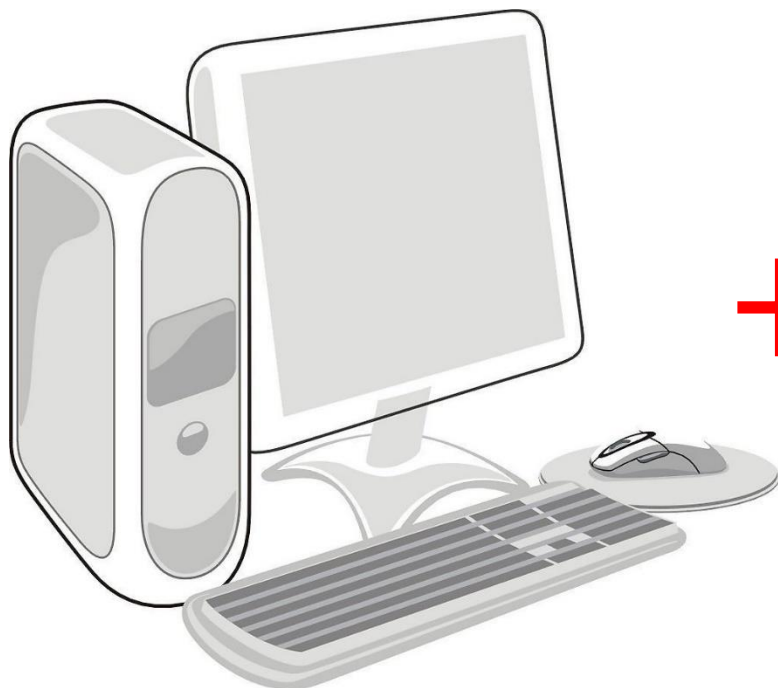


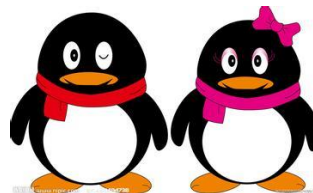
程序设计基础 — 复习课

郭延文

2017级计算机类



+



...

硬件

计算机物理构成
是“物质基础”
决定计算性能！

软件

计算机程序
像“灵魂”
定义功能！

教师是人类灵魂的“工程师”！

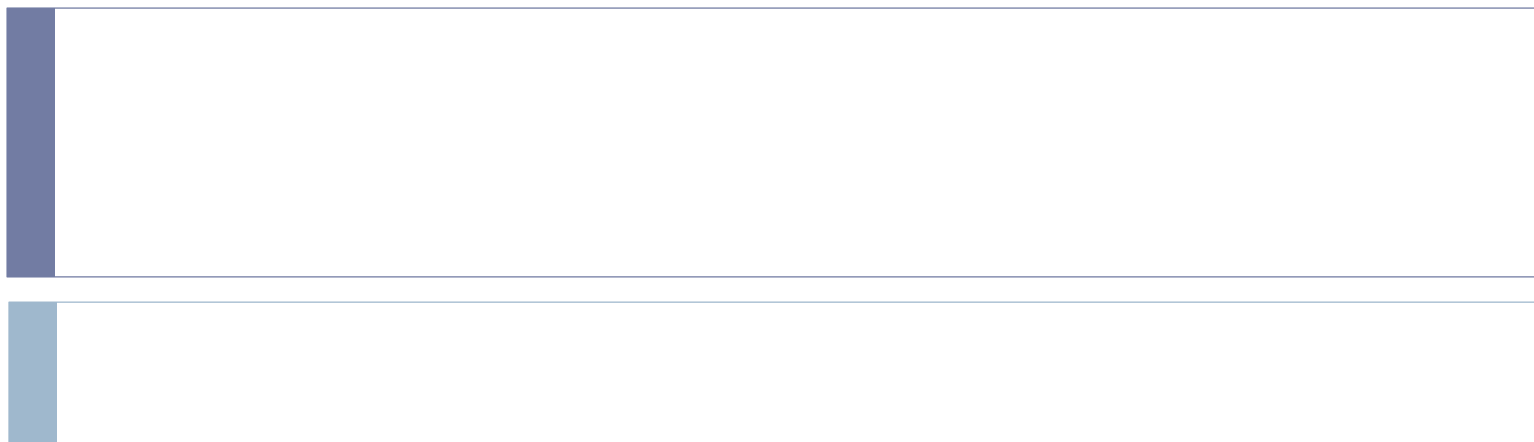
如何做计算机灵魂的工程师？

从学习程序设计基础开始吧！



第一章

冯·诺依曼体系结构



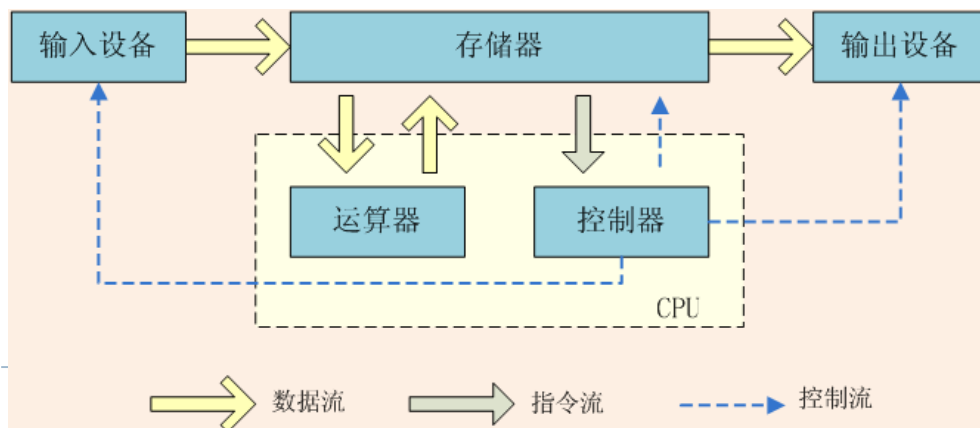
冯·诺依曼体系结构

- ▶ 逻辑上，冯·诺依曼计算机由5个单元构成
 - ▶ **输入**单元：从外界获得数据
 - ▶ **存储**单元：存储程序（指令序列）和数据
 - ▶ **运算**单元：进行算术/逻辑运算
 - ▶ **控制**单元：控制程序的执行和根据指令向其它单元发出控制信号
 - ▶ **输出**单元：向外界输出结果

 - ▶ 从ENIAC到目前最新式计算机多采用该体系结构！
-

冯·诺依曼计算机工作过程

- ▶ 借助**输入单元**把待执行的程序装入到存储单元；
- ▶ 控制单元从存储单元中逐条地读取程序中的指令执行，把其中的**计算指令**交给运算单元完成；
- ▶ 程序执行中，从输入单元或存储单元中获得所需要的数据；
- ▶ 程序执行产生的临时结果保存在存储单元中，程序的最终执行结果通过**输出单元**输出



冯·诺依曼计算机的物理组成

执行计算机指令。包含**控制器**、**运算器**以及**寄存器**

中央处理器
(CPU)

内存
(Memory)

存储运行中的计算机程序和正在使用的数据，由**存储单元**构成

总线

外设
(Devices)

输入/输出和**外部存储**

永久性存储程序和**数据**

外存

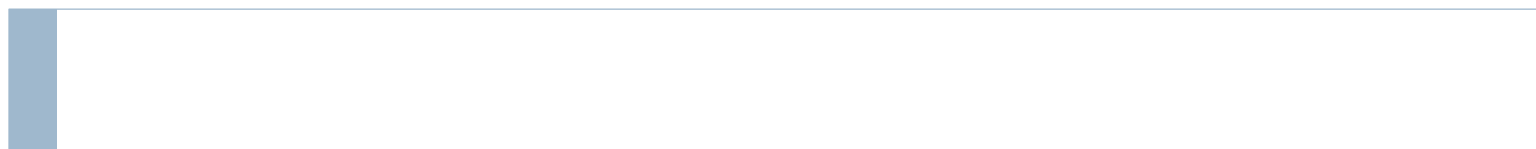
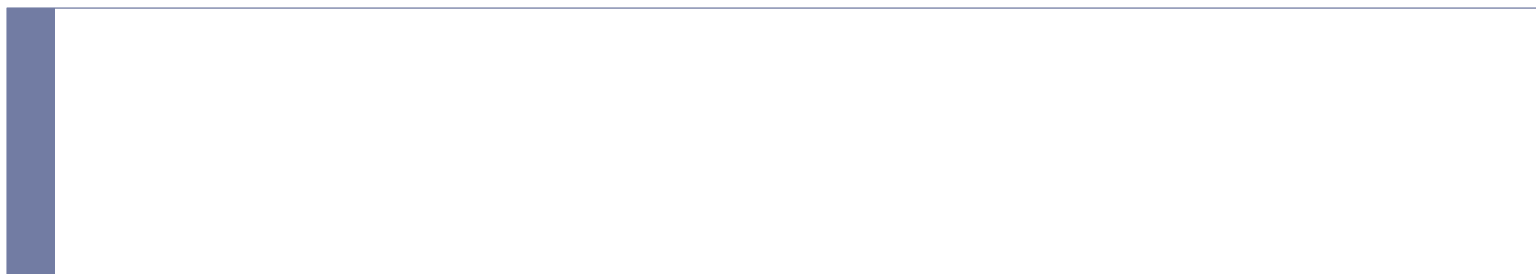
输入/输出

软盘、硬盘、光盘、闪存盘等

键盘、显示器、打印机、鼠标器等

第二章

基本数据类型



数据类型

- ▶ 数据是程序的一个重要组成部分，每个数据都属于某种数据类型。
- ▶ 一种数据类型可以看成由两个集合构成：
 - ▶ 值集：规定了该数据类型能包含哪些值（包括这些值的结构）。
 - ▶ 操作（运算）集：规定了对值集中的值能实施哪些运算。
 - ▶ 例如：整数类型就是一种数据类型，它的值集就是由整数所构成的集合，它的操作集包括：加、减、乘、除等运算。



C++ 数据类型

基本数据类型

- 整数类型
- 实数类型
- 字符类型
- 逻辑类型
- 空值类型

构造数据类型

- 枚举类型
- 数组类型
- 结构与联合类型
- 指针类型
- 引用类型

抽象数据类型

- 类
- 派生类

数据在C++程序中的表示

- ▶ 在程序中，数据以两种形式出现：
 - ▶ **常量**：用于表示在程序执行过程中不变（或不能被改变）的数据。
 - ▶ **变量**：用于表示在程序执行过程中可变的数据。
 - ▶ 例如，在计算圆的周长表达式 $2*PI*r$ 中，
 - ▶ 2和圆周率PI是常量。
 - ▶ 半径r是变量，它的值可能在程序运行时从用户处得到，或由程序的其它部分计算得到。

符号常量

- ▶ 符号常量是指先通过常量定义给常量取一个名字，并可指定一个类型；然后，在程序中通过常量名来使用这些常量。
- ▶ 符号常量的定义格式为：

`#define <常量名> <值>`

或

`const <类型名> <常量名>=<值>;`

例如：

`#define PI 3.1415926`

或，

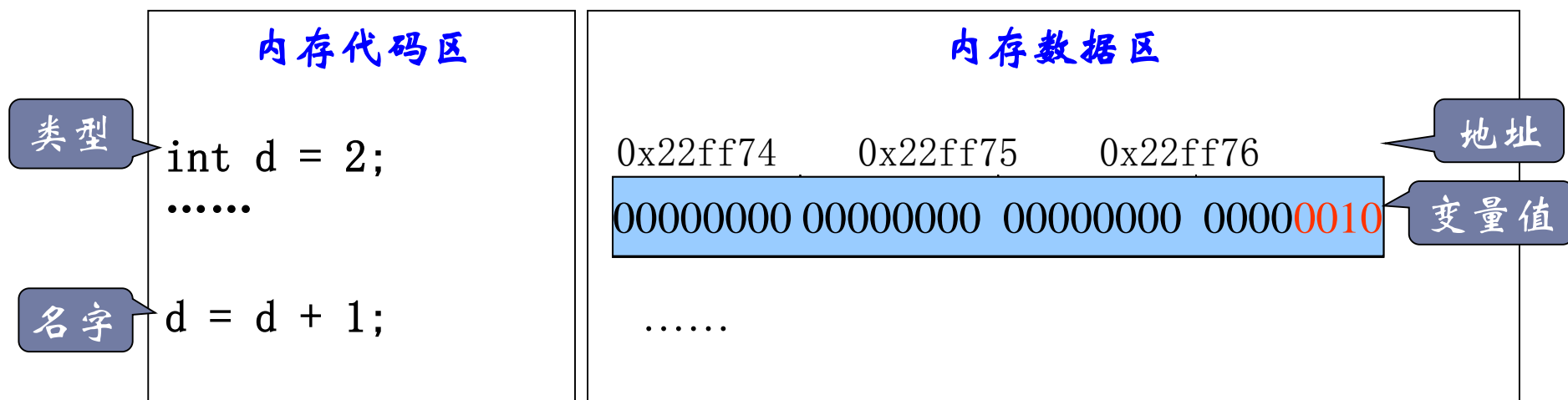
`const double PI=3.1415926;`

- ▶ 符号常量的使用：

`2*PI*r ;`（此为语句；也可作为表达式参加运算）

变量的属性

- ▶ 程序执行到变量定义处，系统会为变量分配一定大小的空间，用以存储变量的值。
- ▶ 存储空间里起初是一些0/1组成的无意义的值，可以通过赋值或输入值来获得有意义的值；存储空间由地址来标识，一般由系统自动管理。



用户定义变量类型和名字

系统决定地址，存储二进制值

整数类型

- ▶ 整数类型用于描述通常的整数。根据精度分成：
 - ▶ int
 - ▶ short int 或 short
 - ▶ long int 或 long
 - ▶ 一般情况下，
“short int”的范围 \leq “int”的范围 \leq “long int”的范围
 - ▶ 具体大小由实现决定，例如
 - ▶ int 占4个字节，一般由计算机的字长决定
 - ▶ short int 占2个字节 (-32768~32767)
 - ▶ long int 占4个字节 (-2147483648~2147483647)
 - ▶ 在计算机内部，整数一般用补码表示。
-

例：整型数据范围溢出示例

```
#include <stdio.h>
int main()
{
    int a = 2147483647, b = 1;
    printf("%d \n", a + b);
    return 0;
}
```

-2147483648

- 如果将输出格式符%d改成%u（即按unsigned int型数据输出），则结果为2147483648。

2147483647的补码

01 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11

+1

10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- 2147483648的补码

补码

▶ 补码的简单求法

▶ **正数**：同原码

▶ **负数**：符号位同原码，其余各位取反，末位加1

真值X:	+1010111	-1010111
[X] _原 (8位):	01010111	11010111
[X] _原 (16位):	0000000001010111	1000000001010111
[X] _补 (8位):	01010111	10101001
[X] _补 (16位):	0000000001010111	1111111110101001

实数类型(浮点型)

- ▶ 实数类型又称浮点型，它用于描述通常的实数。根据精度可分为：

- ▶ float （单精度型）
- ▶ double （双精度型）
- ▶ long double （长双精度型）



例：实浮点型数据的精度问题

```
#include<stdio.h>
int main( )
{
    float x = 0.1f;
    float y = 0.2f;
    float z = x + y;
    if(z == 0.3)
        printf("They are equal.\n");
    else
        printf("They are not equal! The value of z is %.10f", z);
    return 0;
} //输出 “They are not equal! The value of z is 0.30000000119”
```



$$\begin{aligned}
(0.1)_{10} &= (0.0001100110011\dots)_2 \\
&= (0.01100110011001100110011)_2 \times 2^{-2} \\
(0.2)_{10} &= (0.0011001100110\dots)_2 \\
&= (0.11001100110011001100110)_2 \times 2^{-2} \\
(0.1+0.2)_{10} &= (1.00110011001100110011001)_2 \times 2^{-2} \\
(0.3)_{10} &= (0.01100110011\dots)_2 \\
&= (1.10011001100110011001101)_2 \times 2^{-2}
\end{aligned}$$

为什么？

1: 二进制

2: 浮点数的存储格式

特别注意!

