附录 A　计数系统

人们使用很多计数系统来表示数字。有些计数系统（如罗马数字）不适合用于算术运算；而印度计数系统经过改进，并传入到欧洲后变成了阿拉伯计数系统，这种数字方便了数学、科学和商业计算。现代的计算机计数系统是基于占位符概念的，使用了最先出现在印度计数系统中的零。然而，这种原理被推广到其他计数系统。因此，虽然在日常生活中使用的是下一节将介绍的十进制，但计算领域通常使用八进制、十六进制和二进制。

A.1　十进制数

数字的书写方式是基于 10 的幂数。例如，对于数字 2468，2 表示 2 个 1000，4 表示 4 个 100，6 表示 6 个 10，8 表示 8 个 1。

$2468 = 2 \times 1000 + 4 \times 100 + 6 \times 10 + 8 \times 1$

一千是 $10 \times 10 \times 10$ 或 10 的 3 次幂，用 10^3 表示。使用这种表示法，可以这样书写上述关系：

$2468 = 2 \times 10^3 + 4 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$

因为这种数字表示法是基于 10 的幂，所以将它称作基数为 10 的表示法或十进制表示法。可以用任何数作基数。例如，C++允许使用基数 8（八进制）和基数 16（十六进制）来书写整数（请注意，10^0 为 1，任何非零数的 0 次幂都为 1）。

A.2　八进制整数

八进制数是基于 8 的幂的，所以基数为 8 的表示法用数字 0-7 来书写数字。C++用前缀 0 来表示八进制表示法。也就是说，0177 是一个八进制值。可以用 8 的幂来找到对应的十进制值：

八　进　制	十　进　制
177	$= 1 \times 8^2 + 7 \times 8^1 + 7 \times 8^0$
	$= 1 \times 64 + 7 \times 8 + 7 \times 1$
	$= 127$

由于 UNIX 操作系统常使用八进制来表示值，因此 C++和 C 提供了八进制表示法。

A.3　十六进制数

十六进制数是基于 16 的幂的。这意味着十六进制的 10 表示 16 + 0，即 16。为表示 9-16 值，需要其他一些数字，标准的十六进制表示法使用字母 a-f。C++接受这些字符的大写和小写版本，如表 A.1 所示。

表 A.1　　十六进制数

十六进制数	十 进 制 值
a 或 A	10
b 或 B	11
c 或 C	12
d 或 D	13
e 或 E	14
f 或 F	15

C++使用 0x 或 0X 来指示十六进制表示法。因此 0x2B3 是一个十六进制值，可使用 16 的幂来得到对应的十进制值。

十六进制	十 进 制
0x2B3	$= 2 \times 16^2 + 11 \times 16^1 + 3 \times 16^0$
	$= 2 \times 256 + 11 \times 16 + 3 \times 1$
	$= 691$

硬件文档常使用十六进制来表示诸如内存单元和端口号等值。

A.4 二进制数

不管是使用十进制、八进制，还是十六进制表示法来书写整数，计算机都将它存储为二进制值（即基数为 2）。二进制表示法只使用两个数字——0 和 1。例如，10011011 就是二进制数。但 C++没有提供二进制表示法来书写数字的方式。二进制数是基于 2 的幂。

二 进 制	十 进 制
10011011	$= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
	$= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$
	$= 155$

二进制表示法与计算机内存完全对应，在内存中，每个单元（位）都可以设置成开或关。只是将关表示为 0，将开表示为 1。内存通常是以字节为单位组织的，每个字节包含 8 位（正如第 2 章指出的，C++字节并非一定是 8 位，但本附录采用常见的做法，用字节表示八位组）。字节中的位被编号，对应于相关的 2 的幂。这样，最右侧的位编号为 0，然后是 1，依此类推。例如，图 A.1 表示一个 2 字节的整数。

$$\begin{aligned} 值 &= 1 \times 2^{11} + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^1 \\ &= 2048 + 256 + 32 + 2 \\ &= 2338 \end{aligned}$$

图 A.1 2 字节整数值

A.5 二进制和十六进制

十六进制表示法常用于提供更为方便的二进制数据（如内存地址或存储位标记设置的整数）视图。这样做的原因是，每个十六进制位对应于 4 位。表 A.2 说明了这种对应关系。

表 A.2 十六进制数和对应的二进制数

十六进制位	对应的二进制数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000

续表

十六进制位	对应的二进制数
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

要将十六进制值转换为二进制，只需将每个十六进制位替换为相应的二进制数即可。例如，十六进制 0xA4 对应于二进制数 10100100。同样，可以轻松地将二进制值转换为十六进制，方法是将每 4 位转换为对应的十六进制位。例如，二进制值 10010101 将被转换为 0x95。

Big Endian 和 Little Endian

奇怪的是，都使用整数的二进制表示的两个计算平台对同一个值的表示可能并不相同。例如，Intel 计算机使用 Little Endian 体系结构来存储字节，而 Motorola 处理器、IBM 大型机、SPARC 处理器和 ARM 处理器使用 Big Endian 方案（但最后的两种系统可配置成使用上述任何一种方案）。

术语 Big Endian 和 Little Endian 是从“Big End In”和“Little End In”（指内存中单词（通常为两个字节）的字节顺序）衍生而来的。在 Intel 计算机（Little Endian）中，先存储低位字节，这意味着十六进制值 0xABCD 在内存中将被存储为 0xCD 0xAB。Motorola（Big Endian）计算机按相反的顺序存储，因此 0xABCD 在内存中被存储为 0xAB 0xCD。

这些术语最先出现在 Jonathan Swift 编写的《Gulliver’s Travels》一书中。为讽刺众多政治斗争的非理性，Swift 杜撰了假想国中两个喜欢争论的政治派别：Big Endians 和 Little Endians，前者坚持认为从大的一头打破鸡蛋更合理，而后者坚持认为从小的一头打破鸡蛋更合理。

作为软件工程师，应了解目标平台的词序，它会影响对通过网络传输的数据的解释方式以及数据在二进制文件中的存储方式。在上面的例子中，二字节内存模式 0xABCD 在 Little Endian 计算机上表示十进制值 52651，而在 Big Endian 计算机上表示十进制值 43981。

附录 B　C++保留字

C++保留了一些单词供自己和 C++库使用。程序员不应将保留字用作声明中的标识符。保留字分三类：关键字、替代标记（alternative token）和 C++库保留名称。

B.1　C++关键字

关键字是组成编程语言词汇表的标识符，它们不能用于其他用途，如用作变量名。表 B.1 列出了 C++关键字，其中以粗体显示的关键字也是 ANSI C99 标准中的关键字，而以斜体显示的关键字是 C++11 新增的。

表 B.1　C++关键字

alignas	alignof	asm	**auto**	bool
break	**case**	catch	**char**	char16_t
char32_t	class	**const**	const_cast	constexpr
continue	decltype	**default**	delete	**do**
double	dynamic_cast	**else**	**enum**	explicit
export	**extern**	false	**float**	**for**
friend	**goto**	**if**	**inline**	**int**
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
register	reinterpret_cast	**return**	**short**	**signed**
sizeof	**static**	static_assert	static_cast	**struct**
switch	template	this	thread_local	throw
true	try	**typedef**	typeid	typename
union	**unsigned**	using	virtual	**void**
volatile	wchar_t	**while**		

B.2　替代标记

除关键字外，C++还有一些运算符的字母替代表示，它们被称为替代标记。替代标记也被保留，表 B.2 列出了替代标记及其表示的运算符。

表 B.2　C++保留的替代标记及其含义

标　　记	含　　义
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

B.3　C++库保留名称

编译器不允许程序员将关键字和替代标记用作名称。还有另一类禁止使用（但并非绝对不能用）的名称——保留名称，它们是保留给 C++库使用的名称。如果您将这种名称用作标识符，后果将是不确定的。也就是说，可能导致编译器错误、警告、程序不能正确运行或根本不会导致任何问题。

C++语言保留了库头文件中使用的宏名。如果程序包含某个头文件，则不应将该头文件（以及该头文件包含的头文件，依此类推）中定义的宏名用作其他目的。例如，如果您直接或间接地包含了头文件<climits>，则不应将 CHAR_BIT 用作标识符，因为它已被用作该头文件中一个宏的名称。

C++语言保留了以两个下划线或下划线和大写字母打头的名称，还将以单个下划线打头的名称保留用作全局变量。因此，程序员不能在全局名称空间使用诸如__gink、__Lynx 和_lynx 等名称。

C++语言保留了在库头文件中被声明为链接性为外部的名称。对于函数，这包括函数的特征标（名称和参数列表）。例如，假设有如下代码：

```
#include <cmath>
using namespace std;
```

则函数特征标 tan（double）被保留。这意味着您的程序不应声明一个原型如下所示的函数：

```
int tan(double); // don't do it
```

该原型确实与库函数 tan() 的原型不同，因为后者的返回类型为 double，但特征标部分确实相同。然而，定义下面的原型是可以的：

```
char * tan(char *); // ok
```

这是因为虽然其名称与库函数 tan()相同，但特征标不同。

B.4　有特殊含义的标识符

C++社区讨厌新增关键字，因为它们可能与现有代码发生冲突。这就是标准委员会改变关键字 auto 的用法，并赋予其他关键字（如 virtual 和 delete）新用法的原因所在。C++11 提供了另一种避免新增关键字的机制，即使用具有特殊含义的标识符。这些标识符不是关键字，但用于实现语言功能。编译器根据上下文来判断它们是常规标识符还是用于实现语言功能：

```
class F
{
    int final;                          // #1
public:
...
    virtual void unfold() {...} = final; // #2
};
```

在上述代码中，语句#1 中的 final 是一个常规标识符，而语句#2 中的 final 使用了一种语言功能。这两种用法彼此不会冲突。

另外，C++还有很多经常出现在程序中，但不被保留的标识符。这包括头文件名、库函数名和 main（必不可少的函数的名称，程序从该函数开始执行）。只要不发生名称空间冲突，就可将这些标识符用于其他目的，但没有理由这样做。也就是说，完全可以编写下面的代码，但常识告诉您不应这样做：

```
// allowable but silly
#include <iostream>
int iostream(int a);
int main ()
{
    std::
cout << iostream(5) << '\n';
    return 0;
}

int iostream(int a)
{
    int main = a + 1;
    int cout = a -1;
    return main*cout;
}
```

附录 C　ASCII 字符集

计算机使用数字代码来存储字符。ASCII 码是美国最常用的编码，它是 Unicode 的一个子集（一个非常小的子集）。C++使得能够直接表示大多数字符，方法是将字符用单引号括起，例如‘A’表示字符 A。也可以用前面带反斜杠的八进制或十六进制编码来表示单个字符，例如，‘\012’和‘\0xa’表示的都是换行符（LF）。这种转义序列还可放在字符串中，如“Hello, \012my dear”。

表 C.1 列出了以各种方式表示的 ASCII 字符集。在该表中，当被用作前缀时，^字符表示使用 Ctrl 键。

表 C.1　　ASCⅡ字符集

十　进　制	八　进　制	十 六 进 制	二　进　制	字　　符	ASCII 名称
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I，tab	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[，Esc	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	空格	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	”	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	

续表

十　进　制	八　进　制	十 六 进 制	二　进　制	字　　符	ASCII 名称
38	046	0x26	00100110	&	
39	047	0x27	00100111	'	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	'	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	

续表

十 进 制	八 进 制	十 六 进 制	二 进 制	字 符	ASCII 名称
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	-	
96	0140	0x60	01100000	'	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	Del	

附录 D　运算符优先级

运算符优先级决定了运算符用于值的顺序。C++运算符分为 18 个优先级组，如表 D.1 所示。第 1 组中的运算符的优先级最高，第 2 组中运算符的优先级次之，依此类推。如果两个运算符被用于同一个操作数，则首先应用优先级高的运算符。如果两个运算符的优先级相同，则 C++使用结合性规则来决定哪个运算符结合得更为紧密。同一组中运算符的优先级和结合性相同，不管是从左到右（表中 L-R）还是从右到左（表中 R-L）结合。从左到右的结合性意味着首先应用最左边的运算符，而从右到左的结合性则意味着首先应用最右边的运算符。

表 D.1　　C++运算符的优先级和结合性

运　算　符	结　合　性	含　　义
优先级第 1 组		
::		作用域解析运算符
优先级第 2 组		
（表达式）		分组
()	L-R	函数调用
()		值构造，即 type(expr)
[ ]		数组下标
->		间接成员运算符
.		直接成员运算符
const_cast		专用的类型转换
dynamic_cast		专用的类型转换
reinterpret_cast		专用的类型转换
static_cast		专用的类型转换
typeid		类型标识
++		加 1 运算符，后缀
- -		减 1 运算符，后缀
优先级第 3 组（全是一元运算符）		
!	R-L	逻辑非
~		位非
+		一元加号（正号）
-		一元减号（负号）
++		加 1 运算符，前缀
- -		减 1 运算符，前缀
&		地址
*		解除引用（间接值）
()		类型转换，即(type)expr
sizeof		长度，以字节为单位
new		动态分配内存
new [ ]		动态分配数组
delete		动态释放内存
delete [ ]		动态释放数组

