环结束。

14-15. 实现两文件的拷贝的程序

```
void main(void)
```

```
{    char filename1[256],filename2[256];
```

```
    cout<<"Input source file name: ";
```

```
    cin>>filename1;
```

输入文件(源文件)名

```
    cout<<"Input destination: ";
```

```
    cin>>filename2;
```

输出文件(目的文件)名

```
    ifstream infile(filename1);
```

用构造函数打开文件

```
    ofstream outfile(filename2);
```

```
    char ch;
```

```
    while(infile.get(ch))
```

从源文件中读取一个字符，至文件尾停止循环

```
        outfile.put(ch);
```

```
    infile.close();
```

将该字符输出至目的文件

```
    outfile.close();
```

关闭文件

```
}
```

```
void main(void)
{
    char filename1[256],filename2[256];
    char buf[300];
    cout<<"Input source file name: ";
    cin>>filename1;
    cout<<"Input destination: ";
    cin>>filename2;
    fstream infile,outfile;
    infile.open(filename1,ios::in);
    outfile.open(filename2,ios::out);
    while(infile.getline(buf,300))
        outfile<<buf<<endl;
    outfile.close();
    infile.close();
}
```

输入文件(源文件)名

输出文件(目的文件)名

用函数打开文件

从源文件中读取一行字符，至文件尾停止循环

将该行字符输出至目的文件

关闭文件

二进制文件的读写操作

若在文件的打开方式中没有特别说明，打开的文件均为ASCII码文件，若要打开二进制文件，则要特别说明并用特定的读写函数。

```
fstream infile,outfile;
```

```
infile.open("inf1.dat", ios::in|ios::binary);
```

文件名

输入方式打开

二进制文件

```
outfile.open("outf1.dat", ios::out|ios::binary);
```

文件名

输出方式打开

二进制文件

由于二进制文件中的数据不是ASCII码，故不能直接对其读写，必须要通过特定的函数予以转换。

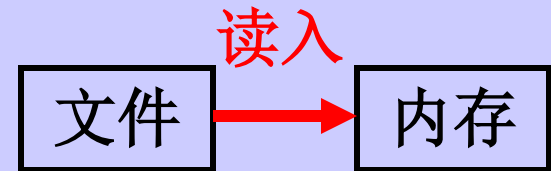
输入函数：

一次读入的字节数

`infile.read(char *, int)`

输入文件
对象名

数据进入的内存地址



`int i;`

地址要强制转换成字符型

`infile.read((char *)&i, sizeof(int));`//从文件中输入一个整型数到i

`int a[10];`

`infile.read((char *)a, 10*sizeof(int));`//从文件中输入十个整型数到a

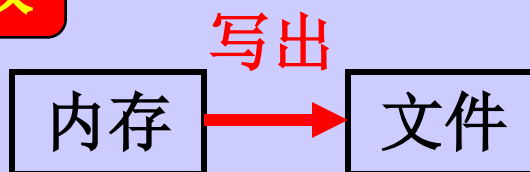
输出函数:

一次输出的字节数

`outfile.write(char *, int)`

输出文件
对象名

要输出的数据在内存中的地址



`int i=4;`

地址要强制转换成字符型

`outfile.write((char *)&i, sizeof(int));`//向文件输出一个整型数i

`int a[10]={0,1,2,3,4,5,6,7,8,9};`

`outfile.write((char *)a, 10*sizeof(int));`//向文件输出一个整型数组a

判断二进制文件是否读到了文件尾？

infile.eof()

当到达文件结束位置时，该函数返回一个非零值；**否则返回零。**

```
fstream infile;
```

```
infile.open("data1.dat",ios::in|ios::binary);
```

```
if(!infile) 判断打开是否出错
```

```
{ cout<<"Open Error!\n"; exit(1); }
```

```
char str[300];
```

```
while(!infile.eof())
```