- ▶ 对浮点数进行关系操作时, 往往得不到正确的结果, 应避免对两个浮点数进行 “==” 和 “!=” 操作
 - ▶ $x == y$ 可写成: $\text{fabs}(x-y) < 1e-6$
 - ▶ $x != y$ 可写成: $\text{fabs}(x-y) > 1e-6$
 - ▶ $z == 0.3$ 可写成: $\text{fabs}(z-0.3) < 1e-6$

字符类型

- ▶ 字符类型用于描述文字类型数据中的一个字符。
- ▶ 字符在**计算机内**存储的是它的**编码**（对应的“**机器数**”）

A **01000001**

a **01100001**

ASCII码表八、十六、十进制对照表

<http://www.feiesoft.com/00007/>

41	21	33	!	141	61	97	a
42	22	34	"	142	62	98	b
43	23	35	#	143	63	99	c
44	24	36	\$	144	64	100	d
45	25	37	%	145	65	101	e
46	26	38	&	146	66	102	f
47	27	39	`	147	67	103	g
50	28	40	(150	68	104	h
51	29	41)	151	69	105	i
52	2a	42	*	152	6a	106	j
53	2b	43	+	153	6b	107	k
54	2c	44	,	154	6c	108	l
55	2d	45	-	155	6d	109	m
56	2e	46	.	156	6e	110	n
57	2f	47	/	157	6f	111	o
60	30	48	0	160	70	112	p
61	31	49	1	161	71	113	q
62	32	50	2	162	72	114	r
63	33	51	3	163	73	115	s
64	34	52	4	164	74	116	t
65	35	53	5	165	75	117	u
66	36	54	6	166	76	118	v
67	37	55	7	167	77	119	w
70	38	56	8	170	78	120	x
71	39	57	9	171	79	121	y
72	3a	58	:	172	7a	122	z
73	3b	59	;	173	7b	123	{
74	3c	60	<	174	7c	124	
75	3d	61	=	175	7d	125	}
76	3e	62	>	176	7e	126	~
77	3f	63	?	177	7f		

例： 格式符%d将各种类型的数据显示为十进制整数

```
#include <stdio.h>
int main( )
{
    printf("ASCII code of the character is: %d \n", 'A');
    printf("ASCII code is: %d \n", 65);
    printf("ASCII code in decimal is: %d \n", 0x41);
    printf("ASCII code of the character is: %d \n", '7');
    printf("ASCII code of the character is: %d \n", '\a');
    return 0;
}
```

```
ASCII code of the character is: 65
ASCII code is: 65
ASCII code in decimal is 65
ASCII code of the character is: 55
ASCII code of the character is: 7
```

例：数字字符与整数的区别示例

```
#include<stdio.h>
int main( )
{
    int i = 3;
    char ch = '3';
    printf("10i = %d, 10ch = %d \n", 10 * i, 10 * ch);
    return 0;
}
```

实际应用中，数字字符更多是用来描述字符串的一分子，比如，“以3结尾的学号”，而不是用来参加数值运算。

30, 510

3

'3'

00000011

00110011

例： 字符型变量值的输入及其参与关系和算术操作 对输入的大写字母A-Z， 转化为小写字母（重点！）

```
#include <stdio.h>
int main( )
{  char ch;
   do
   {
       printf("Input Y or N (y or n) :");
       scanf("%c", &ch); // ch = getchar();
       if(ch >= 'A' && ch <= 'Z')
           ch += 32;
       // 可改写为: ch = (ch >= 'A' && ch <= 'Z') ? ch + 'a' - 'A' : ch;
       printf("%c", ch);
   } while(ch != 'y' && ch != 'n');
   if(ch == 'y')
       .....
   else
       }
}
```

逻辑类型

- ▶ 逻辑类型用于描述“真”和“假”这样的逻辑值，分别表示条件的满足和不满足
- ▶ 在C++中，逻辑类型用`bool`表示，它的值只有两个：`true`和`false`，分别对应“真”和“假”
- ▶ 在大多数的C++实现中，`bool`类型的值一般占用一个字节的空
间，`true`存储的是1，`false`存储的是0



枚举类型

- ▶ 程序员用关键词enum构造出来的数据类型，程序员构造这种类型时，要逐个列举出该类型变量所有可能的取值。根据构造的枚举类型再定义具体的枚举变量。

- ▶ 比如，

```
enum Color {RED, YELLOW, BLUE};
```

```
Color c1, c2, c3;
```

- ▶ Color是构造的枚举类型名，花括号里列出了Color类型变量可以取的值，它们又叫枚举符或枚举常量（标识符的一种，习惯用大写字母的英文单词表示）
- ▶ c1、c2和c3是三个类型为Color的枚举变量，这三个变量的取值都只能是RED、YELLOW或BLUE。

▶ 常见的枚举类型还有：

- ▶ `enum Weekday {SUN, MON, TUE, WED, THU, FRI, SAT};`
- ▶ `enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};`

例：写程序求部分基本数据类型占的位数

...

```
unsigned char uint8 = 0;    signed char int8 = 0;
unsigned short uint16 = 0;  signed short int16 = 0;
unsigned int uint32 = 0;    signed int int32 = 0;
unsigned long ulong = 0;    float fp32 = 0;
double fp64 = 0;
```

```
printf("unsigned char is %d bit\n\r", sizeof(uint8)*8);
printf("signed char is %d bit\n\r",  sizeof(int8)*8);
printf("unsigned short is %d bit\n\r", sizeof(uint16)*8);
printf("signed short is %d bit\n\r",  sizeof(int16)*8);
printf("unsigned int is %d bit\n\r",   sizeof(uint32)*8);
printf("signed int is %d bit\n\r",     sizeof(int32)*8);
printf("unsigned long is %d bit\n\r",  sizeof(ulong)*8);
printf("float fp32 is %d bit\n\r",     sizeof(fp32)*8);
printf("double fp64 is %d bit\n\r",    sizeof(fp64)*8);
```

...



例：写程序求部分基本数据类型占的位数

运行结果：

- ▶ unsigned char is 8 bit
- ▶ signed char is 8 bit
- ▶ unsigned short is 16 bit
- ▶ signed short is 16 bit
- ▶ unsigned int is 32 bit
- ▶ signed int is 32 bit
- ▶ unsigned long is 32 bit
- ▶ float fp32 is 32 bit
- ▶ double fp64 is 64 bit

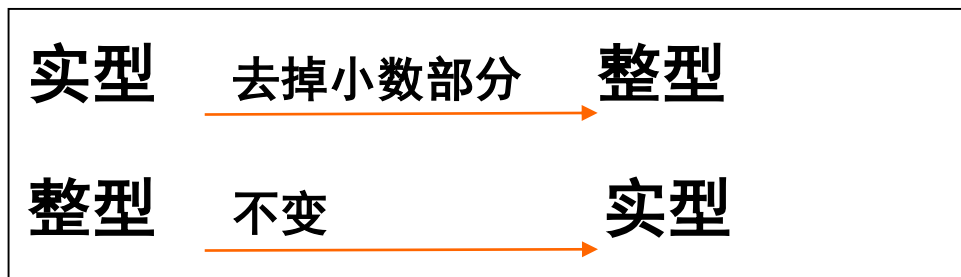


基本类型的转换

- ▶ 程序执行过程中，要求参加双目操作的两个操作数类型相同；当不同类型的操作数进行这类操作时，会进行类型转换，即操作数的类型会转换成另一种数据类型
 - ▶ 常指基本类型，不是基本类型的两个不同类型操作数往往不能转换
- ▶ 类型转换方式有两种：
 - ▶ 隐式类型转换：由系统自动按一定规则进行的转换
 - ▶ 显式类型转换：由程序员在程序代码中标明，进行强制转换
- ▶ 不管哪一种方式，类型转换都是“临时”的，即在类型转换过程中，操作数本身的类型并没有被转换，只是参加当前操作的数值被临时“看作”另一种类型的数值而已。

隐式类型转换规则小结

- ▶ C程序运行期间，函数和变量的类型以定义的类型为准

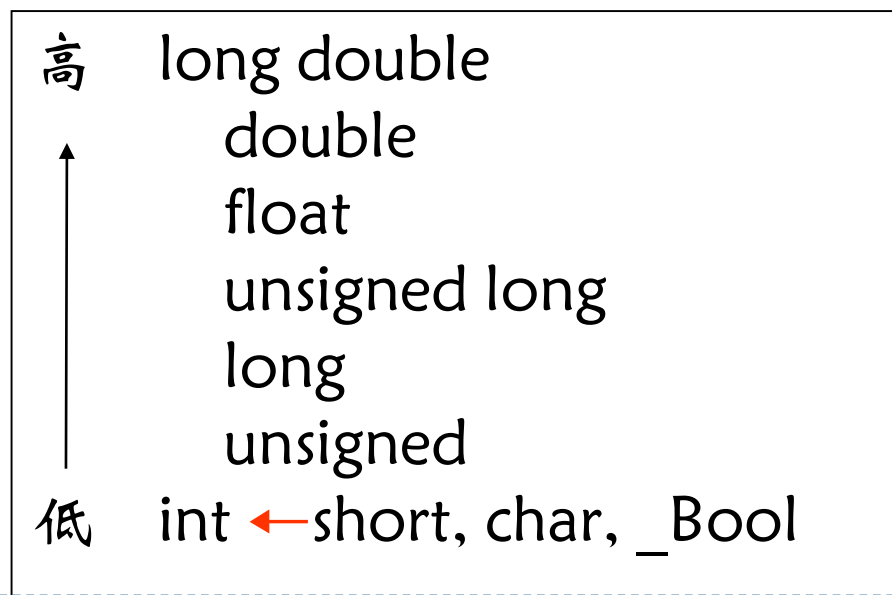


- ▶ 逻辑操作：

- ▶ 非0数→true， 0→false

- ▶ 其他：

- ▶ 精度低→精度高



例：隐式类型转换存在的问题示例

```
#include<stdio.h>
```

```
int main( )
```

```
{
```

```
    int i = -10;
```

```
    unsigned int j = 3;
```

```
    if(i + j < 0)
```

```
        printf("-7\n");
```

//应改成if(i+(int)j<0) printf("-7\n");

```
    else
```

```
        printf("error.\n");
```

```
    if(i < j)
```

```
        printf("i<j\n");
```

//应改成if(i<(int)j)

printf("i<j\n");

```
    else
```

```
        printf("i>j\n");
```

```
    return 0;
```

```
}
```

利用显式类型转换，则结果可显示-7和i<j。



显式（强制）类型转换

- ▶ 隐式类型转换有时不能满足要求，于是C语言提供了显式类型转换机制，由程序员用类型关键词明确地指出要转换的类型，**强制**系统进行类型转换：

`i + (int) j ;`

建议进行显式类型转换！

- ▶ 避免理解的“模糊”（提升程序可读性）和执行的“问题”
- ▶ 此外，对于一些对操作数类型有约束的操作，可以用显式类型转换保证操作的正确性。
- ▶ 比如，C语言中的求余数运算要求操作数必须是整型数据，
“`int x = 10%3.4;`”应改为
“`int x = 10%(int)3.4;`”，否则编译会出错。



例：类型转换后的数据精度问题

```
//...  
int main()  
{  
    double a=3.3, b=1.1;  
    int i = a/b;  
    printf("%d \n", i);  
    return 0;  
}
```

2

```
//...  
int main()  
{  
    double a=3.3, b=1.1;  
    printf("%.0f \n", a/b);  
    return 0;  
}
```

3

例：生成随机数（几次操作得到不一样的随机数）

```
▶ #include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define random(x) (rand() % x)

void main()
{
    srand((int)time(0)); // srand(time(NULL));
    for(int x=0;x<10;x++)
        printf("%d/n",random(100));
}
```



C++操作符的种类

- ▶ 算术操作符
- ▶ 关系与逻辑操作符
- ▶ 位操作符
- ▶ 赋值操作符
- ▶ 其它操作符



模运算：求余数 %

- ▶ C语言中用%表示计算两个整数相除的余数
 - ▶ 操作数只能为整数
 - ▶ 较小的数模较大的数，结果一定为较小的数
 - ▶ 对于正整数 m 和 n ， $(m/n)*n+m\%n$ 一般等于 m
- ▶ 求余数运算在一些实际问题的处理中，常常能发挥比较巧妙的作用



例：求所有的三位水仙花数

- ▶ 设计程序求所有的三位水仙花数（一个三位水仙花数等于其各位数字的立方和，比如， $153 = 1^3 + 3^3 + 5^3$ ），要求不用嵌套的循环。
- ▶ 分析：利用除法和求余数运算，可以分离出三位数每一位上的数字。

```
for(int n = 100; n < 999; n++)  
{  
    int i = n / 100;           //百位数字  
    int j = n / 10 % 10;      //十位数字  
    int k = n % 10;           //个位数字  
    if(n == i * i * i + j * j * j + k * k * k)  
        printf("%d\t", n);  
}
```


自增/自减操作

▶ **++/--**：变量的值自增1/自减1

▶ **前缀**——前缀操作符置于操作数的前面，除了修改操作数的值外，操作结果是操作后操作数的值

```
i = 3;
++i;           //相当于i += 1, 也即i = i + 1, i的值变为4
--i;           //相当于i -= 1, 也即i = i - 1, i的值变回为3
j = ++i;       //相当于i=i+1; j=i; 则i、j的值均变为4
```

先自增；
再赋值

▶ **后缀**——后缀操作符置于操作数的后面，除了修改操作数的值外，操作结果是操作前操作数的值

```
m = 3;
m++;           //则m的值变为4
m--;           //则m的值变回为3
n = m++;       //相当于n=m; m=m+1; n的值为3, m的值变为4;
```

先赋值
再自增；

if (n == 0) V.S. if (0 == n)

- ▶ 为了避免将比较操作符==误写成=，即少写一个等于号，程序员常常将常量写在该操作符的左边，这样，编译器可以帮助发现这个错误。比如

```
if(n == 0) //如果误写成if(n = 0)，编译器不会报错，且n变为0
    n++;
else
    n = 1 / n;
```

可以写成：

```
if(0 == n) //如果误写成if(0 = n)，编译器会报错，因为不能给常量赋值
    n++;
else
    n = 1 / n;
```

逻辑操作符

- ▶ 逻辑操作符实现逻辑运算，用于复杂条件的表示。包括：

- ▶ `!`（逻辑非）
- ▶ `&&`（逻辑与）
- ▶ `||`（逻辑或）

- ▶ 操作数为bool类型，例如：

- ▶ `!(a > b)`
- ▶ `(age < 10) && (weight > 30)`
- ▶ `(ch < '0') || (ch > '9')`

- ▶ 结果为bool类型

`!true -> false`
`!false -> true`

`false && false -> false`
`false && true -> false`
`true && false -> false`
`true && true -> true`

`false || false -> false`
`false || true -> true`
`true || false -> true`
`true || true -> true`

短路求值

▶ **&&** 和 **||**

- ▶ 如果第一个操作数已能确定操作结果，则不再计算第二个操作数的值。
- ▶ “不成立 **&&** x” 的结果为 不成立
- ▶ “成立 **||** x” 的结果为 成立

▶ 短路求值

- ▶ 能够提高逻辑操作的效率
- ▶ 有时能为逻辑操作中的其他操作提供一个“**卫士** (guard)”
 - ▶ $(\text{number} \neq 0) \ \&\& \ (1/\text{number} > 0.5)$ 在 number 为 0 时，不会进行除法运算。

位操作

- ▶ 将整型操作数看作二进制位序列进行操作，包括
 - ▶ 逻辑位操作
 - ▶ 移位操作
- ▶ 序列的长度与机器及操作数的类型有关（以32位机、int类型为例）
- ▶ 操作数如果是负数，则以补码形式参与位操作

位操作 \sim $\&$ \wedge $|$ \ll \gg

- ▶ 位操作(按位取反、与、或、异或)的操作数是整数的二进制位列表示
- ▶ 位操作速度快, 节省存储空间
- ▶ 注意逻辑位操作与逻辑操作区别

~ 8 为 -9
 $\sim 0\dots 01000$
 $(1\dots 10111)$
 $!8$ 为 0(false)

$8 \& 1$ 为 0
 $0\dots 0\ 1000$
 $\& 0\dots 0\ 0001$
 $(0\dots 0\ 0000)$

$8 \& \& 1$ 为 1(true)

$8 | 1$ 为 9
 $0\dots 0\ 1000$
 $| 0\dots 0\ 0001$
 $(0\dots 0\ 1001)$

$8 || 1$ 为 1(true)

移位操作

- ▶ 将左边的整型操作数对应的二进制位序列进行左移或右移操作，移动的次数由右边的整型操作数决定
- ▶ 包括
 - ▶ << (左移)
 - ▶ >> (右移)

左右移举例

5 << 2 的结果为

(0000 0000 0000 0000 0000 0000 0000 0101)

20 (0000 0000 0000 0000 0000 0000 0001 0100)

5 >> 2 的结果为

(0000 0000 0000 0000 0000 0000 0000 0101)

1 (0000 0000 0000 0000 0000 0000 0000 0001)



操作符的目

▶ 一个操作符能连接的操作数的个数

▶ 算术操作符

▶ 取正/取负操作符

▶ 自增/自减操作符

▶ 关系操作符

▶ 逻辑操作符

▶ 位操作符

▶ 赋值操作符

▶ 条件操作符

双目

单目

单目

双目

双目/单目

双目/单目

双目

三目

连接两个操作数

连接一个操作数

连接三个操作数

操作符的优先级 (precedence)

- ▶ 是指操作符的优先处理级别
- ▶ C语言将基本操作符分成若干个级别
 - ▶ 第1级为最高级别，第2级次之，以此类推。
- ▶ C语言操作符的优先级一般按“单目、双目、三目、赋值”依次降低，其中双目操作符的优先级按“算术、移位、关系、逻辑位、逻辑”依次降低

()	高
单目操作符	
* / %	
+ -	
<< >>	
> < >= <=	
== !=	
&	
^	
&&	
? :	
=	
,	低

操作符的结合性

- ▶ 是指操作符与操作数的结合特性
- ▶ 包括：
 - ▶ 左结合：先让左边的操作符与最近的操作数结合起来，
 $3 > 2 > 1$
 - ▶ 右结合：先让右边的操作符与最近的操作数结合起来，
 $a = b = 3$

具体结合性参加课本2.5.2的表格！

思考1:

- ▶ 分析下面程序片段的执行结果:

```
int a =12;
```

```
a = a += a -= a*a;
```

```
printf("%d", a);
```

-264

- ▶ $x = (a=3), 6*a$;的值为:

18



思考2:

$c = a-- ? ++a : --a;$

相当于:

```
if(a--) // 判断a是否为零, 判断后将a自减1;
{
    c = ++a; // a自加1后赋值
}
else
{
    c = --a; // a自减1后赋值
}
```



建议!!!