续表

运 算 符	结 合 性	含 义
优先级第 4 组		
. *	L-R	成员解除引用
->*		间接成员解除引用
优先级第 5 组（全是二元运算符）		
*	L-R	乘
/		除
^		模（余数）
优先级第 6 组（全是二元运算符）		
+	L-R	加
-		减
优先级第 7 组		
<<	L-R	左移
>>		右移
优先级第 8 组		
<	L-R	小于
<=		小于或等于
>=		大于或等于
>		大于
优先级第 9 组		
= =	L-R	等于
!=		不等于
优先级第 10 组（一元运算符）		
&	L-R	按位 AND
优先级第 11 组		
^	L-R	按位 XOF（异或）
优先级第 12 组		
|	L-R	按位 OR
优先级第 13 组		
&&	L-R	逻辑 AND
优先级第 14 组		
| |	L-R	逻辑 OR
优先级第 15 组		
:?	R-L	条件
优先级第 16 组		
=	R-L	简单赋值
* =		乘并赋值
/=		除并赋值
%=		求模并赋值
+=		加并赋值
-=		减并赋值
&=		按位 AND 并赋值
^=		按位 XOR 并赋值
|=		按位 OR 并赋值
<<=		左移并赋值
>>=		右移并赋值

续表

运　算　符	结　合　性	含　　义
优先级第 17 组		
throw	L-R	引发异常
优先级第 18 组		
,	L-R	将两个表达式合并成一个

有些符号（如*或&）被用作多个运算符。在这种情况下，一种形式是一元（一个操作数），另一种形式是二元（两个操作数），编译器将根据上下文来确定使用哪种形式。对于同一个符号可以两种方式使用的情况，表 D.1 将运算符标记为一元组或二元组。

下面介绍一些优先级和结合性的例子。

对于下面的例子，编译器必须决定先将 5 和 3 相加，还是先将 5 和 6 相乘：

```
3 + 5 * 6
```

*运算符的优先级比+运算符高，所以它被首先用于 5，因此表达式变成 3 +30，即 33。

对于下面的例子，编译器必须决定先将 120 除以 6，还是先将 6 和 5 相乘：

```
120 / 6 * 5
```

/和*的优先级相同，但这些运算符从左到右结合的。这意味着首先应用操作数（6）左侧的运算符，因此表达式变为 20*5，即 100。

对于下面的例子，编译器必须决定先对 str 递增还是先对 str 解除引用：

```
char * str = "Whoa";
char ch = *str++;
```

后缀++运算符的优先级比一元运算符*高，这意味着加号运算符将对 str 进行操作，而不是对*str 进行操作。也就是说，将指针加 1，使之指向下一个字符，而不是修改被指针指向的字符。不过，由于++是后缀形式，因此将在将*str 的值赋给 ch 后，再将指针加 1。因此，上述表达式将字符 W 赋给 ch，然后移动指针 str，使之指向字符 h。

下面是一个类似的例子：

```
char * str = "Whoa";
char ch = *++str;
```

前缀++运算符和一元运算符*的优先级相同，但它们是从右到左结合的。因此，str（不是*str）将被加 1。因为++运算符是前缀形式，所以首先将 str 加 1，然后将得到的指针执行解除引用的操作。因此，str 将指向字符 h，并将字符 h 赋给 ch。

注意，表 D.1 在“优先级”行中使用一元或二元来区分使用同一个符号的两个运算符，如一元地址运算符和二元按位 AND 运算符。

附录 B 列出了一些运算符的替代表示。

附录 E　其他运算符

为了避免篇幅过长，有三组运算符没有在本书正文部分介绍。第一组是按位运算符，能够操纵值中的各个位；这些运算符是从 C 语言继承而来的；第二组运算符是两个成员解除引用运算符，它们是 C++新增的；第三组是 C++11 新增的运算符：alignof 和 noexcept。本附录将简要地对这些运算符做一总结。

E.1　按位运算符

按位运算符对整数值的位进行操作。例如，左移运算符将位向左移，按位非运算符将所有的 1 变成 0，所有的 0 变成 1，C++共有 6 个这样的运算符：<<、>>、~、&、|和^。

E.1.1　移位运算符

左移运算符的语法如下：

```
value << shift
```

其中，value 是要被操作的整数值，shift 是要移动的位数。例如，下面的代码将值 13 的所有位都向左移 3 位：

```
13 << 3
```

腾出的位置用 0 填充，超出边界的位被丢弃（参见图 E.1）。

由于每个位都表示右边一位的 2 倍（参见附录 A），所以左移一位相当于乘以 2。同样，左移 2 位相当于乘以 2^2，左移 n 位相当于乘以 2^n。因此，13<<3 的值为 13×2^3，即 104。

图 E.1　左移运算符

左移运算符提供了通常可以在汇编语言中找到的功能。不过，左移运算符在汇编语言中会直接修改寄存器的内容，而 C++左移运算符生成一个新值，而不修改原来的值。例如，请看下面的代码：

```
int x = 20;
int y = x << 3;
```

上述代码不会修改 x 的值。表达式 x<<3 使用 x 的值来生成一个新值，就像 x+3 会生成一个新值，而不会修改 x 一样。

如果要用左移运算符来修改变量的值，则还必须使用赋值运算符。可以使用常规的赋值运算符或<<=运算符（该运算符将移动与赋值结合在一起）：

```
= x << 4; // regular assignment
y <<= 2; // shift and assign
```

正如所期望的，右移运算符（>>）将位向右移，其语法如下：

```
value >> shift
```

其中，value 是要移动的整数值，shift 是要移动的位数。例如，下面的代码将值 17 中所有的位向右移两位：

```
17 >> 2
```

对于无符号整数，腾出的位置用 0 填充，超过边界的位被删除。对于有符号整数，腾出的位置可能用 0 填充，也可能用原来最左边的位填充，这取决于 C++实现（图 E.2 是一个用 0 填充的例子）。

图 E.2　右移运算符

向右移动一位相当于除以 2。向右移动 n 位相当于除以 2^n。

C++还定义了一个“右移并赋值”运算符，如果要用移动后的值替换变量的值，可以这样做：

```
int q = 43;
q >>= 2;              // replace 43 by 43 >> 2, or 10
```

在有些系统上，使用左移运算符（右移运算符）实现将整数乘（除）以 2 的速度比使用乘（除）法运算符更快，但由于编译器在优化代码方面越来越好，因此这种差异正在减小。

E.1.2　逻辑按位运算符

逻辑按位运算符类似于常规的逻辑运算符，只是它们用于值的每一位，而不是整个值。例如，请看常规的非运算符（!）和位非（或求反）运算符（～）。!运算符将 true（或非零值）转换为 false，将 false 值转换为 true。～运算符将每一位转换为它的反面（1 转换为 0，0 转换为 1）。例如，对于 unsigned char 值 3：

```
unsigned char x = 3;
```

表达式!x 的值为 0。要知道~x 的值，先把它写成二进制形式：00000011。然后将每个 0 转换为 1，将每个 1 转换为 0。这样将得到值 11111100，在十进制中，为 252（图 E.3 是一个 16 位的例子）。新值是原值的补值。

图 E.3　按位非运算符

按位运算符 OR（|）对两个整数值进行操作，生成一个新的整数值。如果被操作的两个值的对应位至少有一个为 1，则新值中相应位为 1，否则为 0（参见图 E.4）。

图 E.4 按位运算符 OR

表 E.1 对 | 运算符的操作方式进行了总结。

表 E.1 b1|b2 的值

位 值	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	1

运算符| =组合了按位运算符 OR 与赋值运算符的功能：

```
a |= b; // set a to a | b
```

按位运算符 XOR（^）将两个整数值结合起来，生成一个新的整数值。如果原始值中对应的位有一个（而不是两个）为 1，则新值中相应位为 1；如果对应的位都为 0 或 1，则新值中相应位为 0（参见图 E.5）。

图 E.5 按位运算符 XOR

表 E.2 总结了^运算符的结合方式。

表 E.2 b1^b2 的值

位 值	b1 = 0	b1 = 1
b2 = 0	0	1
b2 = 1	1	0

^ =运算符结合了按位运算符 XOR 和赋值运算符的功能：

```
a ^= b; // set a to a ^ b
```

按位运算符 AND（&）将两个整数结合起来，生成一个新的整数值。如果原始值中对应位都为 1，则新值中相应位为 1，否则为 0（参见图 E.6）。

图 E.6　按位运算符 AND

表 E.3 总结了&运算符是如何运算的。

表 E.3　　b1&b2 的值

位　　值	b1 = 0	b2 = 1
b2 = 0	0	0
b2 = 1	0	1

& =运算符结合了按位运算符 AND 和赋值运算符的功能：

```
a &= b; // set a to a & b
```

E.1.3　按位运算符的替代表示

对于几种按位运算符，C++提供了替代表示，如表 E.4 所示。它们适用于字符集中不包含传统按位运算符的区域。

表 E.4　　按位运算符的替代表示

标 准 表 示	替 代 表 示
&	bitand
& =	and_eq
|	bitor
| =	or_eq
～	compl
^	xor
^ =	xor_eq

这些替代表示让您能够编写下面这样的语句：

```
b = compl a bitand b; // same as b = ~a & b;
c = a xor b;          // same as c = a ^ c;
```

E.1.4　几种常用的按位运算符技术

控制硬件时，常涉及打开/关闭特定的位或查看它们的状态。按位运算符提供了执行这种操作的途径。下面简要地介绍一下这些方法。

在下面的示例中，lottabits 表示一个值，bit 表示特定位的值。位从右到左进行编号，从 0 开始，因此，第 n 位的值为 2^n。例如，只有第 3 位为 1 的整数的值为 2^3（8）。一般来说，正如附录 A 介绍的，各个位都对应于 2 的幂。因此我们使用术语位（bit）表示 2 的幂；这对应于特定位为 1，其他所有位都为 0 的情况。

1. 打开位

下面两项操作打开 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits | bit;
lottabits |= bit;
```

它们都将对应的位设置为 1，而不管这一位以前的值是多少。这是因为对 1 和 1 或者 0 和 1 执行 OR 操作时，都将得到 1。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 做 OR 操作将得到 0，对 1 和 0 做 OR 操作将生成 1。

2. 切换位

下面两项操作切换 lottabits 中对应于 bit 表示的位。也就是说，如果位是关闭的，则将被打开；如果位是打开的，将被关闭：

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

对 0 和 1 执行 XOR 操作的结果为 1，因此将关闭已打开的位；对 1 和 1 执行 XOR 操作的结果为 0，因此将打开已关闭的位。lottabits 中其他所有位都保持不变，这是因为对 0 和 0 执行 XOR 操作的结果为 0，对 1 和 0 执行 XOR 操作的结果为 1。

3. 关闭位

下面的操作将关闭 lottabits 中对应于 bit 表示的位：

```
lottabits = lottabits & ~bit;
```

该语句关闭相应的位，而不管它以前的状态如何。首先，运算符～bit 将原来为 1 的位设置为 0，原来为 0 的位设置为 1。对 0 和任意值执行 AND 操作都将得到 0，因此关闭相应的位。lottabits 中其他所有位都保持不变，这是因为对 1 和任意值执行 AND 操作时，该位的值将保持不变。

下面是一种更简洁的方法：

```
lottabits &= ~bit;
```

4. 测试位的值

如果要确定 lottabits 中对应于 bit 的位是否为 1，则下面的测试不一定管用：

```
if (lottabits == bit) // no good
```

这是因为即使 lottabits 中对应的位为 1，而其他位也可能为 1。仅当对应的位为 1，而其他位皆为 0 时，上述等式才为 true。因此修补的方式是，首先对 lottabits 和 bit 执行 AND 操作，这样生成的值的对应位保持不变，因为对 1 和任何值执行 AND 操作都将保持该值不变；而其他位都为 0，因为对 0 和任何值执行 AND 操作的结果都为 0。正确的测试如下：

```
if (lottabits & bit == bit) // testing a bit
```

实际应用中，程序员常将上述测试简化为：

```
if (lottabits & bit)        // testing a bit
```

因为 bit 中有一位为 1，而其他位都为 0，因此 lottabits & bit 的结果要么为 0（测试结果为 false），要么为 bit（非零值，测试结果为 true）。

E.2 成员解除引用运算符

C++允许定义指向类成员的指针，对这种指针进行声明或解除引用时，需要使用一种特殊的表示法。为说明需要使用的特殊表示法，先来看一个样本类：

```
class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};
```

如果没有具体的对象，则 inches 成员只是一个标签。也就是说，这个类将 inches 定义为一个成员标识符，但要为它分配内存，必须声明这个类的一个对象：

```
Example ob; // now ob.inches exists
```

因此，可以结合使用标识符 inches 和特定的对象，来引用实际的内存单元（对于成员函数，可以省略对象名，但对象被认为是指针指向的对象）。

C++允许这样定义一个指向标识符 inches 的成员指针：