判断是否读到了文件尾

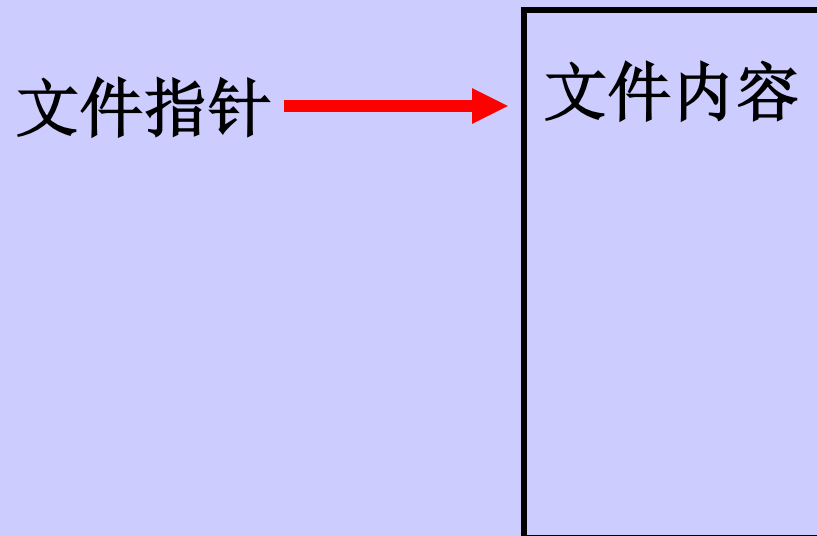
```
infile.read(str, 300);
```

```
void main(void )
{char filename1[256],filename2[256];
char buff[4096];
cout <<"输入源文件名:"; cin >>filename1;
cout <<"输入目的文件名:"; cin >>filename2;
fstream infile,outfile;
infile.open(filename1,ios::in | ios::binary);
outfile.open(filename2,ios::out | ios::binary);
int n;
while (!infile.eof()){           //文件不结束，继续循环
    infile.read(buff,4096);       //一次读4096个字节
    n=infile.gcount();            //取实际读的字节数
    outfile.write(buff,n);        //按实际读的字节数写入文件
}
infile.close();
outfile.close();
}
```

文件指针

当一打开文件，文件指针位于文件头，并随着读写字节数的多少顺序移动。

可以利用成员函数随机移动文件指针。



随机读取二进制文件

infile.seekg(int);//将文件指针移动到由参数指定的字节处

infile.seekg(100);//将文件指针移动到距离文件头100个字节处

infile.seekg(int, ios::_dir);

移动的字节数

相对位置

_dir:	beg: 文件头
	cur: 当前位置
	end: 文件尾

infile.seekg(100, ios::beg);//移动到距文件头100个字节

infile.seekg(-100, ios::cur);//移动到距当前位置前100个字节

infile.seekg(-500, ios::end);//移动到距文件尾前500个字节

```
void main(void )
{ ofstream outfile("data1.dat",ios::out| ios::binary);
  int i;
  for(i=5;i< 1000;i+=2 )
    outfile.write((char*)&i,sizeof(int)); //将奇数写入文件
  outfile.close(); //关闭文件
  ifstream f1("data.dat",ios::in| ios::binary);
  int x;
  f1.seekg(20*sizeof(int)); //将文件指针移到第20个整数的位置
  for(i=0;i<10;i++)
  {f1.read((char *)&x,sizeof(int)); //依次读出第20~29个奇数到x
    cout<< x<< '\t';
  }
  f1.close();
}
```

以读的方式打开原文件

设在缺省目录下有文件file1.txt，文件中内容为：

1 2 3 4 5 6

执行下述程序后，程序的输出是_____。

```
void main(void)
```

```
{
    fstream f1;

    int tmp, sum=0;

    f1.open("file1.txt",ios::in);

    while(f1>>tmp)

        sum+=tmp;

    f1.close( );

    cout<<sum<<endl;

}
```

如果令A,B,C,D,....., X,Y,Z这26个英文字母，分别等于百分之1,2,.....,24,25,26个数值，那么我们就得出如下有趣的结论：

HARD WORD 8+1+18+4+23+15+18+11=98%

KNOWLEDGE 96%

LOVE 54% LUCK 47%

计算一下MONEY STUDY ATTITUDE