- ▶ 以上要掌握知识点，但写程序不建议写“难以理解”的表达式！

例如：

```
int a = 12;
```

```
a = a += a -= a*a;
```

建议写为：

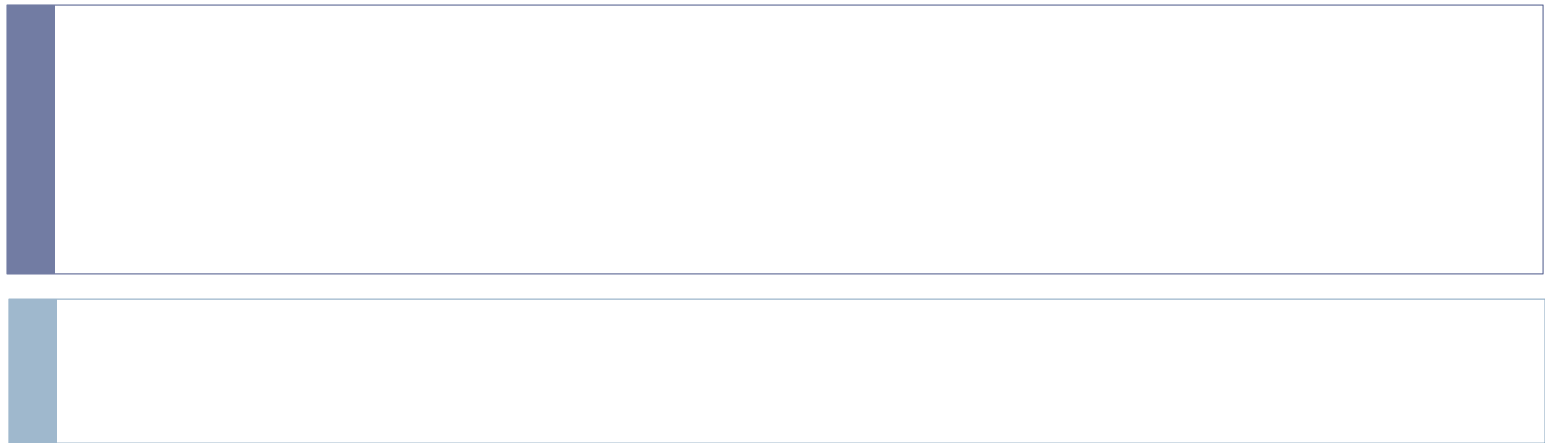
```
a -= a*a;
```

```
a += a;
```



第三章

程序的流程控制方法 (Flow Control)



流程

▶ 语句的执行次序

- ▶ 一般情况下，程序中的语句按照排列次序顺序执行；
- ▶ 如果遇到控制语句，比如C语言中的if语句和while语句等，则会改变执行的顺序；
- ▶ 控制语句改变执行次序有分支和循环两种基本方式。
- ▶ 逻辑上：顺序、分支和循环是程序中的三种基本流程。
- ▶ 一个程序的总流程由基本流程衔接而成。

分支流程的基本语句 **if...else**

```
if (<条件P>) // if 和括号间可以加一空格
{
    ...
}
else
{
    ...
}
```

- ▶ <条件P>一般是带有关系或逻辑操作的表达式，表达式的值为true，则认为条件成立，执行if子句，否则执行else子句
 - ▶ 关系操作：比如 $n > 10$
 - ▶ 逻辑操作：比如 $n > 10 \ \&\& \ n < 100$

if, else if, ..., else

```
if (P1)
    do A1;
else if (P2) // 不满足P1, 满足P2
    do A2;
else if (P3) // 不满足P1, P2, 满足P3,
    do A3;
else        // 不满足P1, P2, P3
    do A4;
```



switch语句

- ▶ switch后面圆括号中的操作结果，与case后面的整数或字符常量进行匹配
- ▶ 如果匹配成功，则从冒号后的语句开始执行，执行到右花括号结束该流程；
- ▶ 如果没有匹配的值，则执行default后面的语句或不执行任何语句（default分支可省略），然后结束该流程。

```
switch (week)    //该行没有分号
{
    case 0: printf("Sunday \n"); break;
    case 1: printf("Monday \n"); break;
    .....
    default: printf("error \n");
}
```

switch语句可以嵌套

- ▶ 这时，内层switch语句里的“break;”语句只能将程序的流程转向内层switch语句的结束处，不能控制外层switch语句的流程

```
switch(x)
{
    case 0: printf("xy = 0 \n"); break;           // 外层分支
    case 1:
        switch(y)
        {
            case 0: printf("xy = 0 \n"); break; // 内层分支
            case 1: printf("xy = 1 \n"); break; // 内层分支
            default: printf("xy = %d \n",y); // 内层分支
        } ◀
        break;                                   // 外层分支
    default: printf("error! \n");                // 外层分支
} ◀
```

循环流程

- ▶ 计算机在完成一个任务时，常常需要对相同的操作重复执行多次（每次操作数的值可能有所不同）
 - ▶ 循环流程用于这种“重复性”计算场合，是程序设计的一种重要流程
 - ▶ 循环流程由循环语句控制
-

while语句

- ▶ 实现循环流程控制的一种语句，其格式为：

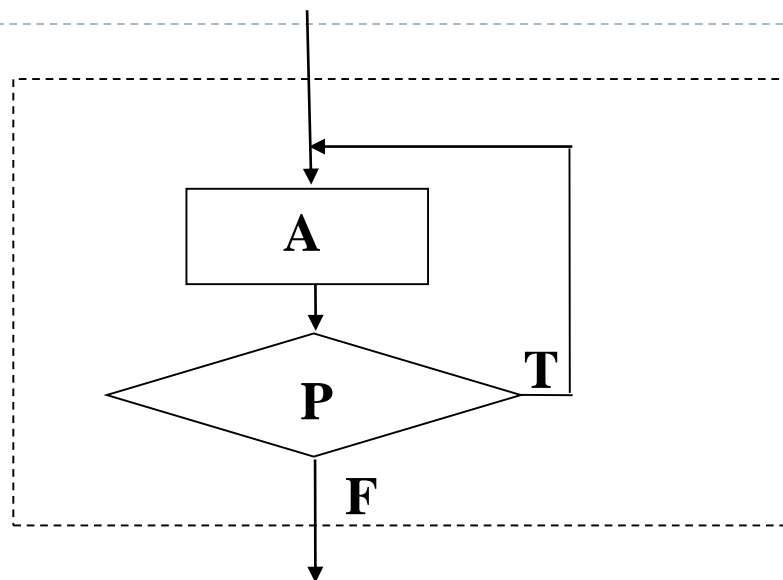
```
while(<条件P>)  
{  
    <任务A>  
}
```

- ▶ 条件P一般是带有关系或逻辑操作的表达式
- ▶ 任务A是while语句的子句，通常是一个复合语句，可以写在右圆括号的后面，但：建议写在下一行，并缩进
- ▶ 条件P表达式的值为true，则认为条件成立，执行任务A，并再次判断条件P；否则不执行任务A，也不再判断条件P，语句结束

do-while语句

“无论如何先干一次”！

```
do  
{  
    <任务A>  
}  
while(<条件P>);
```



- ▶ 任务A是do-while语句的子句，是一个复合语句，写在do的下一行，并缩进。
- ▶ 条件P一般是带有关系或逻辑操作的表达式
 - ▶ 表达式的值为true，则认为条件成立，执行任务A，并再次判断条件P；
 - ▶ 否则不执行任务A，也不再判断条件P

for语句

▶ 语法形式:

```
for (<循环变量赋初值>; <条件P>; <表达式E>)  
{  
    <任务A'>  
}
```

▶ 先对循环变量赋初值

再判断条件P

- ▶ 当条件P成立时, 执行任务A'
- ▶ 计算表达式E
- ▶ 判断条件P, 如此循环往复; 当条件P不成立时, 结束该流程

循环嵌套的优化

- ▶ 嵌套循环中，如果有可能，应将长循环放在最内层，短循环放在最外层，以减少CPU跨切循环层的次数，提高程序的运行效率：

```
for(row = 0; row < 100; row++)           //长循环在最外层，效率低
    for(col = 0; col < 5; col++)
        sum += row * col;
```

- ▶ 可以改写为：

```
for(col = 0; col < 5; col++)             //长循环在最内层，效率高
    for(row = 0; row < 100; row++)
        sum += row * col;
```

循环可以用OpenMP优化(如果每次循环的计算互相独立)

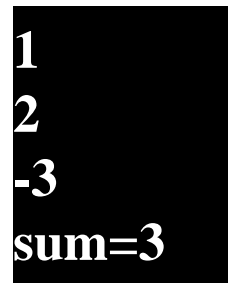
控制循环流程用while、do-while还是for语句？

- ▶ 从表达能力上讲，上述三种语句是等价的，都可嵌套，它们之间可以互相替代
- ▶ 对于某个具体的问题，用其中的某个语句来描述可能会显得比较自然和方便，使用三种语句的一般原则：
 - ▶ 计数控制的循环，用for语句
 - ▶ 事件控制的循环，一般使用while或do-while语句
 - ▶ 如果循环体至少执行一次，则使用do-while语句
- ▶ 由于for语句的结构性较好，循环流程的控制均在循环顶部统一表示，更直观（即for语句可显式地给出：循环变量初始化、循环结束条件以及下一次循环准备）很多情况下都采用for语句。

循环流程的折断(break)

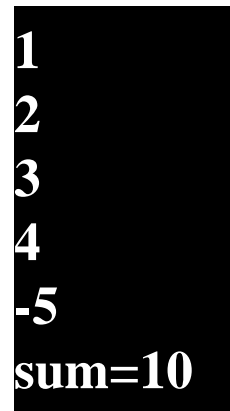
- ▶ C语言中，break语句（通常在循环体中与if结合使用）可以控制循环流程的折断。
- ▶ 执行到break语句，就立即结束循环流程。
- ▶ 例如：输入10个数，依次求和，遇到输入的负数或0就提前终止求和的过程。

```
int d, sum = 0, i = 1;
while(i <= 10)
{
    scanf("%d", &d);
    if (d <= 0)
        break;
    sum += d;
    i++;
}
printf("sum: %d \n", sum);
```



1
2
-3
sum=3

这次运行，循环只执行了2次。



1
2
3
4
-5
sum=10

这次运行，循环只执行了4次。

continue语句：循环流程的接续

- ▶ C语言中提供了一种continue语句（通常在循环体中与if结合使用），可以控制循环流程的接续
- ▶ 执行到continue语句，就跳过循环体后部的任务，流程转向循环的头部
- ▶ 对于while、do-while语句是进行下一次条件判断，对于for语句是计算表达式E

例：输入10个正数，求和；如果输入负数，则忽略该负数

等价于：

```
int d, sum = 0;
int i = 1;
while (i <= 10)
{
    scanf("%d", &d);
    if (d <= 0)
        continue;
    sum += d;
    i++;
}
printf("sum: %d \n", sum);
```

```
1
2
-3
4
5
6
7
8
9
10
11
sum=63
```

```
int d, sum = 0;
int i = 1;
do
{
    scanf("%d", &d);
    if (d <= 0)
        continue;
    sum += d;
    i++;
}
while (i <= 10);
printf("sum: %d \n", sum);
```

goto语句 - 无条件转移控制语句

- ▶ goto语句的格式如下：

goto <语句标号>;

- ▶ <语句标号>为标识符，其定义格式为：

<语句标号>: <语句>

- ▶ goto的含义是：程序转移到带有<语句标号>的语句



goto语句一个用途：跳出多重循环

```
int i, j;  
for (i=1; i<=10; i++)  
    for (j=1; j<=10; j++)  
        if (i*j == 50)  
            goto END;  
END: cout << i << "," << j << endl;  
//输出5 , 10
```

```
int i, j;  
for (i=1; i<=10; i++)  
    for (j=1; j<=10; j++)  
        if (i*j == 50)  
            break;  
cout << i << "," << j << endl;  
//输出11,5
```



C/C++语句的分类



典型例题！

例：用牛顿迭代法求 $\sqrt[3]{a}$

- ▶ 计算 $\sqrt[3]{a}$ 的牛顿迭代公式为：

$$x_{n+1} = \frac{1}{3} \left(2x_n + \frac{a}{x_n^2} \right)$$

- ▶ 取 x_0 为 a （任何值都可以），依次计算 x_1 、 x_2 、 \dots ，直到：
- ▶ $|x_{n+1} - x_n| < \varepsilon$ （ ε 为一个很小的数，可设为 10^{-6} ）时为止， x_{n+1} 即为所求值。

典型例题！

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{ const double eps=1e-6; //一个很小的数
  double a, x1, x2; //x1和x2分别用于存储最新算出的两个值
  cout << "请输入一个数: ";
  cin >> a;
  x1 = a; //第一个值取a
  x2 = (2*x1+a/(x1*x1))/3; //计算第二个值
  while (fabs(x2-x1) >= eps);
  { x1 = x2; //记住前一个值
    x2 = (2*x1+a/(x1*x1))/3; //计算新的值
  }
  cout << a << "的立方根是: " << x2 << endl;
  return 0;
}
```

$$x_{n+1} = \frac{1}{3} \left(2x_n + \frac{a}{x_n^2} \right)$$

例：计算级数和的值

- 编写程序计算当 $x=0.5$ 时下述级数和的近似值，使其误差（前后两次级数和的差）小于某一指定的值 ϵ （例如： $\epsilon=0.000001$ ）

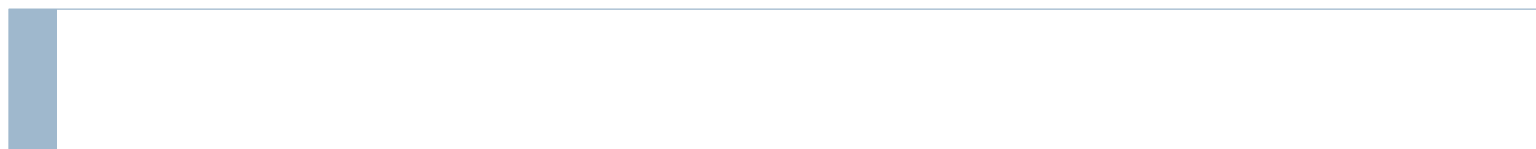
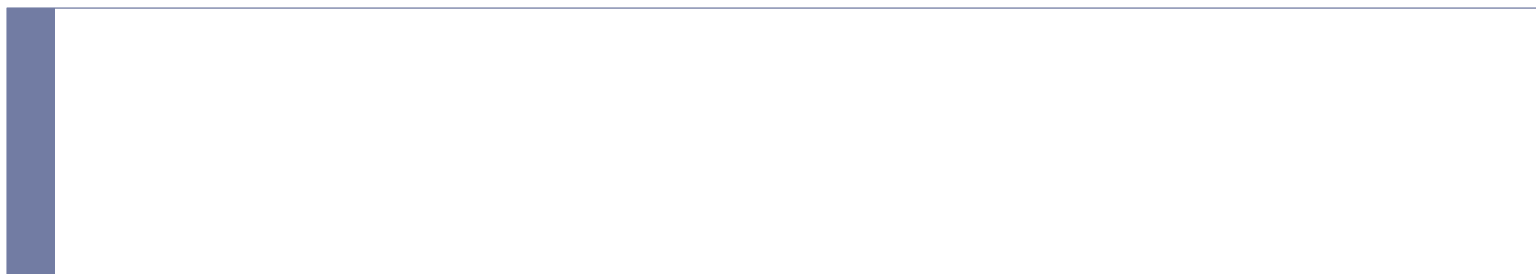
$$x - (x^3)/(3*1!) + (x^5)/(5*2!) - (x^7)/(7*3!) + \dots$$

即：

$$x - \frac{x^3}{3*1!} + \frac{x^5}{5*2!} - \frac{x^7}{7*3!} + \dots$$

第四章

过程抽象与封装-函数



基于过程抽象的程序设计

- ▶ 人们在设计一个复杂的程序时，经常会用到功能分解和复合两种手段：
 - ▶ 功能分解：在进行程序设计时，首先把程序的功能分解成若干子功能，每个子功能又可以分解成若干子功能，等等，从而形成了一种自顶向下（top-down）、逐步精化（step-wise）的设计过程。
 - ▶ 功能复合：把已有的（子）功能逐步组合成更大的（子）功能，从而形成一种自底向上（bottom-up）的设计过程。



C/C++函数

▶ 函数是C/C++提供的用于实现子程序的语言成分。

▶ 函数的定义：

 <返回值类型> <函数名>(<形式参数表>)

{

 <函数体>

}

▶ <返回值类型>描述了函数返回值的类型，

 ▶ 可以为任意的C++数据类型。

 ▶ 当返回值类型为void时，它表示函数没有返回值。

▶ <函数名>用于标识函数的名字，用标识符表示。

▶ <形式参数表>描述函数的形式参数，由零个、一个或多个形参说明（用逗号隔开）构成，形参说明的格式为：

 <类型> <形参名>

函数返回值（通过return返回）的作用

▶ 带回计算结果

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

▶ 确定被调用函数的运行状态；

根据返回值执行下面的步骤

▶ 返回给调用函数的结果，用于函数功能的实现和通信

▶ 有时也为了判断函数是否正确执行而设置返回值