```
int Example::*pt = &Example::inches;
```

这种指针与常规指针有所差别。常规指针指向特定的内存单元，而 pt 指针并不指向特定的内存单元，因为

声明中没有指出具体的对象。指针 pt 指的是 inches 成员在任意 Example 对象中的位置。和标识符 inches 一样，pt 被设计为与对象标识符一起使用。实际上，表达式*pt 对标识符 inches 的角色做了假设，因此，可以使用对象标识符来指定要访问的对象，使用 pt 指针来指定该对象的 inches 成员。例如，类方法可以使用下面的代码：

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl; // display inches member of ob1
cout << ob2.*pt << endl; // display inches member of ob2
cout << po->*pt << endl; // display inches member of *po
```

其中，*和->*都是成员解除引用运算符（member dereferencing operator）。声明对象（如 obl）后，ob1.*pi 指的将是 ob1 对象的 inches 成员。同样，pq->*pt 指的是 pq 指向的对象的 inches 成员。

改变上述示例中使用的对象，将改变使用的 inches 成员。不过也可以修改 pt 指针本身。由于 feet 的类型与 inches 相同，因此可以将 pt 重新设置为指向 feet 成员（而不指向 inches 成员），这样 ob1.*pt 将是 ob1 的 feet 成员：

```
pt = &Example::feet;     // reset pt
cout << ob1.*pt << endl; // display feet member of ob1
```

实际上，*pt 相当于一个成员名，因此可用于标识（相同类型的）其他成员。

也可以使用成员指针来标识成员函数，其语法稍微复杂点。对于不带任何参数、返回值为 void 的函数，声明一个指向该函数的指针的代码如下：

```
void (*pf)(); // pf points to a function
```

声明指向成员函数的指针时，必须指出该函数所属的类。例如，下面的代码声明了一个指向 Example 类方法的指针：

```
void (Example::*pf)() const; // pf points to an Example member function
```

这表明 pf 可用于可使用 Example 方法的地方。注意，Example：：*pf 必须放在括号中，可以将特定成员函数的地址赋给该指针：

```
pf = &Example::show_inches;
```

注意，和普通函数指针的赋值情况不同，这里必须使用地址运算符。完成赋值操作后，便可以使用一个对象来调用该成员函数：

```
Example ob3(20);
(ob3.*pf)(); // invoke show_inches() using the ob3 object
```

必须将 ob3.*pf 放在括号中，以明确地指出，该表达式表示的是一个函数名。

由于 show_feet()的原型与 show_inches()相同，因此也可以使用 pf 来访问 show_feet()方法：

```
pf = &Example::show_feet;
(ob3.*pf)();       // apply show_feet() to the ob3 object
```

程序清单 E.1 中的类定义包含一个 use_ptr()方法，该方法使用成员指针来访问 Example 类的数据成员和函数成员。

程序清单 E.1　memb_pt.cpp

```
// memb_pt.cpp -- dereferencing pointers to class members
#include <iostream>
using namespace std;

class Example
{
private:
    int feet;
    int inches;
public:
    Example();
    Example(int ft);
    ~Example();
    void show_in() const;
    void show_ft() const;
    void use_ptr() const;
};

Example::Example()
{
    feet = 0;
    inches = 0;
```

```
}

Example::Example(int ft)
{
    feet = ft;
    inches = 12 * feet;
}

Example::~Example()
{
}
void Example::show_in() const
{
    cout << inches << " inches\n";
}

void Example::show_ft() const
{
    cout << feet << " feet\n";
}

void Example::use_ptr() const
{
    Example yard(3);
    int Example::*pt;
    pt = &Example::inches;
    cout << "Set pt to &Example::inches:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    pt = &Example::feet;
    cout << "Set pt to &Example::feet:\n";
    cout << "this->pt: " << this->*pt << endl;
    cout << "yard.*pt: " << yard.*pt << endl;
    void (Example::*pf)() const;
    pf = &Example::show_in;
    cout << "Set pf to &Example::show_in:\n";
    cout << "Using (this->*pf)(): ";
    (this->*pf)();
    cout << "Using (yard.*pf)(): ";
    (yard.*pf)();
}

int main()
{
    Example car(15);
    Example van(20);
    Example garage;

    cout << "car.use_ptr() output:\n";
    car.use_ptr();
    cout << "\nvan.use_ptr() output:\n";
    van.use_ptr();

    return 0;
}
```

下面是程序清单 E.1 中程序的运行情况：

```
car.use_ptr() output:
Set pt to &Example::inches:
this->pt: 180
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 15
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 180 inches
Using (yard.*pf)(): 36 inches

van.use_ptr() output:
Set pt to &Example::inches:
this->pt: 240
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 20
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 240 inches
Using (yard.*pf)(): 36 inches
```

这个例子在编译期间给指针赋值。在更复杂的类中，可以使用指向数据成员和方法的成员指针，以便在运行阶段确定与指针关联的成员。

E.3　alignof（C++11）

计算机系统可能限制数据在内存中的存储方式。例如，一个系统可能要求 double 值存储在编号为偶数的内存单元中，而另一个系统可能要求其起始地址为 8 个整数倍。运算符 alignof 将类型作为参数，并返回一个整数，指出要求的对齐方式。例如，对齐要求可能决定结构中信息的组织方式，如程序清单 E.2 所示。

程序清单 E.2　align.cpp

```
// align.cpp -- checking alignment
#include <iostream>
using namespace std;
struct things1
{
    char ch;
    int a;
    double x;
};
struct things2
{
    int a;
    double x;
    char ch;
};

int main()
{
    things1 th1;
    things2 th2;
    cout << "char alignment: " << alignof(char) << endl;
    cout << "int alignment: " << alignof(int) << endl;
    cout << "double alignment: " << alignof(double) << endl;
    cout << "things1 alignment: " << alignof(things1) << endl;
    cout << "things2 alignment: " << alignof(things2) << endl;
    cout << "things1 size: " << sizeof(things1) << endl;
    cout << "things2 size: " << sizeof(things2) << endl;
    return 0;
}
```

下面是该程序在一个系统中的输出：

```
char alignment: 1
int alignment: 4
double alignment: 8
things1 alignment: 8
things2 alignment: 8
things1 size: 16
things2 size: 24
```

两个结构的对齐要求都是 8。这意味着结构长度将是 8 的整数倍，这样创建结构数组时，每个元素的起始位置都是 8 的整数倍。在程序清单 E.2 中，每个结构的所有成员只占用 13 位，但结构要求占用的位数为 8 的整数倍，这意味着需要填充一些位。在每个结构中，double 成员的对齐要求为 8 的整数倍，但在结构 thing1 和 thing2 中，成员的排列顺序不同，这导致 thing2 需要更多的内部填充，以便其边界处于正确的位置。

E.4　noexcept（C++11）

关键字 noexcept 用于指出函数不会引发异常。它也可用作运算符，判断操作数（表达式）是否可能引发异常；如果操作数可能引发异常，则返回 false，否则返回 true。例如，请看下面的声明：

```
int hilt(int);
int halt(int) noexcept;
```

表达式 noexcept(hilt) 的结果为 false，因为 hilt() 的声明未保证不会引发异常，但 noexcept(halt) 的结果为 true。

附录 F　模板类 string

本附录的技术性较强，但如果您只想了解模板类 string 的功能，可以将重点放在对各种 string 类方法的描述上。

string 类是基于下述模板定义的：

```
template<class charT, class traits = char_traits<charT>,
                class Allocator = allocator<charT> >
class basic_string {...};
```

其中，chatT 是存储在字符串中的类型；traits 参数是一个类，它定义了类型要被表示为字符串时，所必须具备的特征。例如，它必须有 length()方法，该方法返回被表示为 charT 数组的字符串的长度。这种数组结尾用 charT(0)值（广义的空值字符）表示。（表达式 charT(0)将 0 转换为 charT 类型。它可以像类型为 char 时那样为零，也可以是 charT 的一个构造函数创建的对象）。这个类还包含用于对值进行比较等操作的方法。Allocator 参数是用于处理字符串内存分配的类。默认的 allocator<char>模板按标准方式使用 new 和 delete。

有 4 种预定义的具体化：

```
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

上述具体化又使用下面的具体化：

```
char_traits<char>
allocator<char>
char_traits<char16_t>
allocator<char_16>
char_traits<char_32>
allocator<char_32>
char_traits<wchar_t>
allocator<wchar_t>
```

除 char 和 wchar_t 外，还可以通过定义 traits 类和使用 basic_string 模板来为其他一些类型创建一个 string 类。

F.1　13 种类型和一个常量

模板 basic_string 定义了几种类型，供以后定义方法时使用：

```
typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

traits 是对应于特定类型（如 char_traits<char>）的模板参数；traits_type 将成为该特定类型的 typedef。下述表示法意味着 char_type 是 traits 表示的类中定义的一个类型名：

```
typedef typename traits::char_type value_type;
```

关键字 typename 告诉编译器，表达式 trait::char_type 是一种类型。例如，对于 string 具体化，value_type 为 char。

size_type 与 size_of 的用法相似，只是它根据存储的类型返回字符串的长度。对于 string 具体化，将根据 char 返回字符串的长度，在这种情况下，size_type 与 size_of 等效。size_type 是一种无符号类型。

differrence_type 用于度量字符串中两个元素之间的距离（单位为元素的长度）。通常，它是底层类型 size_type 有符号版本。

对于 char 具体化来说，pointer 的类型为 char *，而 reference 的类型为 char &类型。然而，如果要为自己设计的类型创建具体化，则这些类型（pointer 和 reference）可以指向类，与基本指针和引用有相同的特征。

为将标准模板库（STL）算法用于字符串，该模板定义了一些迭代器类型：

```
typedef (models random access iterator)          iterator;
typedef (models random access iterator)          const_iterator;
typedef std::reverse_iterator<iterator>          reverse_iterator;
typedef std::reverse_iterator<const_iterator>    const_reverse_iterator;
```

该模板还定义了一个静态常量：

```
static const size_type npos = -1;
```

由于 size_type 是无符号的，因此将-1 赋给 npos 相当于将最大的无符号值赋给它，这个值比可能的最大数组索引大 1。

F.2　数据信息、构造函数及其他

可以根据其效果来描述构造函数。由于类的私有部分可能依赖于实现，因此可根据公用接口中可用的数据来描述这些效果。表 F.1 列出了一些方法，它们的返回值可用来描述构造函数和其他方法的效果。注意，其中的大部分术语来自 STL。

表 F.1　一些 string 数据方法

方　法	返　回　值
begin()	指向字符串第一个字符的迭代器
cbegin()	一个 const_iterator，指向字符串中的第一个字符（C++11）
end()	为超尾值的迭代器
cend()	为超尾值的 const_iterator（C++11）
rbegin()	为超尾值的反转迭代器
crbegin()	为超尾值的反转 const_iterator（C++11）
rend()	指向第一个字符的反转迭代器
crend()	指向第一个字符的反转 const_iterator（C++11）
size()	字符串中的元素数，等于 begin()到 end()之间的距离
length()	与 size()相同
capacity()	给字符串分配的元素数。这可能大于实际的字符数，capacity() – size()的值表示在字符串末尾附加多少字符后需要分配更多的内存
max_size()	字符串的最大长度
data()	一个指向数组第一个元素的 const charT*指针，其第一个 size()元素等于*this 控制的字符串中对应的元素，其下一个元素为 charT 类型的 charT(0)字符（字符串末尾标记）。当 string 对象本身被修改后，该指针可能无效
c_str()	一个指向数组第一个元素的 const charT*指针，其第一个 size()元素等于*this 控制的字符串中对应的元素，其下一个元素是 charT 类型的 charT（0）字符（字符串尾标识）。当 string 对象本身被修改后，该指针可能无效
get_allocator()	用于为字符串 object 分配内存的 allocator 对象的副本

请注意 begin()、rend()、data()和 c_str()之间的差别。它们都与字符串的第一个字符相关，但相关的方式不同。begin()和 rend()方法返回一个迭代器，正如第 16 章讨论的，这是一种广义指针。具体地说，begin()返回一个正向迭代器模型，而 rend()返回反转迭代器的一个副本。这两种方法都引用了 string 对象管理的字符串（由于 string 类使用动态内存分配，因此实际的 string 内容不一定位于对象中，因此，我们使用术语“管理”来描述对象和字符串之间的关系）。可以将返回迭代器的方法用于基于迭代器的 STL 算法中。例如，可以使用 STL reverse()函数来反转字符串的内容：

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

而 data()和 c_str()方法返回常规指针。另外，返回的指针将指向存储字符串字符的数组的第一个元素。

该数组可能（但不一定）是 string 对象管理的字符串的副本（string 对象采用的内部表示可以是数组，但不一定非得是数组）。由于返回的指针可能指向原始数据，而原始数据是 const，因此不能用它们来修改数据。另外，当字符串被修改后，将不能保证这些指针是有效的，这表明它们可能指向原始数据。data()和 c_str()的区别在于，c_str()指向的数组以空值字符（或与之等价的其他字符）结束，而 data()只是确保实际的字符串字符是存在的。因此，c_str()方法期望接受一个 C-风格字符串参数：