```
int func(...)
```

```
{
```

```
    if(case1)
```

```
        return 1;
```

```
    else if (case2)
```

```
        return 2;
```

```
    else if (case3)
```

```
        return 3;
```

```
    else
```

```
        return 0;
```

```
}
```

```
// 根据函数调用的返回值，知道f中发
```

```
// 生了什么，是case1, 2, 3, or
```

```
// nothing!
```

函数返回值（通过return返回）的作用

例如：一只体温计

```
int Fthermometer(...)
```

```
// 测量体温的函数
```

根据Fthermometer()返回的温度，确定下一步的应对措施



函数的调用

- ▶ 对于定义的一个函数，必须要调用它，它的函数体才会执行。
- ▶ 除了函数main外，程序中对其它函数的调用都是从main开始的。main一般是由操作系统来调用。

- ▶ 函数调用的格式如下：

＜函数名＞（＜实在参数表＞）；

- ▶ ＜实在参数表＞由零个、一个或多个表达式构成（逗号分割）
- ▶ 实参的个数和类型应与相应函数的形参相同。类型如果不同，编译器会试图进行隐式转换，转换规则是把实参类型转换成形参类型。



函数调用的执行过程

- ▶ 计算实参的值；
- ▶ 把实参分别传递给被调用函数的形参；
- ▶ 执行函数体；
- ▶ 函数体中执行return语句返回函数调用点，调用点获得返回值（如果有返回值）并执行调用之后的操作。
- ▶ 可以把有返回值的函数调用作为操作数放在表达式中参加运算：

$x + \text{power}(x, y) * z$

值传递

- ▶ 在函数调用时，采用类似变量初始化的形式把实参的值传给形参。
- ▶ 函数执行过程中，通过形参获得实参的值，
- ▶ 函数体中对形参值的改变不会影响相应实参的值。

函数声明

- ▶ 程序中调用的所有函数都要有定义；如果函数定义在其它文件（如：C++的标准库）中或定义在本源文件中使用点之后，则在调用前需要对被调用的函数进行声明。

- ▶ 函数声明的格式如下：

〈返回值类型〉 〈函数名〉(〈形式参数表〉); //函数原型

或

extern 〈返回值类型〉 〈函数名〉(〈形式参数表〉);

- ▶ 在函数声明中，〈形式参数表〉中可以只列出形参的类型而不写形参名

变量的局部性

- 在C++中，根据变量的定义位置，把变量分成：
 - **局部变量**是指在复合语句中定义的变量，它们只能在定义它们的复合语句（包括内层的复合语句）中使用
 - **全局变量**是指在函数外部定义的变量，它们一般能被程序中的所有函数使用（静态的全局变量除外）



变量的生存期（存储分配）

- 把程序运行时一个变量占有内存空间的时间段称为该变量的生存期。
 - ▶ **静态**：从程序开始执行时就进行内存空间分配，直到程序结束才收回它们的空间；**全局变量具有静态生存期**
 - ▶ 具有静态生存期的变量，如果没有显式初始化，系统将把它们初始化成0
 - ▶ **自动**：内存空间在程序执行到定义它们的复合语句（包括函数体）时才分配，当定义它们的复合语句执行结束时，它们的空间将被收回。**局部变量和函数的参数一般具有自动生存期**
 - ▶ **动态**：内存空间在程序中显式地用new操作或malloc库函数分配、用delete操作或free库函数收回。**动态变量具有动态生存期**

存储类修饰符

- ▶ 在定义局部变量时，可以为它们加上存储类修饰符 (auto, static, register) 来显式地指出其生存期
 - ▶ auto: 使局部变量具有自动生存期。局部变量的默认存储类为 auto。
 - ▶ static: 使局部变量具有静态生存期。它只在函数第一次调用时进行初始化，以后调用中不再进行初始化，它的值为上一次函数调用结束时的值。
 - ▶ register: 局部变量仍是自动生存期，但由编译程序根据CPU寄存器的使用情况来决定是否存放在寄存器中（未必放于内存）
-

举 例

```
void f()
{ auto int x=0;      //auto一般不写
  static int y=1;
  register int z=0;
  x++; y++; z++;
  cout << x << y << z << endl;
}
// 在main函数中调用f();
```

* 第一次调用f时，输出：1 2 1

* 第二次调用f时，输出：1 3 1



static的作用

- ▶ 使局部变量具有静态生存期
 - ▶ 它只在函数第一次调用时进行初始化，以后调用中不再进行初始化，它的值为上一次函数调用结束时的值
- ▶ 使全局标识符具有文件作用域
 - ▶ 在全局标识符的定义中加上static修饰符，则该全局标识符就成了具有文件作用域的标识符，它们只能在定义它们的源文件（模块）中使用



C++标识符的作用域

- ▶ C++把标识符的作用域分成若干类，包括：
 - ▶ 局部作用域
 - ▶ 全局作用域
 - ▶ 文件作用域
 - ▶ 函数作用域
 - ▶ 函数原型作用域
 - ▶ 类作用域
 - ▶ 名空间作用域



名空间作用域

- ▶ 对于一个由多个文件构成的程序，有时会面临一个问题：
在一个源文件中要用到两个分别在另外两个源文件中定义的不同全局程序实体（如：全局函数），而这两个全局程序实体的名字相同
 - ▶ C++提供了名空间（**namespace**）设施来解决上述的名冲突问题
 - ▶ 在一个名空间中定义的全局标识符，其作用域为该名空间
 - ▶ 当在一个名空间外部需要使用该名空间中定义的全局标识符时，可用该名空间的名字来修饰或受限
-



```
// 模块1
namespace A
{ int x=1;
  void f()
  { .....
  }
}
```

```
// 模块2
namespace B
{ int x=0;
  void f()
  { .....
  }
}
```

```
// 模块3
```

1、

```
... A::x ... //A中的x
A::f(); //A中的f
... B::x ... //B中的x
B::f(); //B中的f
```

2、

```
using namespace A;
... x ... //A中的x
f(); //A中的f
... B::x ... //B中的x
B::f(); //B中的f
```

3、

```
using A::f;
... A::x ... //A中的x
f(); //A中的f
... B::x ... //B中的x
B::f(); //B中的f
```

C++模块的构成

- ▶ 一个C++模块一般包含两个部分：
 - ▶ 接口（.h文件）：
 - ▶ 给出在本模块中定义的、提供给其它模块使用的一些程序实体（如：函数、全局变量等）的声明；
 - ▶ 实现（.cpp文件）：
 - ▶ 模块的实现给出了模块中的程序实体的定义。



- 如果一个函数在其函数体中直接或间接地调用了自己，则该函数称为**递归函数**。
-

- 直接递归

```
void f()
{ .....
  ... f() ...
  .....
}
```

- 间接递归

```
extern void g();
void f()
{ .....
  ... g() ...
  .....
}
void g()
{ .....
  ... f() ...
  .....
}
```

递归条件和结束条件

■ 在定义递归函数时，一定要对两种情况给出描述：

- ▶ **递归条件**：指出何时进行递归调用，它描述了问题求解的一般情况，包括：分解和综合过程。
- ▶ **结束条件**：指出何时不需递归调用，它描述了问题求解的特殊情况或基本情况。

```
int f(int n)
{
    if (n == 0)
        return 1;
    else
        return n*f(n-1);
}
```



例：小牛问题：若一头小母牛，从出生起第四个年头开始每年生一头母牛（每头新生小牛也符合此规律），按此规律，第n年有多少头母牛？

year	未成熟 母牛头数	成熟 母牛头数	母牛 总头数
1	1	0	1
2	1	0	1
3	1	0	1
4	+1	1	2
5	+1+1	1	3
6	+1+1+1	1	4
7	+1+1+1+1	1+1	6
8	+1+1+1+1+1+1	1+1+1	9

$$F(n) = F(n-1) + F(n-3).$$

类似问题的规律：

$$F(n) = F(n-1) + F(n-(m-1)) \quad (n > m)$$

有时候动手“演算”可能有效，能找到规律！

递归与循环的选择

- 对于一些递归定义的问题，用递归函数来解决会显得比较自然和简洁，而用循环来解决这样的问题，有时会很复杂，不易设计和理解。
- 在实现数据的操作上，它们有一点不同：
 - ▶ 循环是在**同一组变量**上进行重复操作（循环常常又称为**迭代**）
 - ▶ 递归则是在**不同的变量组**（属于递归函数的不同实例）上进行重复操作。
- 递归的缺陷：
 - ▶ 由于递归表达的重复操作是通过函数调用来实现的，而函数调用是需要开销的；
 - ▶ 栈空间的大小也会限制递归的深度。
 - ▶ 递归算法有时会出现重复计算。

有时候通过递归实现的函数可以通过循环实现!

宏定义与内联函数

- ▶ 函数调用需要一定的开销，特别是对一些小函数的频繁调用将可能极大地降低程序的执行效率
 - 比如求两个值 x , y 的最小或最大值
- ▶ C++ 提供了两种办法解决上述问题：
 - ▶ 宏定义
 - ▶ 内联函数



宏定义

- ▶ 在C++中，利用一种编译预处理命令：**宏定义**，用它可以实现类似函数的功能：
 - ▶ `#define <宏名>(<参数表>) <文字串>`
 - 例如：
 - ▶ `#define max(a,b) (((a)>(b))?(a):(b))`
- ▶ 在编译之前，将对宏的使用进行**文字替换**
 - 例如：编译前将把
 - ▶ `cout << max(x,y);`
 - 替换成：
 - ▶ `cout << (((x)>(y))?(x):(y));`

一些需要特别注意的“特点”

■ 需要加上一定的括号，例如：

- ▶ `#define max(a,b) a>b?a:b`
- ▶ `10+max(x,y)+z` 将被替换成：
 - `10+x>y?x:y+z`

■ 有时会出现重复计算，例如：

- ▶ `#define max(a,b) (((a)>(b))? (a):(b))`
- ▶ `max(x+1,y*2)` 将被替换成：
 - `((x+1)>(y*2))?(x+1):(y*2)`

■ 不进行参数类型检查和转换

■ 不利于一些工具对程序的处理



内联函数

- 内联函数是指在定义函数定义时，在函数返回类型之前加上一个关键词**inline**，例如：

```
inline int max(int a, int b)
{
    return a>b?a:b;
}
```

- 内联函数的作用是**建议**编译程序把该函数的函数体展开到调用点，以提高函数调用的效率。
- 内联函数形式上属于函数，它遵循函数的一些规定，如：参数类型检查与转换。
- 使用内联函数时应注意以下几点：
 - ▶ 编译程序对内联函数的限制
 - ▶ 内联函数名具有文件作用域

函数名重载

- ▶ 对于一些功能相同、参数类型或个数不同的函数，有时给它们取相同的名字会带来使用上的方便。例如，把下面的函数：

```
void print_int(int i) { ..... }  
void print_double(double d) { ..... }  
void print_char(char c) { ..... }  
void print_A(A a) { ..... } //A为自定义类型
```

定义为：

```
void print(int i) { ..... }  
void print(double d) { ..... }  
void print(char c) { ..... }  
void print(A a) { ..... }
```



C++标准库函数

■ 为了方便程序设计，C++语言提供了标准库，其中定义了一些语言本身没有提供的功能：

- ▶ 常用的数学函数
- ▶ 字符串处理函数
- ▶ 输入/输出
- ▶ ...



条件编译

- 编译程序根据不同的情况来选择需编译的程序代码。例如，编译程序将根据宏名ABC是否被定义，来选择需要编译的代码：

<代码1> //必须编译的代码

#ifdef ABC

<代码2> //如果宏名ABC有定义，编译之

#else

<代码3> //如果宏名ABC无定义，编译之

#endif

<代码4> //必须编译的代码

- 用于条件编译的宏名ABC在哪里定义？

条件编译的作用

■ 条件编译的作用：

- ▶ 基于多环境的程序编制
- ▶ 程序调试
 - ▶ assert 宏
- ▶



assert宏

- 利用标准库中定义的宏：**assert**

```
#include <cassert> //或<assert.h>
```

```
.....
```

```
assert(x == 1); //断言
```

- assert的定义大致如下：

```
.....
```

```
#ifdef NDEBUG
```

```
#define assert(exp) ((void)0)
```

```
#else
```

```
#define assert(exp) ((exp)?(void)0:<输出诊断信息并调用库函数abort>)
```

```
#endif
```

```
.....
```



举例：assert典型应用

在函数开始处检验传入参数的合法性（非常重要！用来及时检查传递进来的实参是否是合法的，不合法趁早终止程序执行）

如：

```
int resetBufferSize(int nNewSize)
```

```
{
```

```
    //功能:改变缓冲区大小,
```

```
    //参数:nNewSize 缓冲区新长度
```

```
    //返回值:缓冲区当前长度
```

```
    //说明:保持原信息内容不变    nNewSize<=0表示清除缓冲区
```

```
    assert(nNewSize >= 0);
```

```
    assert(nNewSize <= MAX_BUFFER_SIZE);
```

```
    ...
```

```
}
```

VS常用（键盘）快捷键

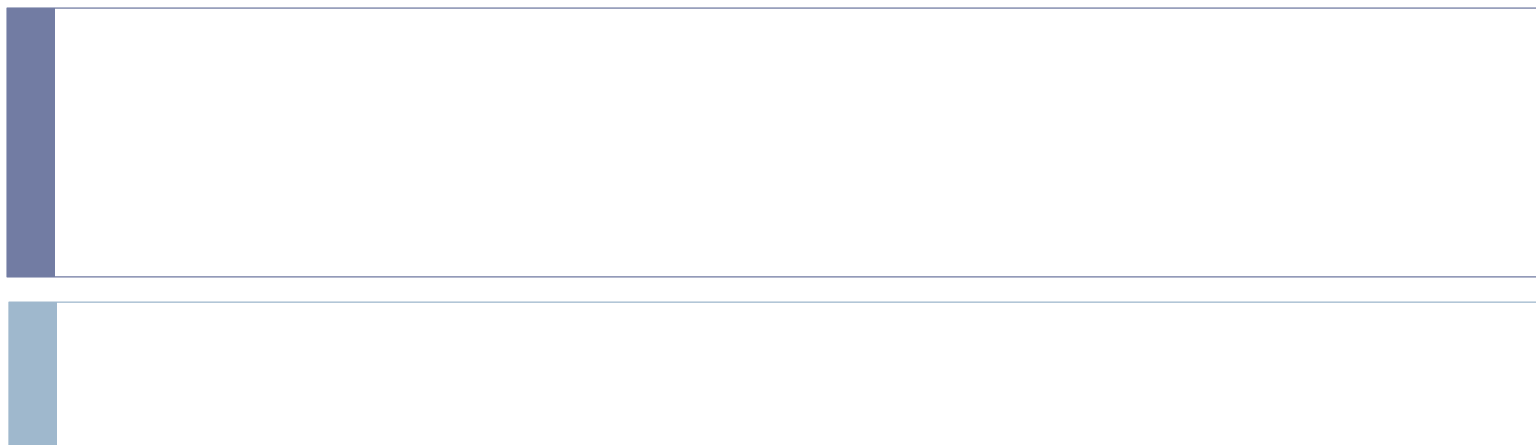
（掌握常用快捷键！）

进入调试后：

- ▶ F5：继续执行程序直到遇到下一个断点。
- ▶ Shift+F5：停止调试。
- ▶ Ctrl+Shift+F5：重新开始调试。
- ▶ F10：Step over，执行下一行语句，若有函数调用不会进入函数体。
- ▶ F11：Step into，执行下一行语句，若有函数调用会进入函数体。
- ▶ Shift+F11：Step out，跳出当前函数。
- ▶ F9：可以在调试状态下新增断点

第5章

复杂数据的描述— 构造数据类型



一维数组

构造类型的同时定义变量

```
int a[6];    // int、[]和6构造了一个一维数组类型  
            // 并用该类型定义了一个一维数组a
```

- a 同时表示a[0]的地址
- 下标元素从0开始 a[0]、a[1]、…、a[5]
- 对元素遍历操作，单循环！

将一维数组作为函数形参

```
#include <stdio.h>
int Max(int x[ ], int num);
int main( )
{
    int a[10] = {12,1,34}, index_max = 0;
    index_max = Max(a, 10);
    printf("数组a的最大元素是: %d\n", a[index_max]);

    return 0;
}
```

```
int Max(int x[ ], int num)
{
    int j = 0;
    for(int i = 1; i < num; i++)
        if(x[i] > x[j]) j = i;
    return j;
}
```

二维数组

```
int b[3][2] = {0, 1, 2, 3, 4, 5};
```

- b 同时表示 $b[0]$ 在内存的起始位置
- 下标从0开始, $b[i][j]$ 是第 $i+1$ 行的第 $j+1$ 个元素
- 可以看作含 $b[0]$, $b[1]$ 和 $b[2]$ 三个一维数组元素
- 对元素遍历操作, 双循环!



例：求矩阵元素的和

```
#include <stdio.h>
```

```
#define M 10
```

```
#define N 5
```

```
int main()
```

```
{
```

双循环！

```
    int sum = 0, mtrx[M][N];
```

```
    for(int i = 0; i < M; i++)    // 0 开始, M-1 结束
```

```
        for(int j = 0; j < N; j++)    // 0 开始, N-1 结束
```

```
        {
```

```
            printf("Input a number: ");
```

```
            scanf("%d", &mtrx[i][j]);
```

$mtrx[0][N * i + j]$

```
            sum += mtrx[i][j];
```

```
        }
```

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

```
    return 0;
```

```
}  
▶
```

将二维数组作为函数形参

例：用函数实现求矩阵元素求和

```
#include <stdio.h>
#define N 5
int Sum(int x[ ][N], int lin);
int main( )
{
    int mtrx[10][N] = {·····};
    printf("sum = %d\n", Sum(mtrx, 10));
    return 0;
}
```

```
int Sum(int x[ ][N], int lin)
{
    int s = 0;
    for(int i = 0; i < lin; i++)
        for(int j = 0; j < N; j++)
            s += x[i][j];
    return s;
}
```

采用一维数组实现

例：用函数实现求矩阵元素求和

```
int Sum(int x[ ], int num)
{
    int s = 0;
    for(int i = 0; i < num; i++)
        s += x[i];
    return s;
}

.....
int m1[10][5];

.....
printf("sum = %d\n", Sum(m1[0], 10 * 5));
```



数组及其应用

- ▶ 一维数组
 - ▶ 可用来表示向量...
- ▶ 二维数组
 - ▶ 可用来表示矩阵...
- ▶ 多维数组
- ▶ 数组的应用
- ▶ 基于数组的排序程序

数组的应用

▶ 实际应用中：

- ▶ 表示有序的（相同属性）一组数据

 - ▶ 向量、数列、同学们的成绩…

- ▶ 还可以利用数组存储一组有序**标志位**，以对应一组（有相同属性）数据的状态

例：求矩阵的乘积

▶ 设有两个矩阵 A_{mn} 、 B_{np} ,

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{pmatrix}$$

乘积矩阵 $C_{mp} = A_{mn} \times B_{np}$, C_{mp} 中每一项的计算公式为:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \times B_{kj}$$

$$i=0, \dots, m-1, j=0, \dots, p-1$$

$$\begin{pmatrix} 4+5+6+7 & 4+5+6+7 \\ 8+10+12+14 & 8+10+12+14 \\ 12+15+18+21 & 12+15+18+21 \end{pmatrix}$$

```

#include <stdio.h>
#define N 4
#define M 3
-----
#define P 2
void prod(int ma[ ][N], int mb[ ][P], int mc[ ][P], int m);
int main( )
{
    int ma[M][N] = {{1, 1, 1, 1}, {2, 2, 2, 2}, {3, 3, 3, 3}};
    int mb[N][P] = {{4, 4}, {5, 5}, {6, 6}, {7, 7}};
    int mc[M][P];
    prod(ma, mb, mc, M);
    for(int i = 0; i < M; i++)
        for(int j = 0; j < P; j++)
        {
            printf(" %d ", mc[i][j]);
            if((j + 1) % P == 0) printf("\n");
        }
    return 0;
}

```

22	22
44	44
66	66



```
void prod(int ma[ ][N], int mb[ ][P], int mc[ ][P], int m)
{
    for(int i = 0; i < m; i++)
        for(int j = 0; j < P; j++)
        {
            int s = 0;
            for(int k = 0; k < N; k++)
                s += ma[i][k] * mb[k][j];
            mc[i][j] = s;
        }
}
```

基于数组的排序

- ▶ 排序是一种常见、非常重要的问题
- ▶ 解决排序问题的算法有很多：
 - ▶ 起泡排序
 - ▶ 选择排序
 - ▶ 插入排序
 - ▶ 快速排序等
 - ▶ 以及它们的变种

至少熟练掌握一种排序方法！
了解其他排序方法的思路

基于数组的排序

- ▶ 如果待排序数据存在数组中，则用程序实现这些排序算法时涉及到
 - ▶ 数组的遍历
 - ▶ 两个元素的比较、交换
 - ▶ 一个元素的插入等操作
 - ▶ ...
- ▶ 需要综合运用循环、分支流程控制及赋值操作等完成

“冒泡” 改写为函数的形式

```
#define N 4
```

```
int main()
```

```
{
```

```
    int a[N];
```

```
    for (int i=0; i<N; i++)
```

```
        scanf("%d", &a[i]);
```

```
    BubbleSort(a, N);
```

```
    ...
```

```
}
```

```
void BubbleSort(int sdata[ ], int count)
```

```
{
```

```
    int temp;
```

```
    for(int i = 0; i < count-1; i++)
```

```
        for(int j = 0; j < count-1-i; j++)
```

```
            if(sdata[j] > sdata[j+1])
```

```
            {
```

```
                temp = sdata[j];
```

```
                sdata[j] = sdata[j+1];
```

```
                sdata[j+1] = temp;
```

```
            }
```

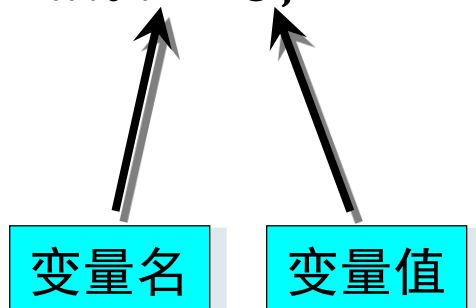
```
}
```

选择排序

```
void SelSort(int sdata[ ], int count)
{
    for(int i = count; i > 1; i--)           // 大循环 count-1次
    {
        int max = 0;
        for(int j = 1; j < i; j++)           // 小循环: “选择” (找到) 当
                                                // 前大循环最大的
            if(sdata[max] < sdata[j])
                max = j;
        if(max != i-1)                       // 交换
        {
            int temp = sdata[max];
            sdata[max] = sdata[i-1];
            sdata[i-1] = temp;
        }
    }
}
```

什么是指针(pointer)

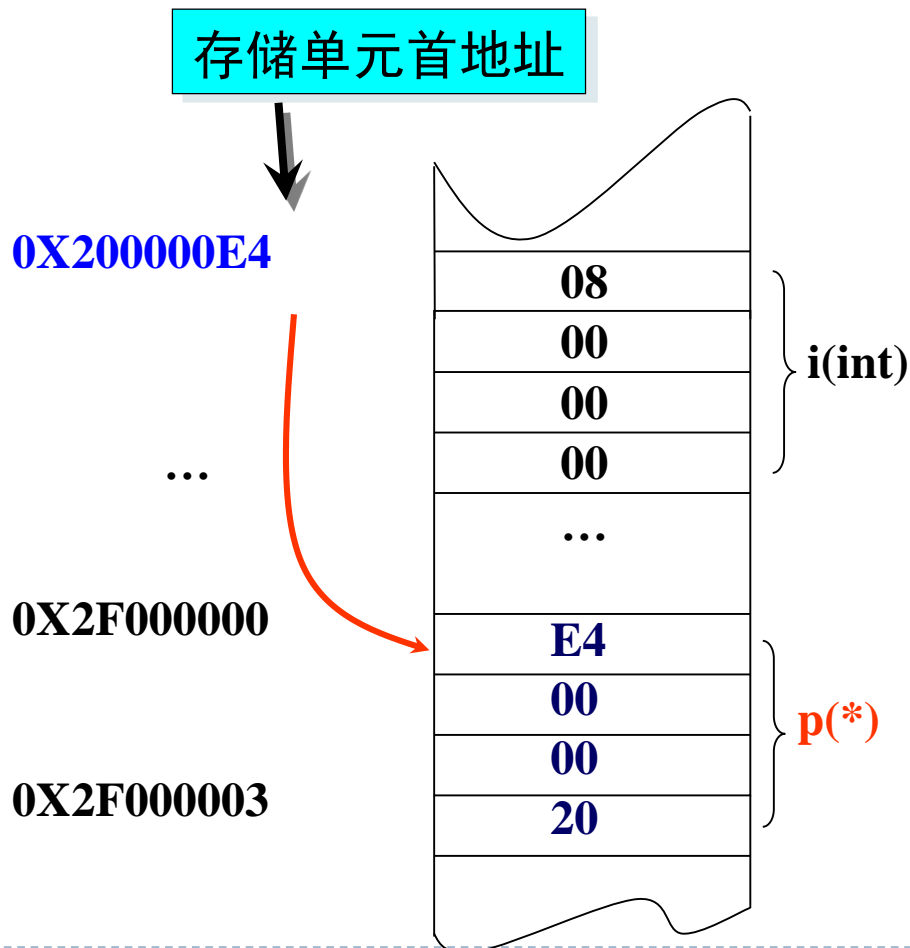
```
int i = 8;
```



```
int *p;
```

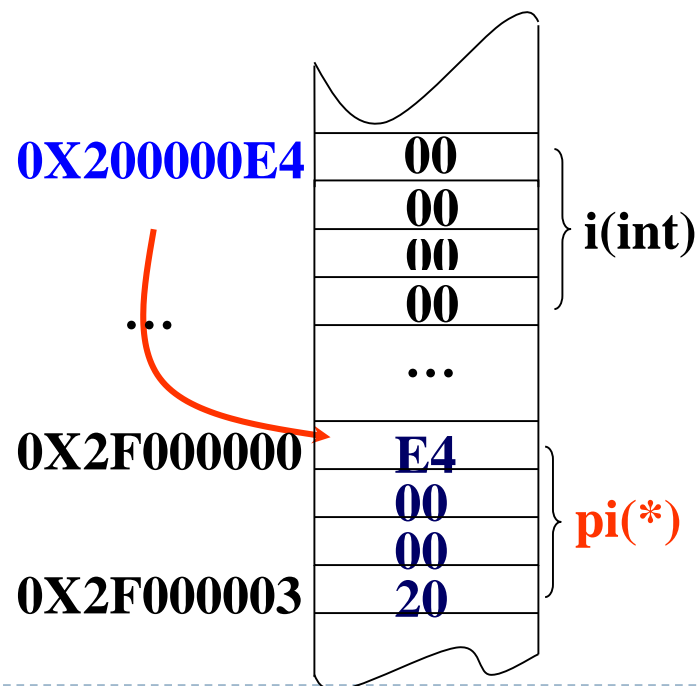
```
p = &i;
```

```
// p就是一个指针，指向i
```



举 例

```
int *pi=NULL;    //这里的 * 不是指针操作符
int i = 0;
pi = &i;         //pi指向i
*pi = 1;         //对pi进行取值操作并对其赋值，相当于i = 1;
*(&i) = 2;       //相当于i = 2;
pi = &(*pi);     //pi仍然指向i
```



指针数组

- ▶ 如果数组的每一个元素都是一个指针类型的数据，
则该数组叫做指针数组。

▶ 比如，

```
int i = 0, j = 1, k = 2;
```

```
int *ap[3] = {&i, &j, &k};
```

//ap这个指针数组的长度是3，各个元素的类型是int *

- ▶ 指针数组一般用于多个字符串的处理

多级指针变量

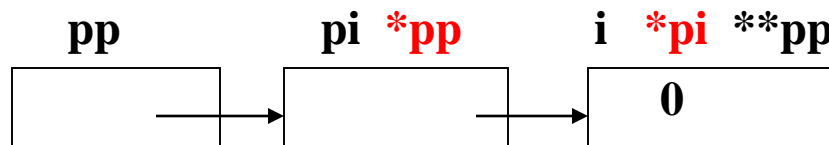
- ▶ C语言的指针变量还可以存储另一个指针变量的地址。

- ▶ 比如,

```
int i = 0;
```

```
int *pi = &i;
```

```
int **pp = &pi; // 指针变量pp存储的是指针变量pi的地址
```



- ▶ 这时，称指针变量pp是一个二级指针变量（指向指针的指针），它指向的变量是一个一级指针变量pi。
- ▶ 如果再定义一个指针变量ppp存储pp的地址，则ppp是一个三级指针变量。
- ▶ 多级指针变量通常用于指针类型数据的传址调用，或者用于多维数组和多个字符串的处理。

例：用指针变量操纵数组

```
#include <stdio.h>
```

```
#define N 10
```

```
int main( )
```

```
{ int i;
```

```
int a[N] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
int *pa;
```

```
for(pa = a, i = 0; i < N; i++)
```

```
    printf("%d ", pa[i]);
```

//下标法，用a[i]也行

```
for(; pa < (a + N); pa++)
```

//指针移动法，用a++不行，因为a是常量

```
    printf("%d ", *pa);
```

```
for(pa = a, i = 0; i < N; i++)
```

//这里没有“pa = a, i = 0”，则会出现越界！

```
    printf("%d ", *(pa + i));
```

//偏移量法，用*(a + i)也行

```
return 0;
```

```
}
```

用指针操纵数组-2

- ▶ 指向**整个数组**的指针变量，即数组的指针（注意与指针数组的区别）。
- ▶ 一个指针类型的基类型可以是int、float这样的基本类型，也可以是数组这样的派生类型。比如，

```
typedef int A[10];  
A *q;
```
- ▶ 或者合并写成，

```
int (*q)[10];
```
- ▶ 该指针变量q可以存储一个类型为A的数组的地址，比如，

```
int a[10];  
q = &a; //q相当于一个二级指针变量
```

int *p[10] V. S. int (*p)[10]

- ▶ int *p[10] 中p是一个数组。可以理解为

int* p[10], 先定义一个一维数组, 再看括号外, 数组中每个变量都是int型指针。

- ▶ int (*p)[10] 中p是一个指针。它的类型是: 指向

int [10]这样的一维数组的指针。



用指针操纵二维数组-1

- ▶ 对于二维数组，可以通过不同级别的指针变量来操纵。
注意：二维数组名表示第一行的地址。

- ▶ 比如，

```
int b[5][10];
```

```
int *p;
```

```
p = &b[0][0]; //或 “p = b[0];”
```

```
// 一级指针变量可以存储二维数组b某一元素的地址
```

- ▶ 然后通过指针移动法指定某个元素地址，再取值指定任一元素，比如 `p++`，`*p`;

用指针操纵二维数组-2

▶ 或,

```
int (*q)[10];
```

```
q = &b[0];          // 或 “q = b;”
```

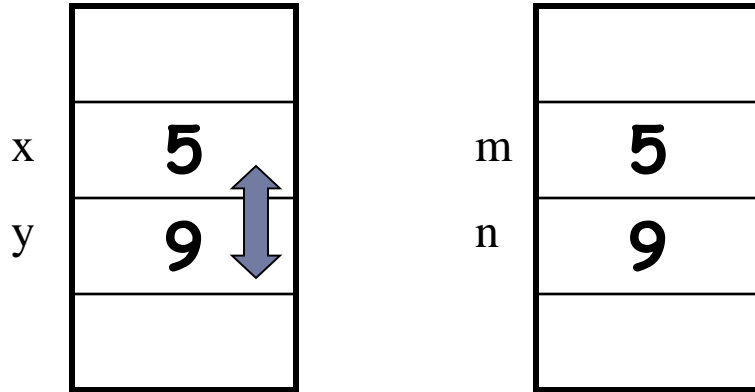
// 二级指针变量可以存储 **二维数组b某一行的地址**

- ▶ 然后通过**下标法**指定元素, 比如`q[i][j]` (相当于`b[i][j]`);
- ▶ 也可以通过**指针移动法**指定某一行的地址, **再取值**指定某一行的首元素, 比如`q++`, `**q`;
- ▶ 还可以通过**偏移量法**指定任一元素, 比如
 - ▶ `*(*(q+i)+j)` (相当于`*(*(b+i)+j)`)
 - ▶ `*(&q[0][0]+10*i+j)` (相当于`*(&b[0][0]+10*i+j)`)
 - ▶ `*(q[i]+j)` (相当于`*(b[i]+j)`)
 - ▶ `*(q+i)[j]` (相当于`*(b+i)[j]`)

▶ **用指针代表的地址来理解!!!**

	形参	实参	特点	举例
传 值 调 用	变量 ←...	常量 变量 值 表达式 值	形参的 改变 不影响 实参	<pre>void Swap1(int x, int y) { int temp = x; x = y; y = temp; } int main() { int m = 5, n = 9; Swap1(m, n); printf("%d %d", m, n); return 0; }</pre> <div>int x=m int y=n</div>
传 址 调 用	指针 ←	地址 值	改变形参 所指向的 变量值来 影响实参	<pre>void Swap (int *pm, int *pn) { int temp = *pm; *pm = *pn; *pn = temp; } int main() { int m = 5, n = 9; Swap(&m, &n); printf("%d %d", m, n); return 0; }</pre> <div>int *pm=&m int *pn=&n</div>

Swap1

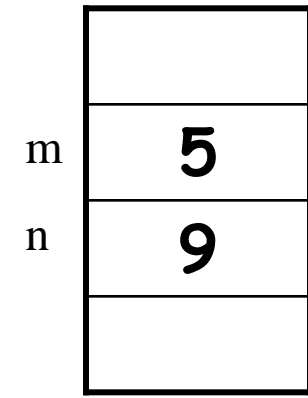
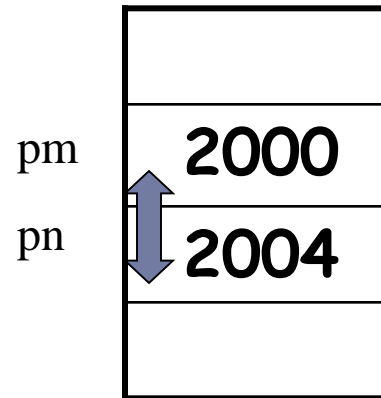
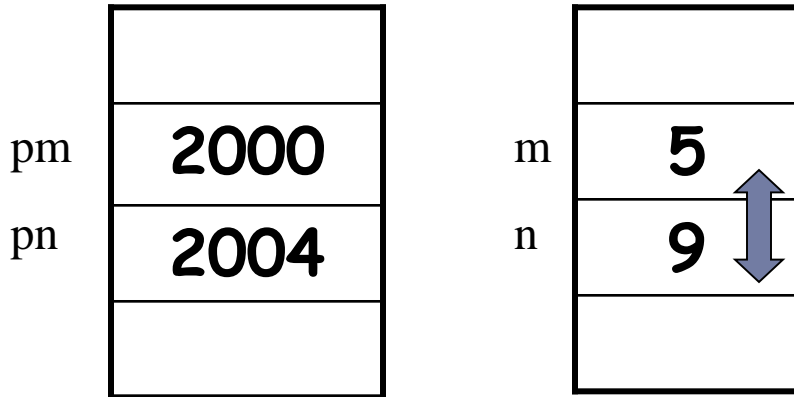