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

同样，data()和 size()可用作这种函数的参数，即接受指向数组元素的指针和表示要处理的元素数目的值：

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

C++实现可能将 string 对象的字符串表示为动态分配的 C-风格字符串，并使用 char*指针来实现正向迭代器。在这种情况下，实现可能让 begin()、data()和 c_str()都返回同样的指针，但返回指向 3 个不同的数据对象的引用也是合法的（虽然更复杂）。

在 C++11 中，模板类 basic_string 有 11 个构造函数（在 C++98 中只有 6 个）和一个析构函数：

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const basic_string& str);
basic_string(const basic_string& str, const Allocator&);
basic_string(const basic_string& str, size_type pos,
    size_type n = npos, const Allocator& a = Allocator());
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string&& str, const Allocator&);
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
    const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
~basic_string();
```

有些新增的构造函数以不同的方式处理参数。例如，C++98 包含如下复制构造函数：

```
basic_string(const basic_string& str, size_type pos = 0,
    size_type n = npos, const Allocator& a = Allocator());
```

而 C++11 用三个构造函数取代了它——上述列表中的第 2～4 个，这提高了编码效率。真正新增的只有移动构造函数（使用右值引用的构造函数，这在第 18 章讨论过）以及使用 initializer_list 参数的构造函数。

注意到大多数构造函数构造函数都有一个下面这样的参数：

```
const Allocator& a = Allocator()
```

Allocator 是用于管理内存的 allocator 类的模板参数名；Allocator()是这个类的默认构造函数。因此，在默认情况下，构造函数将使用 allocator 对象的默认版本，但它们使得能够选择使用 allocator 对象的其他版本。下面分别介绍这些构造函数。

F.2.1 默认构造函数

默认构造函数的原型如下：

```
explicit basic_string(const Allocator& a = Allocator());
```

通常，接受 allocator 类的默认参数，并使用该构造函数来创建空字符串：

```
string bean;
wstring theory;
```

调用该默认构造函数后，将存在下面的关系：

- data()方法返回一个非空指针，可以将该指针加上 0；
- size()方法返回 0；
- capacity()的返回值不确定。

将 data()返回的值赋给指针 str 后，第一个条件意味着 str + 0 是有效的。

F.2.2 使用 C-风格字符串的构造函数

使用 C-风格字符串的构造函数让您能够将 string 对象初始化为一个 C-风格字符串；从更普遍的意义上看，它使得能够将 charT 具体化初始化为一个 charT 数组：

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

为确定要复制的字符数，该构造函数将 traits::length()方法用于 s 指向的数组（s 不能为空指针）。例如，下面的语句使用指定的字符串来初始化 toast 对象：

```
string toast("Here's looking at you, kid.");
```

char 类型的 traits::length()方法将使用空值字符来确定要复制多少个字符。

该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指针，该指针指向数组 s 的一个副本的第一个元素；
- size()方法返回的值等于 trains::length()的值；
- capacity()方法返回一个至少等于 size()的值。

F.2.3　使用部分 C-风格字符串的构造函数

使用部分 C-风格字符串的构造函数让您能够使用 C-风格字符串的一部分来初始化 string 对象；从更广泛的意义上说，该构造函数使得能够使用 charT 数组的一部分来初始化 charT 具体化：

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

该构造函数将 s 指向的数组中的 n 个字符复制到构造的对象中。请注意，如果 s 包含的字符数少于 n，则复制过程将不会停止。如果 n 大于 s 的长度，该构造函数将把字符串后面的内存内容解释为 charT 类型的数据。

该构造函数要求 s 不能是空值指针，同时 n<npos（npos 是一个静态类常量，它是字符串可能包含的最大元素数目）。如果 n 等于 npos，该构造函数将引发一个 out_of_range 异常（由于 n 的类型为 size_type，而 npos 是 size_type 的最大值，因此 n 不能大于 npos）；否则，在该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指针，该指针指向数组 s 的副本的第一个元素；
- size()方法返回 n；
- capacity()方法返回一个至少等于 size()的值。

F.2.4　使用左值引用的构造函数

复制构造函数类似于下面这样：

```
basic_string(const basic_string& str);
```

它使用一个 string 参数初始化一个新的 string 对象：

```
string mel("I'm ok!");
string ida(mel);
```

其中，ida 将是 mel 管理的字符串副本。

下一个构造函数要求您指定一个分配器：

```
basic_string(const basic_string& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data()方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data()指向的；
- size()方法返回 str.size()的值；
- capacity()方法返回一个至少等于 size()的值。

再下一个构造函数让您能够指定多项内容：

```
basic_string(const basic_string& str, size_type pos, size_type n = npos,
    const Allocator& a = Allocator());
```

第二个参数（pos）指定了源字符串中的位置，将从这个位置开始进行复制：

```
string att("Telephone home.");
string et(att, 4);
```

位置编号从 0 开始，因此，位置 4 是字符 p。所以，et 被初始化为“phone home”。

第 3 个参数 n 是可选的，它指定要复制的最大字符数目，因此下面的语句将 pt 初始化为字符串“phone”：

```
string att("Telephone home.");
string pt(att, 4, 5);
```

然而，该构造函数不能跨越源字符串的结尾，例如，下面的语句将在复制句点后停止：

```
string pt(att, 4, 200)
```

因此，该构造函数实际复制的字符数量等于 n 和 str.size()-pos 中较小的一个。

该构造函数要求 pos 不大于 str.size()，也就是说，被复制的初始位置必须位于源字符串中。如果情况并

非如此，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，copy_len 将是 n 和 str.size()-pos 中较小的一个，并存在下面的关系：

- data()方法返回一个指向字符串的指针，该字符串包含 copy_len 个元素，这些元素是从 str 的 pos 位置开始复制而得到的；
- size()方法返回 copy_len；
- capacity()方法返回一个不小于 size()的值。

F.2.5 使用右值引用的构造函数（C++11）

C++11 给 string 类添加了移动语义。正如第 18 章介绍的，这意味着添加一个移动构造函数，它使用右值引用而不是左值引用：

```
basic_string(basic_string&& str) noexcept;
```

在实参为临时对象时将调用这个构造函数：

```
string one("din");     // C-style string constructor
string two(one);       // copy constructor - one is an lvalue
string three(one+two); // move constructor, sum is an rvalue
```

正如第 18 章讨论的，three 将获取 operator + ()创建的对象的所有权，而不是将该对象复制给 three，再销毁原始对象。

第二个使用右值引用的构造函数让您能够指定分配器：

```
basic_string(const basic_string&& str, const Allocator&);
```

调用这两个构造函数中的任何一个后，将存在如下关系：

- data()方法返回一个指针，该指针指向分配的数组副本，该数组的第一个元素是 str.data()指向的；
- size()方法返回 str.size()的值；
- capacity()方法返回一个至少等于 size()的值。

F.2.6 使用一个字符的 n 个副本的构造函数

使用一个字符的 n 个副本的构造函数创建一个由 n 个 c 组成的 string 对象：

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

该构造函数要求 n<npos。如果 n 等于 npos，该构造函数将引发 out_of_range 异常；否则，该构造函数被调用后，将存在下面的关系：

- data()方法返回一个指向字符串第一个元素的指针，该字符串由 n 个元素组成，其中每个元素的值都为 c；
- size()方法返回 n；
- capacity()方法返回不小于 size()的值。

F.2.7 使用区间的构造函数

使用区间的构造函数使用一个用迭代器定义的、STL-风格的区间：

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
                const Allocator& a = Allocator());
```

begin 迭代器指向源字符串中要复制的第一个元素，end 指向要复制的最后一个元素的后面。

这种构造函数可用于数组、字符串或 STL 容器：

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
    input.push_back(ch);
string str_input(input.begin(), input.end());
```

在第一种用法中，InputIterator 的类型为 const char *；在第二种用法中，InputIterator 的类型为 vector<char>::iterator。

调用该构造函数后，将存在下面的关系：

- data()方法返回一个指向字符串的第一个元素的指针，该字符串是通过复制区间[begin，end）中的

元素得到的；

- size()方法返回 begin 到 end 之间的距离（度量距离时，使用的单位为对迭代器解除引用得到的数据类型的长度）；
- capacity()方法返回一个不小于 size()的值。

F.2.8　使用初始化列表的构造函数（C++11）

这个构造函数接受一个 initializer_list<charT>参数：

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

可将一个用大括号括起的字符列表作为参数：

```
string slow({'s', 'n', 'a', 'i', 'l'});
```

这并非初始化 string 的最方便方式，但让 string 的接口类似于 STL 容器类。

initializer_list 类包含成员函数 begin()和 end()，调用该构造函数的影响与调用使用区间的构造函数相同：

```
basic_string(il.begin(), il.end(), a);
```

F.2.9　内存杂记

有些方法用于处理内存，如清除内存的内容、调整字符串长度或容量。表 F.2 列出了一些与内存相关的方法。

表 F.2　一些与内存有关的方法

方　法	作　用
void resize(size_type n)	如果 n>npos，将引发 out_of_range 异常；否则，将字符串的长度改为 n，如果 n<size()，则截短字符串，如果 n>size()，则使用 charT(0)中的字符填充字符串
void resize(size_type n, charT c)	如果 n>npos，将引发 out_of_range 异常；否则，将字符串长度改为 n，如果 n<size()，则截短字符串，如果 n>size()，则使用字符 c 填充字符串
void reserve(size_type res_arg = 0)	将 capacity()设置为大于或等于 res_arg。由于这将重新分配字符串，因此以前的引用、迭代器和指针将无效
void shrink_to_fit()	请求让 capacity()的值与 size()相同，这是 C++11 新增的
void clear() noexcept	删除字符串中所有的字符
bool empty()const noexcept	如果 size()==0，则返回 true

F.3　字符串存取

有 4 种方法可以访问各个字符，其中两种方法使用[ ]运算符，另外两种方法使用 at()方法：

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

第一个 operator[ ]()方法使得能够使用数组表示法来访问字符串的元素，可用于检索或更改值。第二个 operator[ ]()方法可用于 const 对象，但只能用于检索值：

```
string word("tack");
cout << word[0]; // display the t
word[3] = 't';    // overwrite the k with a t
const ward("garlic");
cout << ward[2]; // display the r
```

at()方法提供了相似的访问功能，只是索引是通过函数参数提供的：

```
string word("tack");
cout << word.at(0); // display the t
```

差别在于（除语法差别外）：at()方法执行边界检查，如果 pos>=size()，将引发 out_of_range 异常。pos 的类型为 size_type，是无符号的，因此 pos 的值不能为负；而 operator[ ]()方法不进行边界检查，因此，如果 pos>=size()，则其行为将是不确定的（如果 pos= =size()，const 版本将返回空值字符的等价物）。

因此，可以在安全性（使用 at()检测异常）和执行速度（使用数组表示）之间进行选择。

还有一个这样的函数，它返回原始字符串的子字符串：

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

它返回一个字符串——这是从 pos 开始，复制 n 个字符（或到字符串尾部）得到的。例如，下面的代码将 pet 初始化为“donkey”：

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

C++11 新增了如下四个存取方法：

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

其中 front()方法访问 string 的第一个元素，相当于 operator[] (0)；back()方法访问 string 的最后一个元素，相当于 operator[] (size() - 1)。

F.4 基本赋值

在 C++11 中，有 5 个重载的赋值方法，在 C++98 的基础上增加了两个：

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(basic_string&& str) noexcept; // C++11
basic_string& operator=(initializer_list<charT>);      // C++11
```

第一个方法将一个 string 对象赋给另一个；第二个方法将 C-风格字符串赋给 string 对象；第三个方法将一个字符赋给 string 对象；第四个方法使用移动语义，将一个右值 string 对象赋给一个 string 对象；第五个方法让您能够使用初始化列表进行赋值。因此，下面的操作都是可能的：

```
string name("George Wash");
string pres, veep, source, join, awkward;
pres = name;
veep = "Road Runner";
source = 'X';
join = name + source; // now with move semantics!
awkward = {'C','l','o','u','s','e','a','u'};
```

F.5 字符串搜索

string 类提供了 6 种搜索函数，其中每个函数都有 4 个原型。下面简要地介绍它们。

F.5.1 find()系列

在 C++11 中，find()的原型如下：

```
size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const noexcept;
```

第一个返回 str 在调用对象中第一次出现时的起始位置。搜索从 pos 开始，如果没有找到子字符串，将返回 npos。

下面的代码在一个字符串中查找字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);           // sets loc1 to 1
size_type loc2 = longer.find(shorter, loc1 + 1); // sets loc2 to 16
```

由于第二条搜索语句从位置 2 开始（That 中的 a），因此它找到的第一个 hat 位于字符串尾部。要测试是否失败，可使用 string::npos 值：

```
if (loc1 == string::npos)
    cout << "Not found\n";
```