```
void Swap2(int *pm, int *pn)
```

```
{
    int *temp = pm;
    pm = pn;
    pn = temp;
}
```

```
int main( )
```

```
{
    int m = 5, n = 9;
    Swap2(&m, &n);
    printf("%d %d", m, n);
    return 0;
}
```



Swap

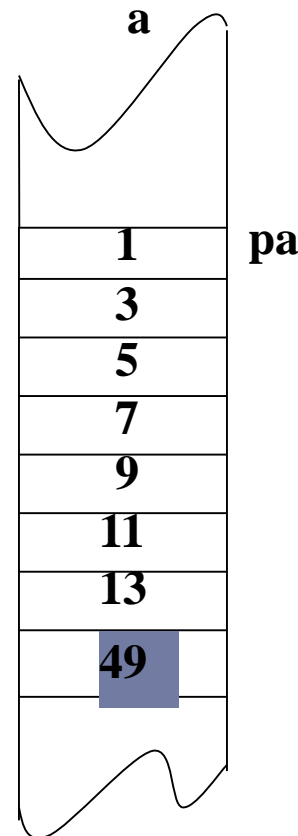
Swap2

例：指针变量作为函数形参的双重作用

```
#include <stdio.h>
#define N 8
void Fun(int * pa, int n);
int main( )
{   int a[N] = {1, 3, 5, 7, 9, 11, 13}; // 注意a[N-1]初始化为0
    Fun(a, N);    // int *pa = a
    printf("%d \n", a[N-1]);
    return 0;
}

void Fun(int * pa, int n)
{   for(int i = 0; i < n-1; i++)
        *(pa + n - 1) += *(pa + i);
}
```

int *pa=a



//既利用传址调用提高了数据正向“传递”的效率

//又利用函数的副作用实现了数据的反向“传递”(修改了数组最后一个元素的值)

const的不同位置含义不同

▶ `const double pi = 3.1415926;`

// 定义const浮点类型的数据

▶ `void F(const int *p, int num)`

// p指向的整形不能被改变值

函数间有多种通讯方式

- ▶ 传值方式(把实参的复制给形参)
- ▶ 利用函数返回值传递数据
- ▶ 通过全局变量传递数据
(根据“切实”需要定义全局变量)
- ▶ 通过指针/引用类型参数传递数据
(函数副作用问题，可以通过指向常量的指针来避免)

指针型函数

- ▶ C语言的return语句只能返回一个值（不能返回数组类型的数据），但可以返回数组或数组元素的地址。当函数的返回值是一个地址时，称该函数为指针型函数

```
int *Max(const int ac[ ], int num)
{
    int max_index = 0;
    for(int i = 1; i < num; i++)
        if(ac[i] > ac[max_index])
            max_index = i;
    return (int *)&ac[max_index];
}
```

将const int 型地址转换成int 型地址

动态变量的定义 (C++形式)

- ▶ 指在程序运行中，由程序根据需要所创建的变量。例如：

- ▶ `int *p1;`

- ▶ `p1 = new int;` //C++; 创建了一个int型动态变量，p1指向之。

- ▶ 再例如：

- ▶ `int *p2;`

- ▶ `p2 = new int[n];`

// 创建了一个由n (n可以是变量) 个int型元素所构成的动

// 数组变量，“相当”于 `int a[n]`



建议后面上机用C++风格的动态变量！

```
int n, *q=NULL; //初始化一般放构造函数中
```

```
n= ...
```

```
q = new int[n]; //创建
```

```
....
```

```
q[i] // i=0,...,n-1
```

```
...
```

```
delete []q; //撤销
```

```
q=NULL; //一般放析构函数中
```



例：动态变量的应用——动态数组

- ▶ 对输入的若干个数进行排序，如果输入时先输入数的个数，然后再输入各个数，则可用下面的动态数组来表示这些数：

```
int n;  
int *p=NULL;  
cin >> n;  
p = new int[n];  
for (int i=0; i<n; i++)  
    cin >> p[i];  
sort(p,n);  
.....  
delete []p;  
p = NULL;
```



例：对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）：

```
const int INCREMENT=10;
int max_len=20,count=0,n,*p=new int[max_len];
cin >> n;
while (n != -1)
{
    if (count >= max_len)
    {
        max_len += INCREMENT;
        int *q=new int[max_len];
        for (int i=0; i<count; i++)
            q[i] = p[i];
        delete []p;
        p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p,count);
.....
delete []p;
p=NULL;
```

用指针操纵函数**

- ▶ C程序运行期间，程序中每个函数的目标代码也占据一定的内存空间。C语言允许将该内存空间的首地址赋给函数指针类型的变量（简称函数指针，注意与指针型函数的区别），然后**通过函数指针来调用函数**。

- ▶ 比如，

```
typedef int (*PFUNC)(int); // 构造了一个函数指针类型  
PFUNC pf;                // 定义了一个函数指针
```

- ▶ 也可以在构造函数指针类型的同时直接定义函数指针：

```
int (*pf)(int);  
// 第一个int为函数的返回值类型，第二个int为参数的类型。
```


例：根据要求(输入的值除8的余数)，执行在函数表中定义的某个函数（8个函数中的一个）

```
#include <stdio.h>
```

```
#include <cmath>
```

```
typedef double (*PF)(double); // 构造函数指针类型
```

```
PF func_list[8] = {sin, cos, tan, asin, acos, atan, log, log10}; // 定义长度为8的函数指针的数组
```

```
int main( )
```

```
{  int index;
```

```
    double x;
```

```
    do
```

```
    {    printf("请输入要计算的函数(0:sin 1:cos 2:tan 3:asin 4:acos 5:atan 6:log 7:log10):\n");
```

```
        scanf("%d", &index);
```

```
    } while(index < 0 );
```

```
    printf("请输入参数: ");
```

```
    scanf("%lf", &x);
```

```
    printf("结果为: %d \n", (*func_list[index%8])(x));
```

```
    return 0;
```

```
}
```

指针总结

▶ 指针及其应用-1

▶ 指针的基本概念

▶ 指针类型的构造

▶ 指针变量的定义与初始化

```
int *  
float *  
double *  
char *
```

```
int *pi = &i;  
float *pf = &f;  
double *px = &x;  
char *pch = &ch;
```

```
int *ap[3] = {&i, &j, &k};
```

▶ 指针数组

```
int ** pp = &pi;
```

▶ 多级指针变量

```
void *
```

```
void *pv = 0;
```

▶ 通用指针与void类型

▶ 指针类型相关的基本操作

* (与& “互逆”)

=

>

<

>=

<=

==

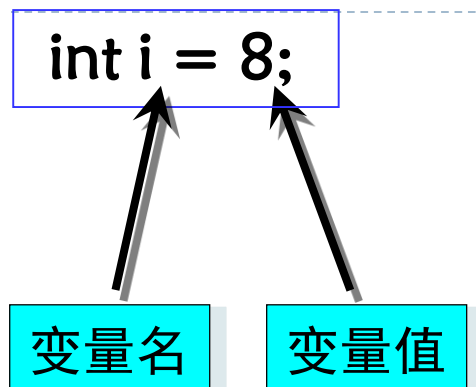
!=

++

--

-

“举个形象的例子”



```
int i, *p;
```

```
p = &i;
```

```
i = 南京大学;
```

```
// p 的值: 南京市栖霞区仙林大道163号
```

```
// *p 的值: 南京大学
```

```
*p = 国立中央大学; // i 也变为了国立中央大学
```

```
p++; // 在 p 的值基础上加其指向的基类型的字节数
```

指针总结

▶ 指针及其应用-2

▶ 用指针操纵数组

▶ 用指针在函数间传递数据

▶ 传址调用

▶ const

▶ 指针型函数

▶ 用指针访问动态变量

▶ 动态变量的创建、访问与撤销

▶ 内存泄露与悬浮指针**

▶ 用指针操纵函数**

```
int a[10];  
int b[5][10];  
int * p = a;  
int (*q)[10] = &a;  
int (*r)[5][10] = &b;
```

```
int * ap[3] = {&i, &j, &k};  
// 注意ap与q的区别
```

```
Fun(a, ...);  
void Fun(int *pa)  
{ *pa...pa++...}
```

```
Fun(&n);  
void Fun(int *pn)  
{ *pn = ...}
```

```
void Fun(const int *pa)  
{ *pa...pa++...} ×  
*pa = ...
```

```
int * Fun(...)  
{...return &...}
```

引用类型

- ▶ 引用类型用于给一个变量取一个别名。例如：

```
int x=0;  
int &y=x;      //y为引用类型的变量，可以看成是x的别名  
cout << x << ', ' << y << endl; //结果为：0, 0  
y = 1;  
cout << x << ', ' << y << endl; //结果为：1, 1
```

- ▶ 在语法上，

- ▶ 对引用类型变量的访问与非引用类型相同

- ▶ 在语义上，

- ▶ 对引用类型变量的访问实际访问的是另一个变量（被引用的变量）
- ▶ 效果与通过指针间接访问另一个变量相同

引用类型作为函数的参数类型

- 提高参数传递的效率。例如：

```
struct A
{
    int i;
    .....
};
void f(A &x) //x引用相应的实参
{
    .....
    ... x.i ... //访问实参
    .....
}
int main()
{
    A a;
    .....
    f(a); //引用传递，提高参数传递效率
    .....
}
```



常量的引用

- ▶ 通过把形参定义成对常量的引用，可以防止在函数中通过引用类型的形参改变实参的值。

```
struct A
{   int i;
    .....
};
void f(const A &x)
{   x.i = 1; //Error
    .....
}
int main()
{   A a;
    .....
    f(a);
    .....
}
```



字符串

- ▶ 字符数组
 - ▶ 字符数组的定义和初始化
 - ▶ 字符数组的输入/输出
 - ▶ 字符数组作为函数的参数
 - ▶ 用字符指针操纵字符数组
- ▶ 字符串常量的访问
- ▶ 常用字符串处理库函数
- ▶ 基于字符的信息检索程序
- ▶ 字符型指针数组与带形参的main函数**

总结1: C字符串三种输入/赋值方法

1. 声明字符串时直接赋值
2. 用scanf函数 (%s格式符)
3. strcpy函数给字符串赋值, 比较常用

char a[10] = "123"; 或

#include <string.h>

... ..

char a[10];

strcpy(a, "123");



总结2: C++字符串输入方法

➤ 使用cin输入字符串的相关问题

➤ cin 使用空白（空格、制表符和换行符）来定字符串的界

这意味着cin在获取字符数组输入时只读取一个单词（遇到空格即止），在读取该单词后，cin将该字符串放到数组中，并自动在结尾添加空字符

➤ 用gets()库函数



字符串常量的初始化：采用指针（第二种方法）

▶ 第一种方法

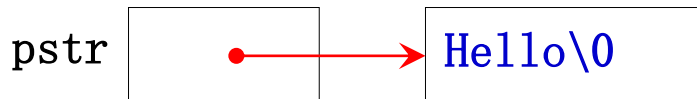
▶ `char astr[] = "Hello";` `astr` Hello\0

//astr在栈区占6个字节空间，存储该字符串常量的副本

▶ 第二种方法

▶ `char *pstr = "Hello";` Hello\0

或者



▶ `char *pstr;`
`pstr = "Hello";`

//pstr在栈区占4个字节空间，存储该字符串常量的首地址

例：从键盘输入一个字符串，然后把该字符串逆向输出（反转）

```
#include <iostream>
```

```
#include <cstring>
```

```
using namespace std;
```

```
int main()
```

```
{ const int MAX_LEN=100;
```

```
char str[MAX_LEN];
```

```
gets(str);
```

```
int len = strlen(str); // str中的字符个数（\0之前）
```

```
for (int i=0,j=len-1; i<len/2; i++,j--) // 两个循环变量，也可以一个
```

```
{
```

```
    char temp;
```

```
    temp = str[i];
```

```
    str[i] = str[j];
```

```
    str[j] = temp;
```

```
}
```

```
cout << str << endl;
```

```
return 0;
```

比如：“Program” 转换为 “margorP”。

对于for循环等，
需要注意循环变量的边界条件
(还记得排序的双重循环，
循环变量的循环条件?)

```
}
```

例：统计输入的字符行中各个数字、空格及其他字符的个数

```
int k, nSpace=0, nOther=0;
char ch;
int nDigit[10];           // 10个数字符号的计数器，类似于“标志位”
for(k = 0; k < 10; k++) nDigit[k] = 0;
printf("Enter string line\n");

while ((ch= getchar()) != '\n')
{
    if (ch >= '0' && ch <= '9')
        ++nDigit[ch - '0'];    // 对应计数器增1
    else if (ch == ' ')
        ++nSpace;
    else
        ++nOther;
}
for(k = 0; k < 10; k++)
    printf( "%d: %d \n", k, nDigit[k]);
printf("space:%d; other:%d \n", nSpace, nOther);
```

字符数组作为函数的参数

- ▶ 当一维数组作为函数的参数时，通常需要把一维数组的名称以及数组元素的个数传给被调用函数，以便被调函数确定数组处理的终点。
- ▶ 对于一维字符数组，则不需要传递数组元素的个数，因为可以凭借 ‘\0’ 来确定其处理终点。

例：字符串的大小写转换（小写转大写）

```
#include <stdio.h>
void to_upper(char s[]);
int main()
{   char str[10];
    printf("Please input a string: \n");
    gets(str);
    to_upper(str);
    printf("The new string is %s. \n", str);
}

void to_upper(char s[])
{   int i;
    for(i = 0; s[i] != '\0'; i++)
    {
        if(s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i] - 'a' + 'A'; //小写转换成大写 OR: -32
    }
}
```

例：数字字符串到整数的转换

▶ 比如 “365” 转换为 $((3*10)+6)*10+5$ 。

$$0*10 + 3 = 3$$

$$3*10 + 6 = 36$$

$$36*10 + 5 = 365$$

```
int str_to_int(char s[ ])
```

```
{
```

```
    if(s[0] == '\0') return 0;
```

//空字符串转换为0

```
    int i, n = 0;
```

//n用于存储转换结果，初始化为0

```
    for(i = 0; s[i] != '\0'; i++)
```

//循环处理各位数字

```
        n = n * 10 + (s[i] - '0');
```

```
    return n;
```

```
}
```

//通过数字字符与字符'0'的ASCII码差值计算数字字符对应的数值

//即 ‘3’ - ‘0’ == 3, ‘6’ - ‘0’ == 6, ‘5’ - ‘0’ == 5

C标准库中的字符串处理函数

(头文件 `cstring` 或 `string.h`)

▶ 计算字符串的长度

- ▶ `int strlen(const char s[]);`

▶ 字符串复制

- ▶ `char *strcpy(char dst[],const char src[]);`

- ▶ `char *strncpy(char dst[],const char src[],int n);`

▶ 字符串拼接

- ▶ `char *strcat(char dst[],const char src[]);`

- ▶ `char *strncat(char dst[],const char src[],int n);`

▶ 字符串比较

- ▶ `int strcmp(const char s1[],const char s2[]);`

- ▶ `int strncmp(const char s1[],const char s2[],int n);`

▶ 模式匹配

- ▶ `char *strstr(char *haystack, char *needle);`

▶ ▶ ...

字符型指针数组**

- ▶ 对于每一个元素都是一个字符型指针的一维数组，可以用来表示多个字符串。比如，

```
char ss[4][5] = {    {'Z', 'h', 'a', 'o', '\0'},  
                  {'Q', 'i', 'a', 'n', '\0'},  
                  {'S', 'u', 'n', '\0'},  
                  { 'L' , 'i' , '\0' } };
```

- ▶ 也可以写成:

```
char ss[4][5] = {"Zhao", "Qian", "Sun", "Li"}; 或
```

```
char *ssp[4] = {"Zhao", "Qian", "Sun", "Li"}; 或
```

```
char a[4][5];
```

```
char *ssp[4] = {a[0], a[1], a[2], a[3]}; 或
```

```
char *ssp[4] = {&a[0][0], &a[1][0], &a[2][0], &a[3][0]};
```

举 例

(字符型指针数组) :

week	0X2002
week+1	0X2012

0X2002	S u n d a y \0
0X2012	M o n d a y \0

```
char *week[7] = {"Sunday", "Monday", "...", "Saturday"};
```

- ▶ 定义了一个含7个元素的一维数组，每个元素都是一个字符型指针，并用7个字符串常量对其进行了初始化，其中：
- ▶ week是一维数组名，表示第一个元素week[0]的地址（假设其值为0x0065fdf0，可用格式符%lx显示）；
- ▶ week+1表示第二个元素week[1]的地址（假设其值为0x0065fdf4（=0x0065fdf0 + 4））；
- ▶ week[0]表示第一个元素，其值为一个字符型地址（假设其值为0x00002002），用%s输出结果为Sunday；
- ▶ week[0] + 1表示第一个元素的值加1（其值为0x00002003），仍然是一个字符型地址，用%s输出结果为 ；
- ▶ week[0][0]表示第一个元素所指向字符串的第一个字符，用%c输出结果为S；
- ▶ week[0][0] + 1表示将第一个元素所指向字符串的第一个字符ASCII码加1，用%c输出结果为 ；

例：多个字符串的排序程序 (将百家姓的拼音按字典顺序重排)

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main( )
```

```
{ char *temp, *name[ ] = {"Zhao", "Qian", "Sun", "Li"};
```

```
    int n=4, i, min;
```

```
    for(i = 0; i < n - 1; i++)
```

```
    {    min = i;
```

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

```
            if(strcmp(name[min], name[j]) > 0)    min = j;
```

```
    if(min != i)
```

```
    {    temp = name[i];
```

```
        name[i] = name[min];
```

```
        name[min] = temp;
```

```
    }
```

```
}
```

用选择排序法

```
for(i = 0; i < n; i++)  
    printf("%s \n", name[i]);  
return 0;  
}
```

二分法查找 写成独立的函数

```
int main()
{  char key,str[ ] = "abcdefghijklmnopqrst";
   printf("please input a letter:");
   scanf("%c", &key);
   int flag = BiSearch(str, key, 0, strlen(str)-1);
   if(flag == -1) printf("\n not found \n");
   else          printf("%d \n", flag );
   return 0;
}
```

```
int BiSearch(char x[ ], char k, int ph, int pt)
{  int pmid;
   while(ph <= pt)
   {    pmid = (ph+pt)/2;
        if (k == x[pmid])          break;
        else if (k > x[pmid])      ph = pmid+1;
        else                      pt = pmid-1;
   }
   if(ph > pt) pmid = -1;
   return pmid;
}
```

写成递归函数

```
int BiSearch(char x[ ], char k, int ph, int pt)
{
    if(ph <= pt)
    {
        int pmid = (ph+pt)/2;
        if(k == x[pmid])                return pmid;
        else if(k > x[pmid])            return BiSearch(x, k, pmid+1, pt);
        else if(k < x[pmid])            return BiSearch(x, k, ph, pmid-1);

    }
    else                                return -1;
}
```

二分法（折半法）查找字符串

```
cp = Bin(temp, "sun", 4, &cn);
```

```
char *Bin(
{
    int left, right, mid;
    left = 0;
    right = n-1;
    while(left <= right)
    {
        ...
    }
    return 0;
}

mid = (left+right)/2;
if (strcmp(str, sp[mid]) < 0)
    right = mid-1;
else if(strcmp(str, sp[mid])>0)
    left = mid+1;
else
{
    *addr = mid;
    return(sp[mid]);
}
```

通过参数传递值只能实参→形参

而通过指针指向实参，则可修改实参

```
char *Bin(char *sp[ ], char *str, int n, int *addr)
```

```
int main( )
{
    int cn;
    char *cp;
    char *temp[] = {"li", "qian", "sun", "zhao"};
    cp = Bin(temp, "sun", 4, &cn);
    if(cp != 0)
        printf("%d\n", cn+1);
    return 0;
}
```


结构/联合及其应用

- ▶ 结构类型基本概念
 - ▶ 结构类型的构造
 - ▶ 结构变量的定义与初始化
 - ▶ 结构类型数据的操作
- ▶ 结构类型数组
- ▶ 用指针操作结构类型数据
- ▶ 联合类型

结构类型的构造

```
struct Student
{
    int number; //成员
    char name;  //成员
    int age;    //成员
};
```

宣布组成的
成员名称和成员类型

```
typedef struct
{
    int month;
    int day;
    int year;
} Date;
```

注意：构造结构类型时，
花括号中至少要定义一个成员。
除void类型和本结构类型外，
结构成员可以是其它任意的类型。

```
struct Employee
```

```
{
```

```
    int number;
```

```
    char name;
```

```
    struct Date
```

```
{
```

```
        int year;
```

```
        int month;
```

```
        int day;
```

```
struct Date
```

```
{
```

```
    int year;
```

```
    int month;
```

```
    int day;
```

```
};
```

```
struct Employee
```

```
{
```

```
    int number;
```

```
    char name;
```

```
    Date birthday;
```

```
} e;
```

```
} birthday ; //其它结构类型的变量可以作为本结构类型的成员
```

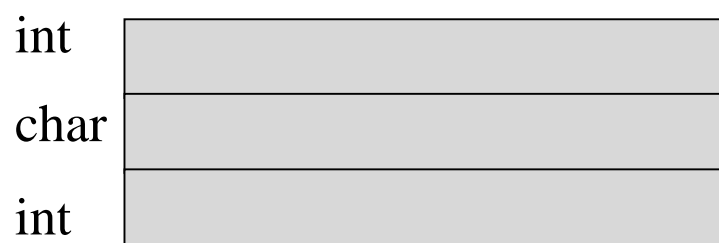
```
} e;
```

结构体变量的存储

- ▶ 系统按构造时的顺序为各个成员分配空间
- ▶ 系统往往以（最大元素所占）字节为单位给结构变量分配空间



×



- ▶ 结构变量一般不加register修饰

结构体变量的存储

例1:

```
struct {  
    short a1;  
    short a2;  
    short a3;  
} A;  
  
struct {  
    long a1;  
    short a2;  
} B;
```

`sizeof(A) = 6`; 这个很好理解，三个short都为2。

`sizeof(B) = 8`; 这个比是不是比预想的大2个字节？long为4，short为2，整个为8，因为原则3。

例2: struct A{

```
    int a;  
    char b;  
    short c;  
};  
  
struct B{  
    char b;  
    int a;  
    short c;  
};
```

`sizeof(A) = 8`; int为4，char为1，short为2，这里用到了原则1和原则3。

`sizeof(B) = 12`; 是否超出预想范围？char为1，int为4，short为2，怎么会是12？还是原则1和原则3。

a b c

A的内存布局: 1111, 1*, 11

b a c

B的内存布局: 1***, 1111, 11**

结构类型数据作为函数参数/返回值

- ▶ 可作为参数传给函数：默认参数传递方式为**值传递**

(实参和形参都是结构变量名，类型相同；但实参和形参代表两个不同的结构变量，运行时分配不同的存储空间)
- ▶ 函数也可以返回一个结构类型的值（结构型函数）

结构类型数组

- ▶ 结构数组可用于表示二维表格。比如，

```
Stu stu_array[5];    //定义了一个一维结构数组
```

```
Stu stu_array[5] = { {1001, 'T', 'M', 20, 90.0}, ...,  
                    {1005, 'L', 'F', 18, 81.0} }; //初始化
```

用指针操作结构类型数据*

- ▶ 将结构类型变量的地址赋给基类型为该结构类型的指针变量，则可利用这个指针操作该结构变量的成员，这时成员操作符写成箭头->的形式，比如，

```
struct
```

```
{
```

```
    int no;
```

```
    float score;
```

```
} s, *ps;
```

```
ps = &s;
```

```
ps -> no = 1001;
```

```
//相当于(*ps).no或s.no
```

```
ps -> score = 90.0;
```

```
//相当于(*ps).score或s.score
```


- ▶ 如果有成员是另一结构类型的指针变量，则可以用若干个箭头形式的成员操作符访问最低一级的成员。比如，

```
struct
{
    Student *p1;
    float score;
} s, *pps ;
pps = &s;
pps -> p1 = &s1;
pps -> p1 -> number = 1220001;
pps -> p1 -> name = 'Q';
pps -> p1 -> age = 19;
pps -> score = 85;
```

```
struct Student
{
    int number; //成员
    char name;  //成员
    int age;    //成员
}s1;
```

- ▶ 结构类型不可以含有本结构类型成员。
- ▶ 但结构类型可以含有**本结构类型的指针成员**。比如，

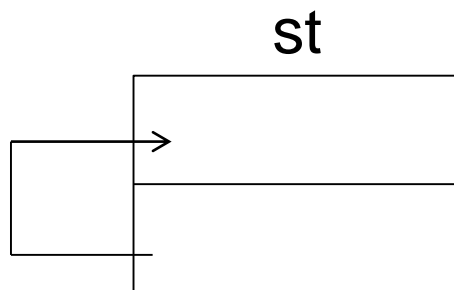
```
struct Stup
```

```
{
```

```
    int no;
```

```
    Stup *p0;
```

```
} st;
```



st.p0 = &st; // 存放本身(st)的地址

- ▶ 结构类型不可以含有本结构类型成员。
- ▶ 但结构类型可以含有**本结构类型的指针成员**。比如，

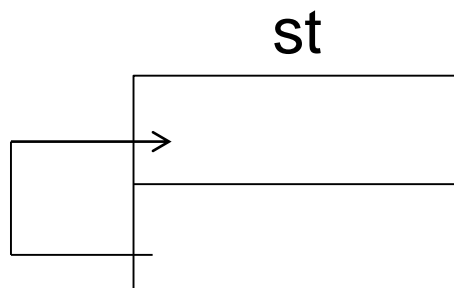
```
struct Stup
```

```
{
```

```
    int no;
```

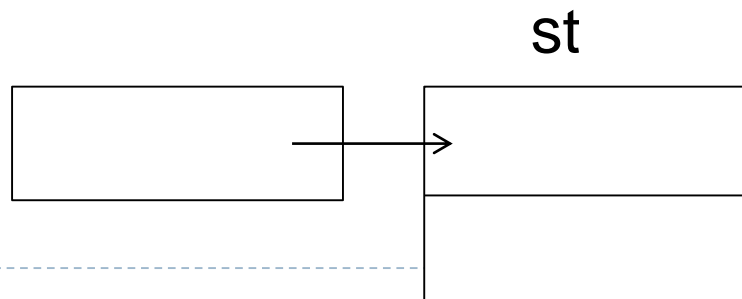
```
    Stup *p0;
```

```
} st;
```



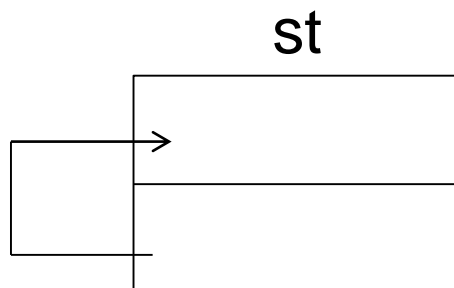
`st.p0 = &st; // 存放本身(st)的地址`

`Stup *pst = &st;`



- ▶ 结构类型不可以含有本结构类型成员。
- ▶ 但结构类型可以含有**本结构类型的指针成员**。比如，

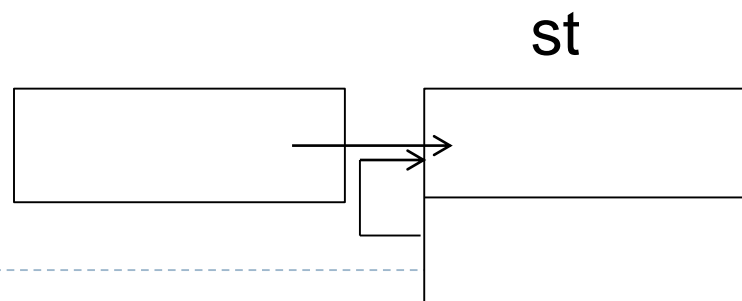
```
struct Stup  
{  
    int no;  
    Stup *p0;  
} st;
```



st.p0 = &st; // 存放本身(st)的地址

Stup *pst = &st;

pst -> p0 = &st;

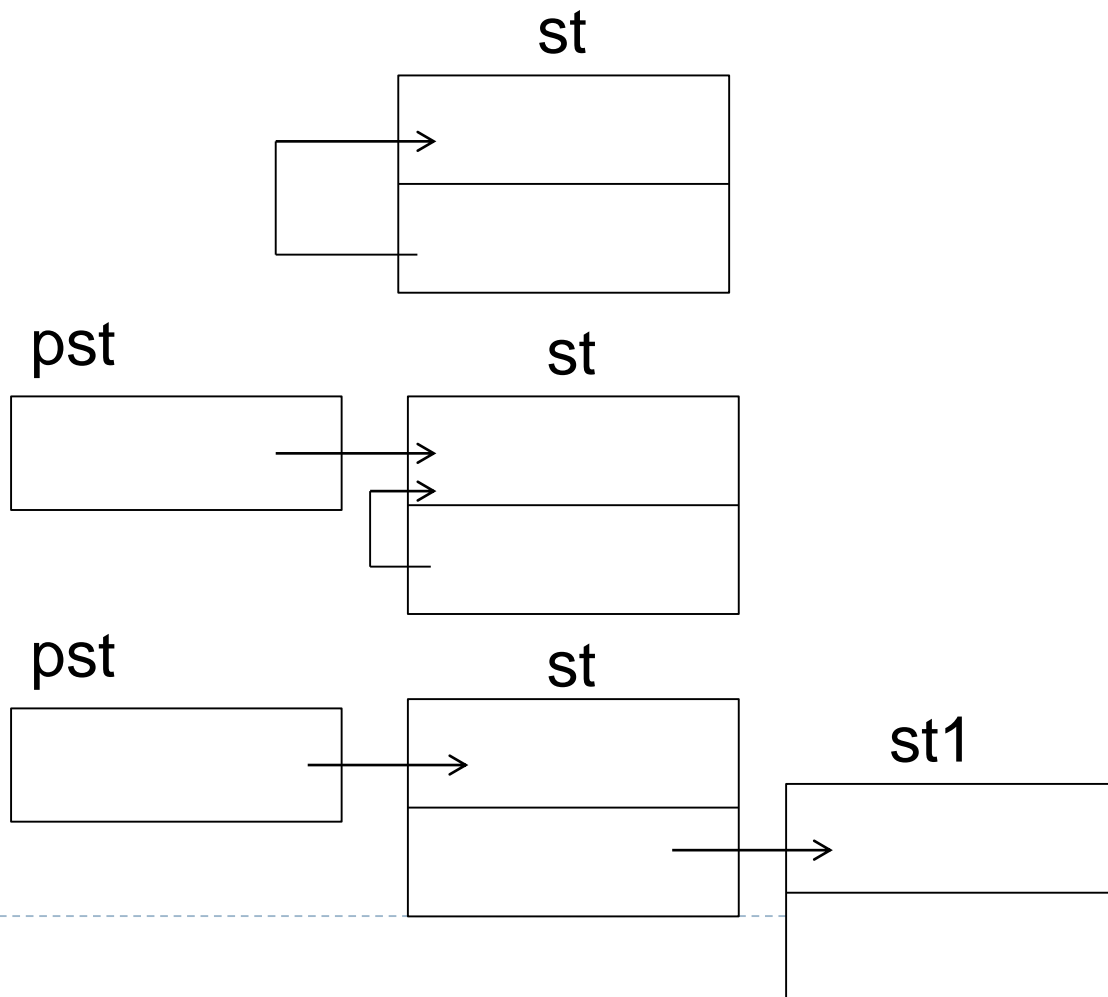


- ▶ 结构类型不可以含有本结构类型成员。
- ▶ 结构类型可以含有本结构类型的指针成员。比如，

```
struct Stup
{
    int no;
    Stup *p0;
} st;
```

```
st.p0 = &st;
Stup *pst = &st;
pst -> p0 = &st;
```

```
Stup st1;
pst -> p0 = &st1;
```



联合（union）类型

```
union myType  
{  
    int i;  
    char c;  
    double d;  
};
```

与结构类型类似，
联合类型由程序员构造而成，
构造时需要用到关键词union。

联合变量的初始化、成员的操作方式
也与结构变量类似。

```
myType v;           //定义了一个myType类型的联合变量v
```

对联合变量的分时操作

- ▶ 对于上述联合变量v，在程序中可以分时操作其中不同数据类型的成员。比如，

```
v.i = 12;           //以下只操作变量v的成员i
```

.....

```
v.c = 'X' ;        //以下只操作变量v的成员c
```

.....

```
v.d = 12.95;       //以下只操作变量v的成员d
```

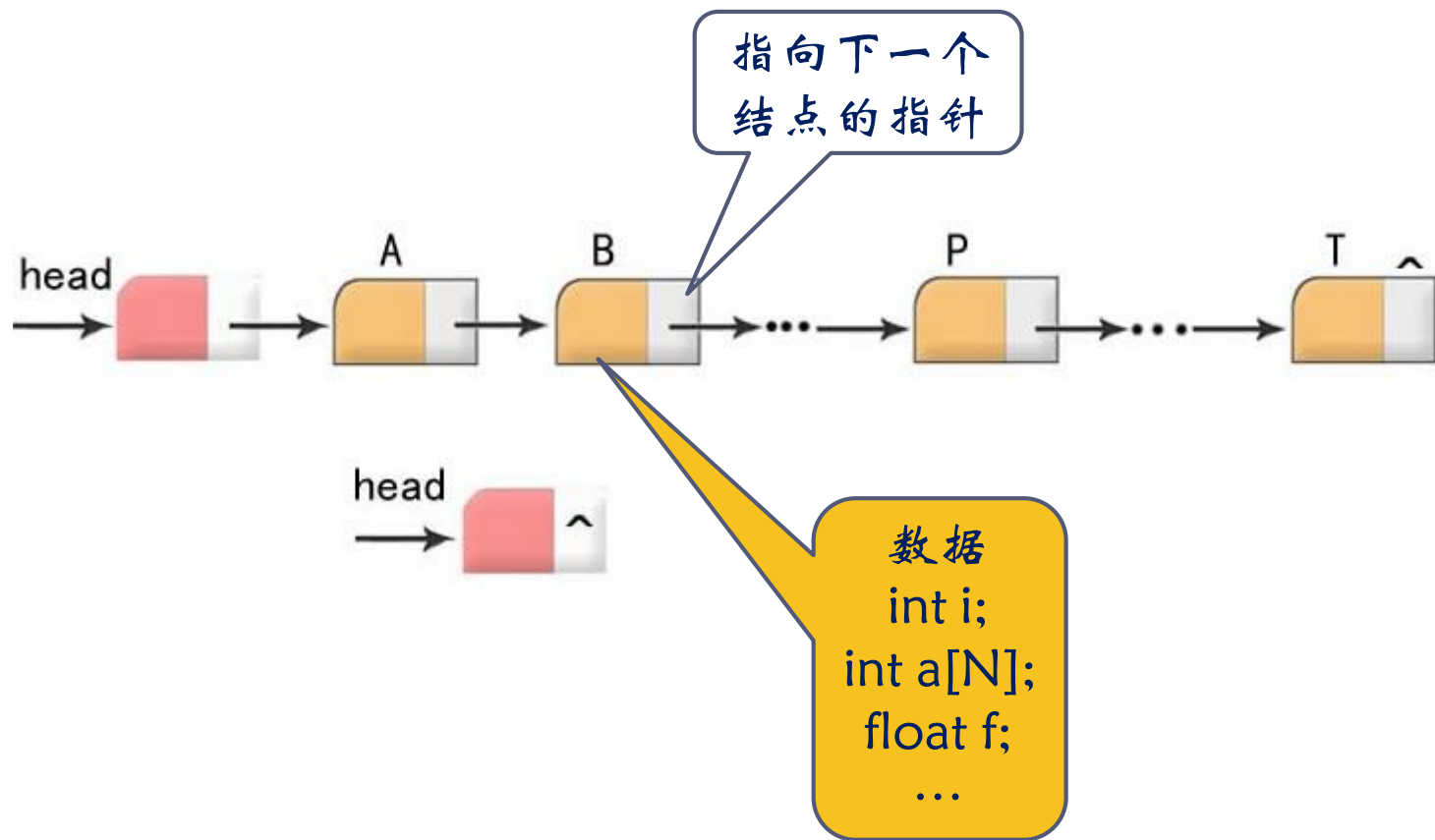
.....