第二个方法完成同样的工作，但它使用字符数组而不是 string 对象作为子字符串：

```
size_type loc3 = longer.find("is"); //sets loc3 to 5
```

第三个方法完成相同的工作，但它只使用字符串 s 的前 n 个字符。这与使用 basic_string（const charT * s，size_type n）构造函数，然后将得到的对象用作第一种格式的 find()的 string 参数的效果完全相同。例如，下面的代码搜索子字符串“fun”：

```
size_type loc4 = longer.find("funds", 3); //sets loc4 to 10
```

第四个方法的功能与第一个相同，但它使用一个字符而不是 string 对象作为子字符串：

```
size_type loc5 = longer.find('a'); //sets loc5 to 2
```

F.5.2　rfind()系列

rfind()方法的原型如下：

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;
```

这些方法与相应 find()方法的工作方式相似，但它们搜索字符串最后一次出现的位置，该位置位于 pos 之前（包括 pos）。如果没有找到，该方法将返回 npos。

下面的代码从字符串末尾开始查找子字符串“hat”的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);           // sets loc1 to 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // sets loc2 to 1
```

F.5.3　find_first_of()系列

find_first_of()方法的原型如下：

```
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find()方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符首次出现的位置。

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter); // sets loc1 to 10
size_type loc2 = longer.find_first_of("fat");   // sets loc2 to 2
```

在 longer 中，首次出现的 fluke 中的字符是 funny 中的 f，而首次出现的 fat 中的字符是 That 中的 a。

F.5.4　find_last_of()系列

find_last_of()方法的原型如下：

```
size_type find_last_of (const basic_string& str,
                        size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 rfind()方法的工作方式相似，但它们不是搜索整个子字符串，而是搜索子字符串中的字符出现的最后位置。

下面的代码在一个字符串中查找字符串“hat”和“any”中字母最后出现的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter); // sets loc1 to 18
size_type loc2 = longer.find_last_of("any");   // sets loc2 to 17
```

在 longer 中，最后出现的 hat 中的字符是 hat 中的 t，而最后出现的 any 中的字符是 hat 中的 a。

F.5.5　find_first_not_of()系列

find_first_not_of()方法的原型如下：

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos,
                             size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
```

这些方法与对应 find_first_of()方法的工作方式相似，但它们搜索第一个不位于子字符串中的字符。

下面的代码在字符串中查找第一个没有出现在“This”和“Thatch”中的字母：

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);  // sets loc1 to 2
size_type loc2 = longer.find_first_not_of("Thatch"); // sets loc2 to 4
```

在 longer 中，That 中的 a 是第一个在 This 中没有出现的字符，而字符串 longer 中的第一个空格是第一个没有在 Thatch 中出现的字符。

F.5.6 find_last_not_of()系列

find_last_not_of()方法的原型如下：

```
size_type find_last_not_of (const basic_string& str,
                            size_type pos = npos) const noexcept;
size_type find_last_not_of (const charT* s, size_type pos,
                            size_type n) const;
size_type find_last_not_of (const charT* s,
                            size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;
```

这些方法与对应 find_last_of()方法的工作方式相似，但它们搜索的是最后一个没有在子字符串中出现的字符。

下面的代码在字符串中查找最后一个没有出现在“That.”中的字符：

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);     // sets loc1 to 15
size_type loc2 = longer.find_last_not_of(shorter, 10); // sets loc2 to 10
```

在 longer 中，最后的空格是最后一个没有出现在 shorter 中的字符，而 longer 字符串中的 f 是搜索到位置 10 时，最后一个没有出现在 shorter 中的字符。

F.6 比较方法和函数

string 类提供了用于比较 2 个字符串的方法和函数。下面是方法的原型：

```
int compare(const basic_string& str) const noexcept;

int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str,
size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1, const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2 ) const;
```

这些方法使用 traits::compare()方法，后者是为用于字符串的字符类型定义的。如果根据 traits::compare()提供的顺序，第一个字符串位于第二个字符串之前，则第一个方法将返回一个小于 0 的值；如果这两个字符串相同，则它将返回 0；如果第一个字符串位于第二个字符串的后面，则它将返回一个大于 0 的值。如果较长的字符串的前半部分与较短的字符串相同，则较短的字符串将位于较长的字符串之前。

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 is < 0
int a12 = s1.compare(s2); // a12 is > 0
```

第二个方法与第一个方法相似，但它进行比较时，只使用第一个字符串中从位置 pos1 开始的 n1 个字符。

下面的示例将字符串 s1 的前 4 个字符同字符串 s2 进行比较：

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 is 0
```

第三个方法与第一个方法相似，但它使用第一个字符串中从 pos1 位置开始的 n1 个字符和第二个字符串中从 pos2 位置开始的 n2 个字符进行比较。例如，下面的语句将对 stout 中的 out 和 about 中的 out 进行比较：

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3); // a3 is 0
```

第四个方法与第一个方法相似，但它将一个字符数组而不是 string 对象作为第二个字符串。

第五和六个方法与第三个方法相似，但将一个字符串数组而不是 string 对象作为第二个字符串。

非成员比较函数是重载的关系运算符：

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

每一个运算符都被重载，使之将 string 对象与 string 对象进行比较、将 string 对象与 C-风格字符串进行比较、将 C-风格字符串与 string 对象进行比较。它们都是根据 compare()方法定义的，因此提供了一种在表示方面更为方便的比较方式。

F.7　字符串修改方法

string 类提供了多个用于修改字符串的方法，其中绝大多数都拥有大量的重载版本，因此可用于 string 对象、字符串数组、单个字符和迭代器区间。

F.7.1　用于追加和相加的方法

可以使用重载的+ =运算符或 append()方法将一个字符串追加到另一个字符串的后面。如果得到的字符串长于最大字符串长度，将引发 length_error 异常。+=运算符使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面：

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

append()方法也使得能够将 string 对象、字符串数组或单个字符追加到 string 对象的后面。此外，通过指定初始位置和追加的字符数，或者通过指定区间，还可以追加 string 对象的一部分。通过指定要使用字符串中的多少个字符，可以追加字符串的一部分。追加字符的版本使得能够指定要复制该字符的多少个实例。下面是各种 append()方法的原型：

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c); // append n copies of c
void push_back(charT c);                        // appends 1 copy of c
```

下面是几个示例：

```
string test("The");
test.append("ory"); // test is "Theory"
test.append(3,'!'); // test is "Theory!!!"
```

operator+()函数被重载，以便能够拼接字符串。该重载函数不修改字符串，而是创建一个新的字符串，该字符串是通过将第二个字符串追加到第一个字符串后面得到的。加法函数不是成员函数，它们使得能够将 string 对象和 string 对象、string 对象和字符串数组、字符串数组和 string 对象、string 对象和字符以及字符和 string 对象相加。下面是一些例子：

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce"; // st3 is "reduce"
string st4 = 't' + st2;   // st4 is "train"
string st5 = st1 + st2;   // st5 is "redrain"
```

F.7.2　其他赋值方法

除了基本的赋值运算符外，string 类还提供了 assign()方法，该方法使得能够将整个字符串、字符串的一部分或由相同字符组成的字符序列赋给 string 对象。下面是各种 assign()方法的原型：

```
basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str) noexcept; // C++11
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
```

```
basic_string& assign(size_type n, charT c);    // assign n copies of c
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>); // C++11
```

下面是几个例子：

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5); // test is "et tu"
test.assign(6, '#");       // test is "######"
```

接受右值引用作为参数的 assign()方法是 C++11 新增的，它支持移动语义；另一个新增的 assign()方法让您能够将 initializer_list 赋给 string 对象。

F.7.3 插入方法

insert()方法使得能够将 string 对象、字符串数组或几个字符插入到 string 对象中。这个方法与 append()方法相似，但它还接受另一个指定插入位置的参数，该参数可以是位置，也可以是迭代器。数据将被插入到插入点的前面。有几种方法返回一个指向得到的字符串的引用。如果 pos1 超过了目标字符串结尾，或者 pos2 超过了要插入的字符串结尾，该方法将引发 out_of_range 异常。如果得到的字符串长于最大长度，该方法将引发 length_error 异常。下面是各种 insert()方法的原型：

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>); // C++11
```

例如，下面的代码将字符串“former”字符串插入到“The banker.”中 b 的前面：

```
string st3("The banker.");
st3.insert(4, "former ");
```

而下面的代码将字符串“ waltzed”（不包括!，它是第 9 个字符）插入到“The former banker.”末尾的句号之前：

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

F.7.4 清除方法

erase()方法从字符串中删除字符，其原型如下：

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator position);
iterator erase(const_iterator first, iterator last);
void pop_back();
```

第一种格式将从 pos 位置开始，删除 n 个字符或删除到字符串尾。第二种格式删除迭代器位置引用的字符，并返回指向下一个元素的迭代器；如果后面没有其他元素，则返回 end()。第三种格式删除区间[first，last）中的字符，即从 first（包括）到 last（不包括）之间的字符；它返回最后一个迭代器，该迭代器指向最后一个被删除的元素后面的一个元素。最后，方法 pop_back()删除字符串中的最后一个字符。

F.7.5 替换方法

各种 replace()方法都指定了要替换的字符串部分和用于替换的内容。可以使用初始位置和字符数目或迭代器区间来指定要替换的部分。替换内容可以是 string 对象、字符串数组，也可以是特定字符的多个实例。对于用于替换的 string 对象和数组，可以通过指定特定部分（使用位置和计数或只使用计数）或迭代器区间做进一步的修改。下面是各种 replace()方法的原型：

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
```

```
                        const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                        const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2,
                        const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                        size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                          InputIterator j1, InputIterator j2);
basic_string& replace(const_iteraor i1, const_iteator i2,
                        initializer)list<charT> il);
```

下面是一个例子：

```
string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); // replace right with left
```

注意，您可以使用 find()来找出要在 replace()中使用的位置：

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
    s3.replace(pos, s1.size(), s2);
```

上述代码将 old 替换为 mature。

F.7.6　其他修改方法：copy()和 swap()

copy()方法将 string 对象或其中的一部分复制到指定的字符串数组中：

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

其中，s 指向目标数组，n 是要复制的字符数，pos 指出从 string 对象的什么位置开始复制。复制将一直进行下去，直到复制了 n 个字符或到达 string 对象的最后一个字符。函数返回复制的字符数，该方法不追加空值字符，同时由程序员负责检查数组的长度是否足够存储复制的内容。

警告：copy()方法不追加空值字符，也不检查目标数组的长度是否足够。

swap()方法使用一个时间恒定的算法来交换两个 string 对象的内容：

```
void swap(basic_string& str);
```

F.8　输出和输入

string 类重载了<<运算符来显示 string 对象，该运算符返回 istream 对象的引用，因此可以拼接输出：

```
string claim("The string class has many features.");
cout << claim << endl;
```

string 类重载了>>运算符，使得能够将输入读入到字符串中：

```
string who;
cin >> who;
```

到达文件尾、读取字符串允许的最大字符数或遇到空白字符后，输入将终止（空白的定义取决于字符集以及 charT 表示的类型）。

有两个 getline()函数，第一个的原型如下：

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
            basic_string<charT,traits,Allocator>& str, charT delim);
```

这个函数将输入流 is 中的字符读入到字符串 str 中，直到遇到定界字符 delim、到达文件尾或者到达字符串的最大长度。delim 字符将被读取（从输入流中删除），但不被存储。第二个版本没有第三个参数，同时使用换行符（或其广义形式），而不是 delim：

```
string str1, str2;
getline(cin, str1);      // read to end-of-line
getline(cin, str2, '.'); // read to period
```

附录 G　标准模板库方法和函数

标准模板库（STL）旨在提供通用算法的高效实现，它通过通用函数（可用于满足特定算法要求的任何容器）和方法（可用于特定容器类实例）来表达这些算法。本附录假设您对 STL 有一定的了解（如通过阅读第 16 章）。例如，本章假设您了解迭代器和构造函数。

G.1　STL 和 C++11

C++11 对 C++语言做了大量修改，本书无法全面介绍，同样 C++11 对 STL 也做了大量修改，本附录无法全面介绍。然而，可以对新增的内容做一总结。

C++11 给 STL 新增了多个元素。首先，它新增了多个容器；其次，给旧容器新增了多项功能；第三，在算法系列中新增了一些模板函数。本附录介绍了所有这些变化，对前两类变化有大致了解将很有帮助。

G.1.1　新增的容器

C++11 新增了如下容器：array、forward_list、unordered_st 以及无序关联容器 unordered_multiset、unordered_map 和 unordered_multimap。

array 容器一旦声明，其长度就是固定的，它使用静态（栈）内存，而不是动态分配的内存。提供它旨在替代数组；array 受到的限制比 vector 多，但效率更高。

容器 list 是一种双向链表，除两端的节点外，每个节点都链接到它前面和后面的节点。forward_list 是一种单向链表，除最后一个节点外，每个节点都链接到下一个节点。相对于 list，它更紧凑，但受到的限制更多。

与 set 和其他关联容器一样，无序关联容器让您能够使用键快速检索数据，差别在于关联容器使用的底层数据结构为树，而无序关联容器使用的是哈希表。

G.1.2　对 C++98 容器所做的修改

C++11 对容器类的方法做了三项主要修改。

首先，新增的右值引用使得能够给容器提供移动语义（参见第 18 章）。因此，STL 现在给容器提供了移动构造函数和移动赋值运算符，这些方法将右值引用作为参数。

其次，由于新增了模板类 initilizer_list（参见第 18 章），因此新增了将 initilizer_list 作为参数的构造函数和赋值运算符。这使得可以编写类似于下面的代码：

```
vector<int> vi{100, 99, 97, 98};
vi = {96, 99, 94, 95, 102};
```

第三，新增的可变参数模板（variadic template）和函数参数包（parameter pack）使得可以提供就地创建（emplacement）方法。这意味着什么呢？与移动语义一样，就地创建旨在提高效率。请看下面的代码段：

```
class Items
{
    double x;
    double y;
    int m;
public:
    Items();                                    // #1
    Items (double xx, double yy, int mm); // #2
    ...
};
...
vector<Items> vt(10);
```

```
...
vt.push_back(Items(8.2, 2.8, 3)); //
```

调用 insert()将导致内存分配函数在 vt 末尾创建一个默认 Items 对象。接下来，构造函数 Items()创建一个临时 Items 对象，该对象被复制到 vt 的开头，然后被删除。在 C++11 中，您可以这样做：

```
vi.emplace_back(8.2, 2.8, 3);
```

方法 emplace_back()是一个可变参数模板，将一个函数参数包作为参数：

```
template <class... Args> void emplace_back(Args&&... args);
```

上述三个实参（8.2、2.8 和 3）将被封装到参数 args 中。参数 args 被传递给内存分配函数，而内存分配函数将其展开，并使用接受三个参数的 Items 构造函数（#2），而不是默认构造函数（#1）。也就是说，它使用 Items(args…)，这里将展开为 Items(8.2, 2.8, 3)。因此，将在矢量中就地创建所需的对象，而不是创建一个临时对象，再将其复制到矢量中。

STL 在多个就地创建方法中使用了这种技术。

G.2　大部分容器都有的成员

所有容器都定义了表 G.1 列出的类型。在这个表中，x 为容器类型，如 vector<int>；T 为存储在容器中的类型，如 int。表 G.1 中的示例阐明了含义

表 G.1　　为所有容器定义的类型

类　　型	值
x::value-type	T，元素类型
x::reference	T &
x::const_reference	const T &
x::iterator	指向 T 的迭代器类型，行为与 T*相似
x::const_iterator	指向 const T 的迭代器类型，行为与 const T *相似
x::different_type	用于表示两个迭代器之间距离的符号整型，如两个指针的差
x::size_type	无符号整型 size_type 可以表示数据对象的长度、元素数目和下标

类定义使用 typedef 定义这些成员。可以使用这些类型来声明适当的变量。例如，下面的代码使用迂回的方式，将由 string 对象组成的矢量中的第一个“bonus”替换为“bogus”，以演示如何使用成员类型来声明变量。

```
using namespace std;
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
    input.push_back(temp);
vector<string>::iterator want=
    find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
    vector<string>::reference r = *want;
    r = "bogus";
}
```

上述代码使 r 成为一个指向（want 指向的）input 中元素的引用。同样，继续前面的例子，可以编写下面这样的代码：

```
vector<string>::value_type s1 = input[0]; // s1 is type string
vector<string>::reference s2 = input[1]; // s2 is type string &
```

这将导致 s1 为一个新 string 对象，它是 input[0]的拷贝；而 s2 为指向 input[1]的引用。在这个例子中，由于已经知道模板是基于 string 类型的，因此编写下面的等效代码将更简单：

```
string s1 = input[0];   // s1 is type string
string & s2 = input[1]; // s2 is type string &
```

然而，还可以在更通用的代码中使用表 G.1 中较精致（其中容器和元素的类型是通用的）的类型。例如，假设希望 min()函数将一个指向容器的引用作为参数，并返回容器中最小的项目。这假设为用于实例化模板的值类型定义了<运算符，而不想使用 STL min_element()算法，这种算法使用迭代器接口。由于参数可能是 vector<int>、list<strint>或 deque<double>，因此需要使用带模板参数（如 Bag）的模板来表示容器

（也就是说，Bag 是一个模板类型，可能被实例化为 vector<int>、list<string>或其他一些容器类型）。因此，函数的参数类型应为 const Bag & b。返回类型是什么呢？应为容器的值类型，即 Bag::value_type。然而，在这种情况下，Bag 只是一个模板参数，编译器无法知道 value_type 成员实际上是一种类型。但可以使用 typename 关键字来指出，类成员是 typedef：

```
vector<string>::value_type st; // vector<string> a defined class
typename Bag::value_type m;     // Bag an as yet undefined type
```

对于上述第一个定义，编译器能够访问 vector 模板定义，该定义指出，value_type 是一个 typedef；对于第二个定义，typename 关键字指出，无论 Bag 将会是什么，Bag::value-type 都将是类型的名称。这些考虑因素导致了下面的定义：

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
    typename Bag::const_iterator it;
    typename Bag::value_type m = *b.begin();
    for (it = b.begin(); it != b.end(); ++it)
        if (*it < m)
            m = *it;
    return m;
}
```

这样，便可以这样使用该模板函数：