- ▶ 当给一个联合变量的某成员赋值后，再访问该变量的另外一个成员，将得不到原来的值。比如，

```
v.i = 12;  
printf("%lf", v.d);           //不会输出12.95
```

- ▶ 即可以分时把v当作不同类型的变量来使用，但不可以同时把v当作不同类型的变量来使用。

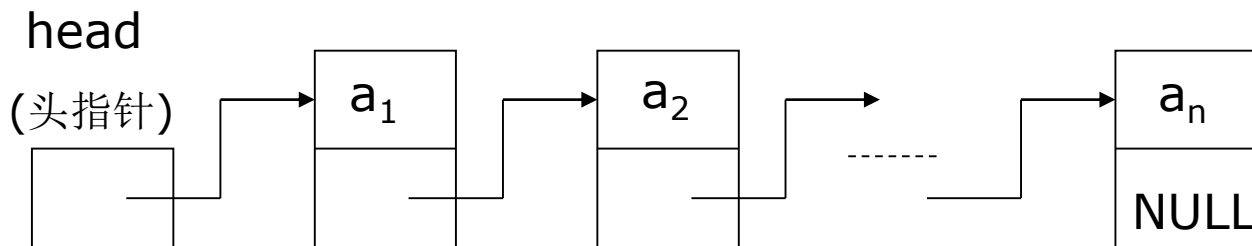
链表



动态变量的应用——链表

- ▶ 链表用于表示由若干个同类型的元素所构成的具有线性结构的复合数据。
 - ▶ 链表元素在内存中不必存放在连续的空间内。
 - ▶ 链表中的每一个元素除了本身的数据外，还包含一个（或多个）指针，它（们）指向链表中下一个（前一个或其它）元素。
 - ▶ 如果每个元素只包含一个指针，则称为单链表，否则称为多链表。
-

单链表



- ▶ 单链表的每个元素只包含一个“指向相邻节点”的指针。
- ▶ 需要一个**头指针**，指向第一个元素。
- ▶ 单链表中的结点类型和表头指针变量可定义如下：

```
struct Node           //结点的类型定义
{
    int content;       //代表结点的数据
    Node *next;        //代表后一个结点的地址
};
```

```
Node *head=NULL; //头指针变量定义，初始状态下
```

▶ //为空值。NULL在cstdio中定义为0

链表操作

基本操作：

- ▶ 链表的建立
- ▶ 链表的遍历和元素定位（头、尾、特定位置和特定值）
- ▶ 链表中节点的插入（头、尾、特定位置）与删除
- ▶ 链表的输出与删除

衍生操作：

- ▶ 链表的反转
- ▶ 基于链表的排序
- ▶ 特定元素检索程序
- ▶ ...

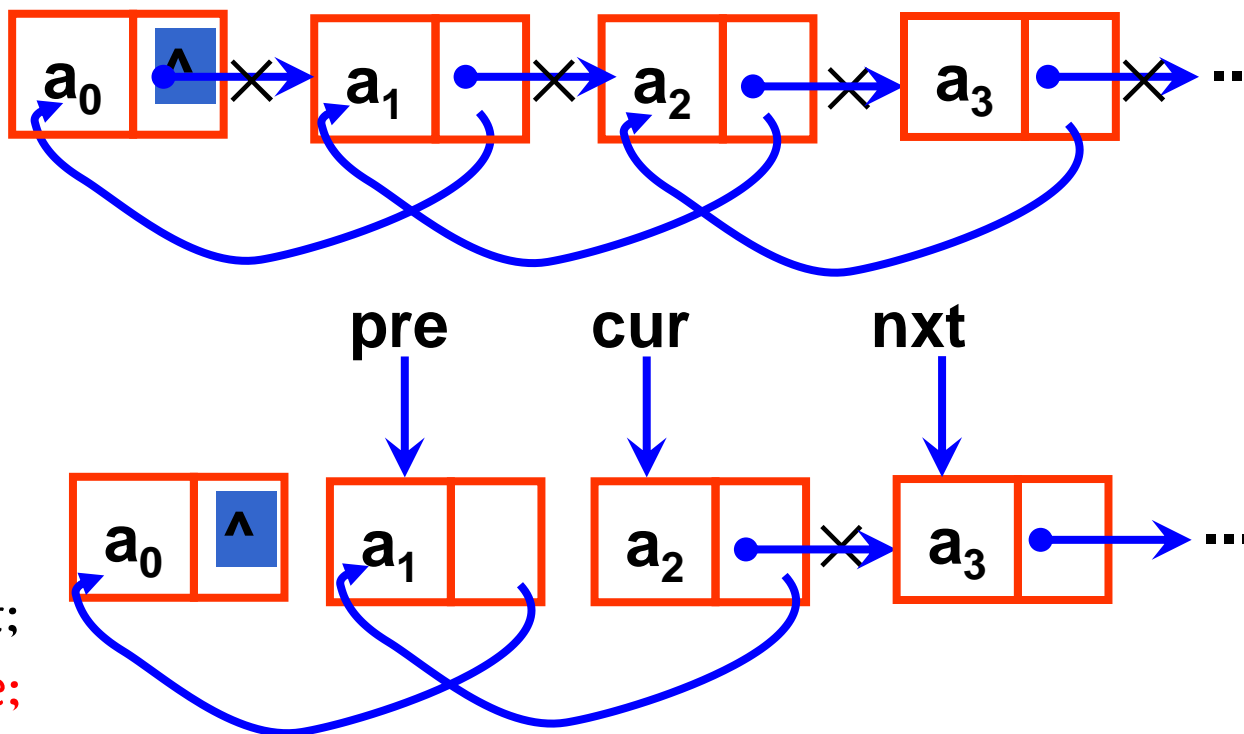


链表的反转

- ▶ 链表的反转指的是将链表的前后关系颠倒，头节点变成尾节点，尾节点变成头节点。

```
Node * Reverse(Node *head)
```


```
{
    Node *pre = NULL;
    Node *cur = NULL;
    Node *nxt = head;
    while(nxt != NULL)
    {
        pre = cur;
        cur = nxt;
        nxt = cur -> next;
        cur -> next = pre;
    }
    return cur;
}
```



其实，用双向链表，易于实现链表的反转！


例：综合应用：对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）

```
struct Node
{
    int content; //代表结点的数据
    Node *next; //代表后一个结点的地址
};
extern Node *input();           //输入数据，建立链表，返回链表的头指针
extern void sort(Node *h);      //排序
extern void output(Node *h);    //输出数据
extern void remove(Node *h);    //删除链表
int main()
{
    Node *head;
    head = input();
    sort(head);
    output(head);
    remove(head);
    return 0;
}
```



```
#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表尾插入数据
{ Node *head=NULL, //头指针
  *tail=NULL; //尾指针

  int x;
  cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = NULL;
    if (head == NULL)
      head = p;
    else
      tail->next = p;
    tail = p;
    cin >> x;
  }
  return head;
}
```



```
#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表头插入数据
{ Node *head=NULL; //头指针
  int x;
  cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = head;
    head = p;
    cin >> x;
  }
  return head;
}
```

特别注意！

建立链表，头指针作为形参

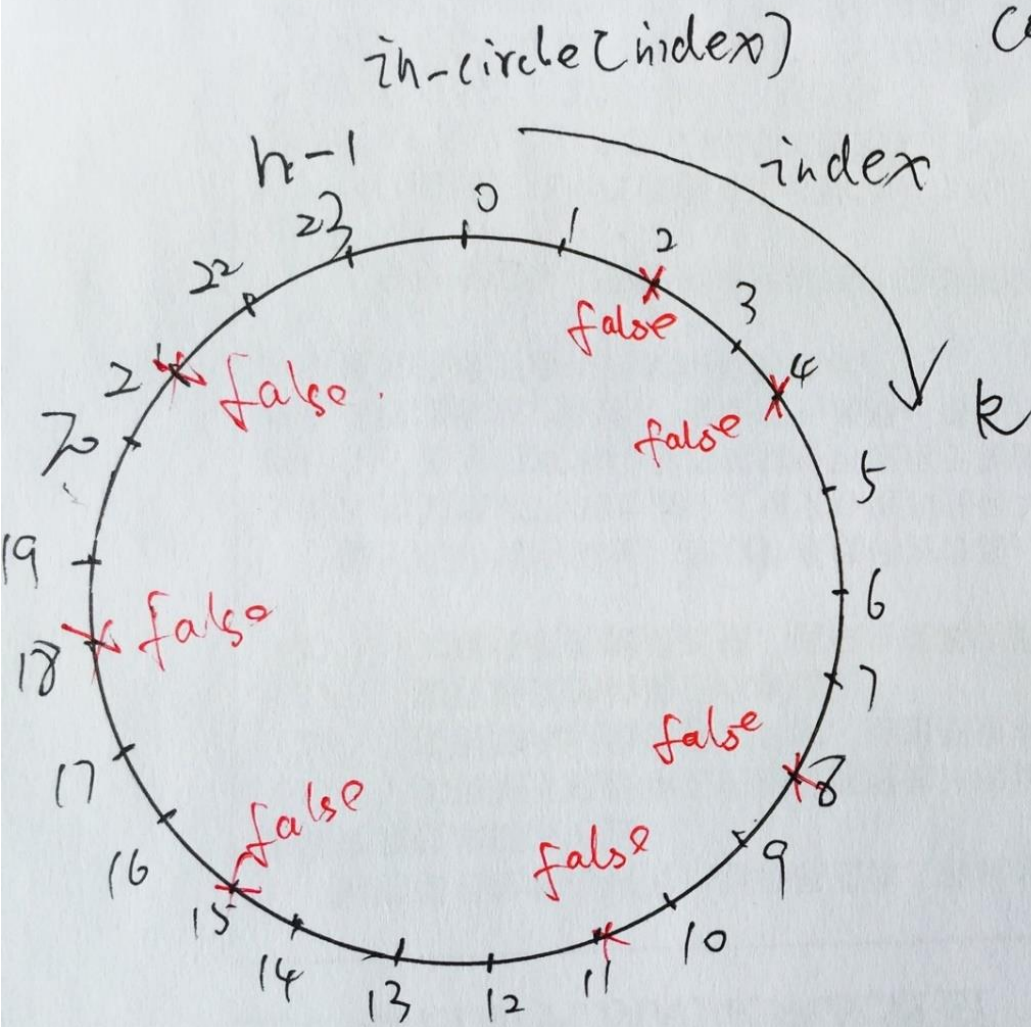
```
.....  
void input(Node *&h) //从表头插入数据，建立链表，h返回头指针  
{  
    int x;  
    cin >> x;  
    while (x != -1)  
    {  
        Node *p=new Node;  
        p->content = x;  
        p->next = h;  
        h = p;  
        cin >> x;  
    }  
}  
int main()  
{  
    Node *head=NULL;  
    input( head);  
    .....  
}
```

*&h

例：用链表实现求解约瑟夫问题

回顾：约瑟夫斯（Josephus）问题

- ▶ 有 n 个小孩站成一圈做游戏。从某个开始顺时针报数，报到 k 的小孩从圈子离开；然后，从下一个人开始重新报数，每报到 k ，相应的小孩从圈子离开；最后只剩下一个小孩；请问这个小孩一开始站在什么位置？



count: 当前计数器
初始为0
=k 则离开.

```
while( ) // 当 count=k
{
    if ( )
        count++;
        index++;
}
```

例：用链表实现求解约瑟夫问题

```
#include <iostream>
using namespace std;
struct Node
{ int no;          //小孩的编号
  Node *next; //指向下一个小孩的指针
};
int main()
{ int n;          //用于存储小孩的个数
  int m;          //用于存储要报的数
  num_of_children_remained; //用于存储圈子里
                           //剩下的小孩个数
  cout << "请输入小孩的个数和要报的数：";
  cin >> n >> m;
```



//构建圈子

```
Node *first,*last;           //first和last用于分别指向第一个和
                               //最后一个小孩子
first = last = new Node; //生成第一个结点
first->no = 0;                //第一个小孩子的编号为0
for (int i=1; i<n; i++) //循环构建其它小孩结点
{   Node *p=new Node; //生成一个小孩子结点
    p->no = i;          //新的小孩结点的编号为i
    last->next = p;     //最后一个小孩子的next指向新生成
                        //的小孩结点
    last = p; //把新生成的小孩结点成为最后一个结点
}
last->next = first;        //把最后一个小孩子的下一个小孩
                           //设为第一个小孩
```

//开始报数

```
num_of_children_remained = n;           //报数前的圈子中小孩个数
Node *previous=last; //previous指向开始报数的前一个小孩
while (num_of_children_remained > 1)
{
    for (int count=1; count<m; count++) //循环m-1次
        previous = previous->next;
    //循环结束时，previous指向将要离开圈子的小孩的前一个小孩
    Node *p=previous->next;           //p指向将要离圈的小孩结点
    previous->next = p->next;         //小孩离开圈子
    delete p;                        //释放离圈小孩结点的空间
    num_of_children_remained--;       //圈中小孩数减1
}
```

//输出胜利者的编号

```
cout << "The winner is No." << previous->no << "\n";
delete previous;           //释放胜利者结点的空间
return 0;
}
```

链表的构造与操作

基本操作：

- ▶ 链表的建立
- ▶ 链表的遍历和元素定位（头、尾、特定位置和特定值）
- ▶ 链表中节点的插入（头、尾、特定位置）与删除
- ▶ 链表的输出与删除

衍生操作：

- ▶ 链表的反转
- ▶ 基于链表的排序
- ▶ 特定元素检索程序

▶ ...

链表Coding Tips:

1.能够确定头(head)、尾(tail)、第i个节点

(最后一个node->next为空)

2.弄清指向关系和操作顺序!

3.留意特殊情况!

(例如:为空的初始情况或链表只有1个节点)

4.注意归还空间!



调试Tips

1. 对链表调试，重要的依然是断点（cout输出做辅助）

“数据为王”，把数据（content、甚至有时还有地址）打印出来可能会好点…

当牵涉Node交换（不仅是`p->content`），地址有时也要查看

2. 程序不是单靠“瞪大眼睛”看出来的，更重要的是针对测试用例，关键步骤的输出是否符合我们的“大脑”推理、手工计算

3. （对于复杂程序和测试用例）除非你有极强的逻辑思维和记忆力，否则请拿起笔和纸来…



调试Tips

```
struct Node
{
    int content; //代表结点的数据
    Node *next; //代表后一个结点的地址
};

int main()
{
    Node *head;
    head = input();
    sort(head);
    output(head);
    remove(head);
    return 0;
}
```

发现最后结果不对：

1. (Head, hand) 记下不对的例子（测试用例）
2. 从后向前（或从前往后）排查每个函数的输出是否正确，以确定那个函数的问题
3. 发现sort不对，也有可能是input的问题
- ...



调试Tips

```
void sort(Node *h) //采用选择排序，小的往前放
```

```
{ if (h == NULL || h->next == NULL) return;
```

```
    //从链表头开始逐步缩小链表的范围
```

```
    for (Node *p1=h; p1->next != NULL; p1 = p1->next)
```

```
    {
```

```
        Node *p_min=p1; //p_min指向最小的结点，初始化为p1
```

```
        //从p1的下一个开始与p_min进行比较
```

```
        for (Node *p2=p1->next; p2 != NULL; p2=p2->next)
```

```
            if (p2->content < p_min->content)
```

```
                p_min = p2;
```

```
        if (p_min != p1)
```

```
        {   int temp = p1->content;
```

```
            p1->content = p_min->content;
```

```
            p_min->content = temp;
```

```
        }
```

```
    }
```

```
}
```

当确定一个函数

有问题后…

指针的引用

特别注意!!!

.....

void input(Node *&h) //从表头插入数据，建立链表，h返回头指针

```
{ int x;
  cin >> x;
  while (x != -1)
  { Node *p=new Node;
    p->content = x;
    p->next = h;
    h = p;
    cin >> x;
  }
```

```
}
int main()
{ Node *head=NULL;
  input( head);
  .....
```

```
}
```

当在单链表的头部插入或删除结点时，链表的头部指针会发生改变，这时调用（在单链表插入或删除结点的）相应函数，如果链表的头指针作为被调用函数的实参（传给形参），需使用指针的引用，即：

Node *&h

如果作为返回值返回头指针呢？

链表元素的遍历：

```
for(p = head; p != NULL; p = p->next)
    ...
```

```
for(... ; head != NULL; head = head->next)
    ...
```

// 只要做一次 head=head->next, ... 因此以上...

回顾课程目标

- ▶ 掌握基本的程序设计概念和方法
- ▶ 能够用C/C++熟练语言编写程序，解决一般问题
- ▶ 培养良好的编程习惯！
- ▶ 为后续课程打基础





希望大家取得好成绩！