```
vector<int> temperatures;
// input temperature values into the vector
int coldest = min(temperatures);
```

temperatures 参数将使得 Bag 被谓词为 vector<int>，而 typename Bag::value-type 被谓词为 vector <int>::value_type，进而为 int。

所有的容器都还可以包含表 G.2 列出的成员函数或操作。其中，X 是容器类型，如 vector<int>；而 T 是存储在容器中的类型，如 int。另外，a 和 b 是类型为 X 的值；u 是标识符；r 是类型为 X 的非 const 值；rv 是类型为 X 的非 const 右值，而移动操作是 C++11 新增的。

表 G.2　　为所有容器定义的操作

操　　作	描　　述
X u;	创建一个名为 u 的空对象
X()	创建一个空对象
X(a)	创建对象 x 的拷贝
X u(a)	u 是 a 的拷贝（复制构造函数）
X u = a;	u 是 a 的拷贝（复制构造函数）
r = a	r 等于 a 的值（复制赋值）
X u(rv)	u 等于 rv 的原始值（移动构造函数）
X u = rv	u 等于 rv 的原始值（移动构造函数）
a = rv	u 等于 rv 的原始值（移动赋值）
(&a)->~X()	对 a 的每个元素执行析构函数
begin()	返回一个指向第一个元素的迭代器
end()	返回一个指向超尾的迭代器
cbegin()	返回一个指向第一个元素的 const 迭代器
cend()	返回一个指向超尾的 const 迭代器
size()	返回元素数目
maxsize()	返回容器的最大可能长度
empty()	如果容器为空，则返回 true
swap()	交换两个容器的内容
= =	如果两个容器的长度相同、包含的元素相同且元素排列的顺序相同，则返回 true
! =	a!=b 返回!(a= =b)

使用双向或随机迭代器的容器（vector、list、deque、array、set 和 map）是可反转的，它们提供了表 G.3 所示的方法。

表 G.3　为可反转容器定义的类型和操作

操　作	描　述
X::reverse_iterator	指向类型 T 的反向迭代器
X::const_reverse_iterator	指向类型 T 的 const 反向迭代器
a.rbegin()	返回一个反向迭代器，指向 a 的超尾
a.rend()	返回一个指向 a 的开头的反向迭代器
a.crbegin()	返回一个 const 反向迭代器，指向 a 的超尾
a.crend()	返回一个指向 a 的开头的 const 反向迭代器

无序集合（set）和无序映射（map）无需支持表 G.4 所示的可选容器操作，但其他容器必须支持。

表 G.4　可选的容器操作

操　作	描　述
<	如果 a 按词典顺序排在 b 之前，则 a<b 返回 true
>	a>b 返回 b<a
<=	a<=b 返回!(a>b)
>=	a>=b 返回!(a<b)

容器的>运算符假设已经为值类型定义了>运算符。词典比较是一种广义的按字母顺序排序，它逐元素地比较两个容器，直到两个容器中对应的元素相同时为止。在这种情况下，元素对的顺序将决定容器的顺序。例如，如果两个容器的前 10 个元素都相同，但第一个容器的第 11 个元素比第二个容器的第 11 个元素小，则第一个容器将排在第二个容器之前。如果两个容器中的元素一直相同，直到其中一个容器中的元素用完，则较短的容器将排在较长的容器之前。

G.3　序列容器的其他成员

模板类 vector、forward_list、list、deque 和 array 都是序列容器，它们都前面列出的方法，但 forward_list 不是可反转的，不支持表 G.3 所示的方法。序列容器以线性顺序存储一组类型相同的值。如果序列包含的元素数是固定的，通常选择使用 array；否则，应首先考虑使用 vector，它让 array 的随机存取功能以及添加和删除元素的功能于一身。然而，如果经常需要在序列中间添加元素，应考虑使用 list 或 forward_list。如果添加和删除操作主要是在序列两端进行的，应考虑使用 deque。

array 对象的长度是固定的，因此无法使用众多序列方法。表 G.5 列出除 array 外的序列容器可用的其他方法（forward_list 的 resize()方法的定义稍有不同）。同样，其中 X 是容器类型，如 vector<int>；T 是存储在容器中的类型，如 int；a 是类型为 X 的值；t 是类型为 x::value_type 的左值或 const 右值；i 和 j 是输入迭代器；[i, j]是有效的区间；il 是类型为 initilizer_list<value_type>的对象；p 是指向 a 的有效 const 迭代器；q 是可解除引用的有效 const 迭代器；[q1, q2]是有效的 const 迭代器区间；n 是 x::size_type 类型的整数；Args 是模板参数包，而 args 是形式为 Args&&的函数参数包。

表 G.5　为序列容器定义的其他操作

操　作	描　述
X(n, t)	创建一个序列容器，它包含 t 的 n 个拷贝
X a(n, t)	创建一个名为 a 的序列容器，它包含 t 的 n 个拷贝
X(i, j)	使用区间[i, j] 内的值创建一个序列容器
X a(i, j)	使用区间[i, j) 内的值创建一个名为 a 的序列容器
X(il)	创建一个序列容器，并将其初始化为 il 的内容
a = il;	将 il 的值复制到 a 中
a.emplace(p, args);	在 p 前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数

续表

操　　作	描　　述
a.insert(p, t)	在 p 之前插入 t 的拷贝，并返回指向该拷贝的迭代器。T 的默认值为 T()，即在没有显式初始化时，用于 T 类型的值
a.insert(p, rv)	在 p 之前插入 rv 的拷贝，并返回指向该拷贝的迭代器；可能使用移动语义
a.insert(p, n, t)	在 p 之前插入 t 的 n 个拷贝
a.insert(p, i, j)	在 p 之前插入[i, j)区间内元素的拷贝
a.insert(p, il)	等价于 a.insert(p, il.begin(), il.end())
a.resize(n)	如果 n > a.size()，则在 a.end()之前插入 n - a.size()个元素；用于新元素的值为没有显式初始化时，用于 T 类型的值；如果 n < a.size()，则删除第 n 个元素之后的所有元素
a.resize(n, t)	如果 n > a.size()，则在 a.end()之前插入 t 的 n - a.size()个拷贝；如果 n<a.size()，则删除第 n 个元素之后的所有元素
a.assign(i, j)	使用区间[i，j)内的元素拷贝替换 a 当前的内容
a.assign(n, t)	使用 t 的 n 个拷贝替换 a 的当前内容。t 的默认值为 T()，即在没有显式初始化时，用于 T 类型的值
a.assign(il)	等价于 a.assign(il.begin(), il.end())
a.erase(q)	删除 q 指向的元素；返回一个指向 q 后面的元素的迭代器
a.erase(q1, q2)	删除区间[q1, q2]内的元素；返回一个迭代器，该迭代器指向 q2 原来指向的元素
a.clear()	与 erase(a.begin(), a.end()) 等效
a.front()	返回*a.begin()（第一个元素）

表 G.6 列出了一些序列类（vector、forward_list、list 和 deque）都有的方法。

表 G.6　　为某些序列定义的操作

操　　作	描　　述	容　　器
a.back()	返回*a.end()（最后一个元素）	vector、list、deque
a.push_back(t)	将 t 插入到 a.end()前面	vector、list、deque
a.push_back(rv)	将 rv 插入到 a.end()前面；可能使用移动语义	vector、list、deque
a.pop_back()	删除最后一个元素	vector、list、deque
a.emplace_back(args)	追加一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数	vector、list、deque
a.push_front(t)	将 t 的拷贝插入到第一个元素前面	forward_list、list、deque
a.push_front(rv)	将 rv 的拷贝插入到第一个元素前面；可能使用移动语义	forward_list、list、deque
a.emplace_front()	在最前面插入一个类型为 T 的对象，创建该对象时使用与 args 封装的参数匹配的构造函数	forward_list、list、deque
a.pop_front()	删除第一个元素	forward_list、list
a[n]	返回*(a.begin()+ n)	vector、deque、array
a.at(n)	返回*(a.begin()+ n)；如果 n>a.size，则引发 out_of_range 异常	vector、deque、array

模板 vector 还包含表 G.7 列出的方法。其中，a 是 vector 容器，n 是 x::size_type 型整数。

表 G.7　　vector 的其他操作

操　　作	描　　述
a.capacity()	返回在不要求重新分配内存的情况下，矢量能存储的元素总量
a.reserve(n)	提醒 a 对象：至少需要存储 n 个元素的内存。调用该方法后，容量至少为 n 个元素。如果 n 大于当前的容量，则需要重新分配内存。如果 n 大于 a.max_size()，该方法将引发 length_error 异常

模板 list 还包含表 G.8 列出的方法。其中，a 和 b 是 list 容器；T 是存储在链表中的类型，如 int；t 是类型为 T 的值；i 和 j 是输入迭代器；q2 和 p 是迭代器；q 和 q1 是可解除引用的迭代器；n 是 x::size_type 型整数。该表使用了标准的 STL 表示法[i, j)，这指的是从 i 到 j（不包括 j）的区间。

表 G.8　list 的其他操作

方　法	描　述
a.splice(p, b)	将链表 b 的内容移到链表 a 中，并将它们插在 p 之前
a.splice(p, b, i)	将 i 指向的链表 b 中的元素移到链表 a 的 p 位置之前
a.splice(p, b, i, j)	将链表 b 中[i，j)区间内的元素移到链表 a 的 p 位置之前
a.remove(const T& t)	删除链表 a 中值为 t 的所有元素
a.remove_if(Predicate pred)	如果 i 是指向链表 a 中元素的迭代器，则删除 pred(*i)为 true 的所有值（Predicate 是布尔值函数或函数对象，参见第 15 章）
a.unique()	删除连续的相同元素组中除第一个元素之外的所有元素
a.unique(BinaryPredicate bin_pred)	删除连续的 bin_pred(*i, *(i - 1))为 true 的元素组中除第一个元素之外的所有元素（BinaryPredicate 是布尔值函数或函数对象，参见第 15 章）
a.merge(b)	使用为值类型定义的<运算符，将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则 a 中的元素将放在前面。合并后，链表 b 为空
a.merge(b, Compare comp)	使用 comp 函数或函数对象将链表 b 与链表 a 的内容合并。如果链表 a 的某个元素与链表 b 的某个元素相同，则链表 a 中的元素将放在前面。合并后，链表 b 为空
a.sort()	使用<运算符对链表 a 进行排序
a.sort(Compare comp)	使用 comp 函数或函数对象对链表 a 进行排序
a.reverse()	将链表 a 中的元素顺序反转

forward_list 的操作与此类似，但由于模板类 forward_list 的迭代器不能后移，有些方法必须调整。因此，用 insert_after()、erase_after ()和 splice_after()替代了 insert()、erase()和 splice()，这些方法都对迭代器后面而不是前面的元素进行操作。

G.4　set 和 map 的其他操作

关联容器（集合和映射是这种容器的模型）带有模板参数 Key 和 Compare，这两个参数分别表示用来对内容进行排序的键类型和用于对键值进行比较的函数对象（被称为比较对象）。对于 set 和 multiset 容器，存储的键就是存储的值，因此键类型与值类型相同。对于 map 和 multimap 容器，存储的值（模板参数 T）与键类型（模板参数 Key）相关联，值类型为 pair<const Key, T>。关联容器有其他成员来描述这些特性，如表 G.9 所示。

表 G.9　为关联容器定义的类型

类　型	值
X::key_type	Key，键类型
X::key_compare	Compare，默认为 less<key_type>
X::value_compare	二元谓词类型，与 set 和 multiset 的 key_compare 相同，为 map 或 multimap 容器中的 pair<const Key, T>值提供了排序功能
X::mapped_type	T，关联数据类型（仅限于 map 和 multimap）

关联容器提供了表 G.10 列出的方法。通常，比较对象不要求键相同的值是相同的；等价键（equivalent key）意味着两个值（可能相同，也可能不同）的键相同。在该表中，X 为容器类，a 是类型为 X 的对象。如果 X 使用唯一键（即为 set 或 map），则 a_uniq 将是类型为 X 的对象。如果 x 使用多个键（即为 multiset 或 multimap），则 a_eq 将是类型为 X 的对象。和前面一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用的迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.10 为 set、multiset、map 和 multimap 定义的操作

操　作	描　述
X(i, j, c)	创建一个空容器，插入区间[i, j]中的元素，并将 c 用作比较对象
X a(i, j, c)	创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 c 用作比较对象
X(i, j)	创建一个空容器，插入区间[i, j]中的元素，并将 Compare()用作比较对象
X a(i, j)	创建一个名为 a 的空容器，插入区间[i, j]中的元素，并将 Compare()用作比较对象
X(il);	等价于 X(il.begin(), il.end())
a = il	将区间[il.begin(), il.end()]的内容赋给 a
a.key_comp()	返回在构造 a 时使用的比较对象
a.value_comp()	返回一个 value_compare 对象
a_uniq.insert(t)	当且仅当 a 不包含具有相同键的值时，将 t 值插入到容器 a 中。该方法返回一个 pair<iterator, bool>值。如果进行了插入，则 bool 的值为 true，否则为 false。iterator 指向键与 t 相同的元素
a_eq.insert(t)	插入 t 并返回一个指向其位置的迭代器
a.insert(p, t)	将 p 作为 insert()开始搜索的位置，将 t 插入。如果 a 是键唯一的容器，则当且仅当 a 不包含拥有相同键的元素时，才插入；否则，将进行插入。无论是否进行了插入，该方法都将返回一个迭代器，该迭代器指向拥有相同键的位置
a.insert(i, j)	将区间[i, j]中的元素插入到 a 中
a.insert(il)	将 initializer_list il 中的元素插入到 a 中
a_uniq.emplace(args)	类似于 a_uniq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a_eq.emplace(args)	类似于 a_eq.insert(t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a.emplace_hint(args)	类似于 a.insert(p, t)，但使用参数列表与参数包 args 的内容匹配的构造函数
a.erase(k)	删除 a 中键与 k 相同的所有元素，并返回删除的元素数目
a.erase(q)	删除 q 指向的元素
a.erase(q1, q2)	删除区间[q1, q2)中的元素
a.clear()	与 erase(a.begin(), a.end())等效
a.find(k)	返回一个迭代器，该迭代器指向键与 k 相同的元素；如果没有找到这样的元素，则返回 a.end()
a.count(k)	返回键与 k 相同的元素的数量
a.lower_bound(k)	返回一个迭代器，该迭代器指向第一个键不小于 k 的元素
a.upper_bound(k)	返回一个迭代器，该迭代器指向第一个键大于 k 的元素
a.equal_range(k)	返回第一个成员为 a.lower_bound(k)，第二个成员为 a.upper_bound(k)的值对
a.operator[ ](k)	返回一个引用，该引用指向与键 k 关联的值（仅限于 map 容器）

G.4 无序关联容器（C++11）

前面说过，无序关联容器（unordered_set、unordered_multiset、unordered_map 和 unordered_multimap）使用键和哈希表，以便能够快速存取数据。下面简要地介绍这些概念。哈希函数（hash function）将键转换为索引值。例如，如果键为 string 对象，哈希函数可能将其中每个字符的数字编码相加，再计算结果除以 13 的余数，从而得到一个 0～12 的索引。而无序容器将使用 13 个桶（bucket）来存储 string，所有索引为 4 的 string 都将存储在第 4 个桶中。如果您要在容器中搜索键，将对键执行哈希函数，进而只在索引对应的桶中搜索。理想情况下，应有足够多的桶，每个桶只包含为数不多的 string。

C++11 库提供了模板 hash<key>，无序关联容器默认使用该模板。为各种整型、浮点型、指针以及一些模板类（如 string）定义了该模板的具体化。

表 G.11 列出了用于这些容器的类型。

无序关联容器的接口类似于关联容器。具体地说，表 G.10 也适用于无序关联容器，但存在如下例外：不需要方法 lower_bound()和 upper_bound()，构造函数 X(i, j, c) 亦如此。常规关联容器是经过排序的，这让它们能够使用表示“小于”概念的比较谓词。这种比较不适用于无序关联容器，因此它们使用基于概念“等于”的比较谓词。

表 G.11　为无序关联容器定义的类型

类　型	值
X::key_type	Key，键类型
X::key_equal	Pred，一个二元谓词，检查两个类型为 Key 的参数是否相等
X::hasher	Hash，一个这样的二元函数对象，即如果 hf 的类型为 Hash，k 的类型为 Key，则 hf(k) 的类型为 std::size_t
X::local_iterator	一个类型与 X::iterator 相同的迭代器，但只能用于一个桶
X::const_local_iterator	一个类型与 X::const_iterator 相同的迭代器，但只能用于一个桶
X::mapped_type	T，关联数据类型（仅限于 map 和 multimap）

除表 G.10 所示的方法外，无序关联容器还包含其他一些必不可少的方法，如表 G.12 所示。在该表中，X 为无序关联容器类，a 是类型为 X 的对象，b 可能是类型为 X 的常量对象，a_uniq 是类型为 unordered_set 或 unordered_map 的对象，a_eq 是类型为 unordered_multiset 或 unordered_multimap 的对象，hf 是类型为 hasher 的值，eq 是类型为 key_equal 的值，n 是类型为 size_type 的值，z 是类型为 float 的值。与以前一样，i 和 j 也是指向 value_type 元素的输入迭代器，[i, j]是一个有效的区间，p 和 q2 是指向 a 的迭代器，q 和 q1 是指向 a 的可解除引用迭代器，[q1, q2]是有效区间，t 是 X::value_type 值（可能是一对），k 是 X::key_type 值，而 il 是 initializer_list<value_type>对象。

表 G.12　为无序关联容器定义的操作

操　作	描　述
X(n, hf, eq)	创建一个至少包含 n 个桶的空容器，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略了 eq，则将 key_equal()用作键值相等谓词；如果也省略了 hf，则将 hasher()用作哈希函数
X a(n, hf, eq)	创建一个名为 a 的空容器，它至少包含 n 个桶，并将 hf 用作哈希函数，将 eq 用作键值相等谓词。如果省略 eq，则将 key_equal()用作键值相等谓词；如果也省略了 hf，则将 hasher()用作哈希函数
X(i, j, n, hf, eq)	创建一个至少包含 n 个桶的空容器，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal()用作键值相等谓词；如果省略了 hf，将 hasher()用作哈希函数；如果省略了 n，则包含桶数不确定
X a(i, j, n, hf, eq)	创建一个名为 a 的的空容器，它至少包含 n 个桶，将 hf 用作哈希函数，将 eq 用作键值相等谓词，并插入区间[i, j]中的元素。如果省略了 eq，将 key_equal()用作键值相等谓词；如果省略了 hf，将 hasher()用作哈希函数；如果省略了 n，则包含桶数不确定
b.hash_function()	返回 b 使用的哈希函数
b.key_eq()	返回创建 b 时使用的键值相等谓词
b.bucket_count()	返回 b 包含的桶数
b.max_bucket_count ()	返回一个上限数，它指定了 b 最多可包含多少个桶
b.bucket(k)	返回键值为 k 的元素所属桶的索引
b.bucket_size(n)	返回索引为 n 的桶可包含的元素数
b.begin(n)	返回一个迭代器，它指向索引为 n 的桶中的第一个元素
b.end(n)	返回一个迭代器，它指向索引为 n 的桶中的最后一个元素
b.cbegin(n)	返回一个常量迭代器，它指向索引为 n 的桶中的第一个元素
b.cend(n)	返回一个常量迭代器，它指向索引为 n 的桶中的最后一个元素
b.load_factor()	返回每个桶包含的平均元素数
b.max_load_factor()	返回负载系数的最大可能取值；超过这个值后，容器将增加桶
b.max_load_factor(z)	可能修改最大负载系统，建议将它设置为 z
a.rehash(n)	将桶数调整为不小于 n，并确保 a.bucket_count() > a.size() / a.max_load_factor()
a.reserve(n)	等价于 a.rehash(ceil(n/a.max_load_factor()))，其中 ceil(x)返回不小于 x 的最小整数

G.5　STL 函数

STL 算法库（由头文件 algorithm 和 numeric 支持）提供了大量基于迭代器的非成员模板函数。正如第

16 章介绍的，选择的模板参数名指出了特定参数应模拟的概念。例如，ForwardIterator 用于指出，参数至少应模拟正向迭代器的要求；Predicate 用于指出，参数应是一个接受一个参数并返回 bool 值的函数对象。C++标准将算法分成 4 组：非修改式序列操作、修改式序列操作、排序和相关运算符以及数值操作（C++11 将数值操作从 STL 移到了 numeric 库中，但这不影响它们的用法）。序列操作（sequence operation）表明，函数将接受两个迭代器作为参数，它们定义了要操作的区间或序列。修改式（mutating）意味着函数可以修改容器的内容。

G.5.1 非修改式序列操作

表 G.13 对非修改式序列操作进行了总结。这里没有列出参数，而重载函数只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.13 非修改式序列操作

函 数	描 述
all_of()	如果对于所有元素的谓词测试都为 true，则返回 true。这是 C++11 新增的
any_of()	只要对于任何一个元素的谓词测试为 true，就返回 true。这是 C++11 新增的
none_of()	如果对于所有元素的谓词测试都为 false，则返回 true。这是 C++11 新增的
for_each()	将一个非修改式函数对象用于区间中的每个成员
find()	在区间中查找某个值首次出现的位置
find_if()	在区间中查找第一个满足谓词测试条件的值
find_if_not()	在区间中查找第一个不满足谓词测试条件的值。这是 C++11 新增的
find_end()	在序列中查找最后一个与另一个序列匹配的值。匹配时可以使用等式或二元谓词
find_first_of()	在第二个序列中查找第一个与第一个序列的值匹配的元素。匹配时可以使用等式或二元谓词
adjacent_find	查找第一个与其后面的元素匹配的元素。匹配时可以使用等式或二元谓词
count()	返回特定值在区间中出现的次数
count_if()	返回特定值与区间中的值匹配的次数，匹配时使用二元谓词
mismatch()	查找区间中第一个与另一个区间中对应元素不匹配的元素，并返回指向这两个元素的迭代器。匹配时可以使用等式或二元谓词
equal()	如果一个区间中的每个元素都与另一个区间中的相应元素匹配，则返回 true。匹配时可以使用等式或二元谓词
is_permutation()	如果可通过重新排列第二个区间，使得第一个区间和第二个区间对应的元素都匹配，则返回 true，否则返回 false。匹配可以是相等，也可以使用二元谓词进行判断。这是 C++11 新增的
search()	在序列中查找第一个与另一个序列的值匹配的值。匹配时可以使用等式或二元谓词
search_n()	查找第一个由 n 个元素组成的序列，其中每个元素都与给定值匹配。匹配时可以使用等式或二元谓词

下面更详细地讨论这些非修改型序列操作。对于每个函数，首先列出其原型，然后做简要地描述。和前面一样，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。有些函数接受两个区间，这两个区间的容器类型可以不同。例如，可以使用 equal()来对链表和矢量进行比较。作为参数传递的函数是函数对象，这些函数对象可以是指针（如函数名），也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。

1. all_of()（C++11）

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 true，或者该区间为空，则函数 all_of()返回 true；否则返回 false。

2. any_of()（C++11）

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 any_of()返回 false；否则返回 true。

3. none_of()（C++11）

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last,
             Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 all_of()返回 true；否则返回 false。

4. for_each()

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f);
```

for_each()函数将函数对象 f 用于[first, last]区间中的每个元素，它也返回 f。

5. find()

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,
                   const T& value);
```

find()函数返回一个迭代器，该迭代器指向区间[first, last]中第一个值为 value 的元素；如果没有找到这样的元素，则返回 last。

6. find_if()

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if()函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 true 的元素；如果没有找到这样的元素，则返回 last。

7. find_if_not()

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if_not()函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 false 的元素；如果没有找到这样的元素，则返回 last。

8. find_end()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_end(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
```

find_end()函数返回一个迭代器，该迭代器指向[first1, last1] 区间中最后一个与[first2, last2] 区间的内容匹配的序列的第一个元素。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都返回 last1。

9. find_first_of()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 find_first_of(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
```

find_first_of() 函数返回一个迭代器，该迭代器指向区间[first1, last1]中第一个与[first2, last2]区间中的任

何元素匹配的元素。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都将返回 last1。

10. adjacent_find()

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
         ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
         ForwardIterator last, BinaryPredicate pred);
```

adjacent_find()函数返回一个迭代器，该迭代器指向[first1, last1] 区间中第一个与其后面的元素匹配的元素。如果没有找到这样的元素，则返回 last。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

11. count()

```
template<class InputIterator, class T>
 typename iterator_traits<InputIterator>::difference_type
  count(InputIterator first, InputIterator last, const T& value);
```

count()函数返回[first, last)区间中与值 value 匹配的元素数目。对值进行比较时，将使用值类型的= =运算符。返回值类型为整型，它足以存储容器所能存储的最大元素数。

12. count_if()

```
template<class InputIterator, class Predicate>
 typename iterator_traits<InputIterator>::difference_type
  count_if(InputIterator first, InputIterator last, Predicate pred);
```

count if()函数返回[first, last]区间中这样的元素数目，即将其作为参数传递给函数对象 pred 时，后者的返回值为 true。

13. mismatch()

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
   mismatch(InputIterator1 first1,
            InputIterator1 last1, InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
                              class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1,
             InputIterator1 last1, InputIterator2 first2,
             BinaryPredicate pred);
```

每个 mismatch()函数都在[first1, last1)区间中查找第一个与从 first2 开始的区间中相应元素不匹配的元素，并返回两个迭代器，它们指向不匹配的两个元素。如果没有发现不匹配的情况，则返回值为 pair<last1, first2 + (last1 - first1)>。第一个版本使用= =运算符来测试匹配情况；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 false，则 it1 和 it2 指向的元素不匹配。

14. equal()

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果[first1, last1)区间中每个元素都与以 first2 开始的序列中相应元素匹配，则 equal()函数返回 true，否则返回 false。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

15. is_permutation()（C++11）

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
```

```
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);
```

如果通过对从 first2 开始的序列进行排列，可使其与区间[first1, last1]相应的元素匹配，则函数 is_permutation()返回 true，否则返回 false。第一个版本使用值类型的==运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素，也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。

16. search()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         BinaryPredicate pred);
```

search()函数在[first1, last1]区间中搜索第一个与[first2, last2] 区间中相应的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

17. search_n()

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value, BinaryPredicate pred);
```

search_n()函数在[first1, last1)区间中查找第一个与 count 个 value 组成的序列匹配的序列；如果没有找到这样的序列，则返回 last1。第一个版本使用值类型的= =运算符来对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

G.5.2　修改式序列操作

表 G.14 对修改式序列操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.14　修改式序列操作

函　数	描　述
copy()	将一个区间中的元素复制到迭代器指定的位置
copy_n()	从一个迭代器指定的地方复制 n 个元素到另一个迭代器指定的地方，这是 C++11 新增的
copy_if()	将一个区间中满足谓词测试的元素复制到迭代器指定的地方，这是 C++11 新增的
copy_backward()	将一个区间中的元素复制到迭代器指定的地方。复制时从区间结尾开始，由后向前进行
move()	将一个区间中的元素移到迭代器指定的地方，这是 C++11 新增的
move_backward()	将一个区间中的元素移到迭代器指定的地方；移动时从区间结尾开始，由后向前进行。这是 C++11 新增的
swap()	交换引用指定的位置中存储的值
swap_ranges()	对两个区间中对应的值进行交换
iter_swap()	交换迭代器指定的位置中存储的值
transform()	将函数对象用于区间中的每一个元素（或区间对中的每对元素），并将返回的值复制到另一个区间的相应位置
replace()	用另外一个值替换区间中某个值的每个实例
replace_if()	如果用于原始值的谓词函数对象返回 true，则使用另一个值来替换区间中某个值的所有实例
replace_copy()	将一个区间复制到另一个区间中，使用另外一个值替换指定值的每个实例
replace_copy_if()	将一个区间复制到另一个区间，使用指定值替换谓词函数对象为 true 的每个值

续表

函　　数	描　　述
fill()	将区间中的每一个值设置为指定的值
fill_n()	将 n 个连续元素设置为一个值
generate()	将区间中的每个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象
generate_n()	将区间中的前 n 个值设置为生成器的返回值，生成器是一个不接受任何参数的函数对象
remove()	删除区间中指定值的所有实例，并返回一个迭代器，该迭代器指向得到的区间的超尾
remove_if()	将谓词对象返回 true 的值从区间中删除，并返回一个迭代器，该迭代器指向得到的区间的超尾
remove_copy()	将一个区间中的元素复制到另一个区间中，复制时忽略与指定值相同的元素
remove_copy_if()	将一个区间中的元素复制到另一个区间中，复制时忽略谓词函数对象返回 true 的元素
unique()	将区间内两个或多个相同元素组成的序列压缩为一个元素
unique_copy()	将一个区间中的元素复制到另一个区间中，并将两个或多个相同元素组成的序列压缩为一个元素
reverse()	反转区间中的元素的排列顺序
reverse_copy()	按相反的顺序将一个区间中的元素复制到另一个区间中
rotate()	将区间中的元素循环排列，并将元素左转
rotate_copy()	以旋转顺序将区间中的元素复制到另一个区间中
random_shuffle()	随机重新排列区间中的元素
shuffle()	随机重新排列区间中的元素，使用的函数对象满足 C++11 对统一随机生成器的要求
is_partitioned()	如果区间根据指定的谓词进行了分区，则返回 true
partition()	将满足谓词函数对象的所有元素都放在不满足谓词函数对象的元素之前
stable_partition()	将满足谓词函数对象的所有元素放置在不满足谓词函数对象的元素之前，每组中元素的相对顺序保持不变
partition_copy()	将满足谓词函数对象的所有元素都复制到一个输出区间中，并将其他元素都复制到另一个输出区间中，这是 C++11 新增的
partition_point()	对于根据指定谓词进行了分区的区间，返回一个迭代器，该迭代器指向第一个不满足该谓词的元素

下面详细地介绍这些修改型序列操作。对于每个函数，首先列出其原型，然后做简要的描述。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last]区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受两个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

copy()函数将[first, last)区间中的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

2. copy_n()（C++11）

```
template<class InputIterator, class Size class OutputIterator>
OutputIterator copy(InputIterator first, Size n,
                    OutputIterator result);
```

函数 copy_n()从位置 first 开始复制 n 个元素到区间[result, result + n] 中，并返回 result + n，即指向被复制到的最后一个位置后面的迭代器。该函数不要求目标和源不重叠。

3. copy_if()（C++11）

```
template<class InputIterator, class OutputIterator,
         class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);
```

函数 copy_if()将[first, last)区间中满足谓词 pred 的元素复制到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)

区间中，也就是说，目标不能与源重叠。

4. copy_backward()

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 copy_backward()将[first, last)区间中的元素复制到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

5. move()（C++11）

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

函数 move()使用 std::move()将[first, last)区间中的元素移到区间[result, result + (last – first))中，并返回 result + (last – first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中，也就是说，目标不能与源重叠。

6. move_backward()（C++11）

```
template<class BidirectionalIterator1,
         class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,

BidirectionalIterator1 last, BidirectionalIterator2 result);
```

函数 move_backward() std::move()将[first, last)区间中的元素移到区间[result - (last - first), result)中。复制从 last - 1 开始，该元素被复制到位置 result - 1，然后由后向前处理，直到 first。该函数返回 result - (last - first)，即指向被复制到的最后一个位置后面的迭代器。该函数要求 result 不位于[first, last)区间中。然而，由于复制是从后向前进行的，因此目标和源可能重叠。

7. swap()

```
template<class T> void swap(T& a, T& b);
```

swap()函数对引用指定的两个位置中存储的值进行交换（C++11 将这个函数移到了头文件 utility 中）。

8. swap_ranges()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
                         ForwardIterator1 first1, ForwardIterator1 last1,
                         ForwardIterator2 first2);
```

swap_ranges()函数将[first1, last1]区间中的值与从 first2 开始的区间中对应的值交换。这两个区间不能重叠。

9. iter_swap()

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

iter_swap()函数将迭代器指定的两个位置中存储的值进行交换。

10. transform()

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

第一个版本的 transform()将一元函数对象 op 应用到[first, last)区间中每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置为 op(*first)，依此类推。该函数返回 result + (last - first)，即目标区间的超尾值。

第二个版本的 transform()将二元函数对象 op 应用到[first1, last1)区间和[first2, last2)区间中的每个元素，并将返回值赋给从 result 开始的区间中对应的元素。因此，*result 被设置成 op(*first1, *first2)，依此类推。该函数返回 result + (last – first)，即目标区间的超尾值。

11. replace()

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

replace()函数将[first, last]中的所有 old_value 替换为 new_value。

12. replace_if()

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

replace_if()函数使用 new_value 值替换[first, last]区间中 pred（old）为 true 的每个 old 值。

13. replace_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
     OutputIterator result,const T& old_ value, const T& new_ value);
```

replace_copy()函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替所有的 old_value。该函数返回 result + (last - first)，即目标区间的超尾值。

14. replace_copy_if()

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
    OutputIterator result, Predicate pred, const T& new_ value);
```

replace_copy_if()函数将[first, last]区间中的元素复制到从 result 开始的区间中，但它使用 new_value 代替 pred(old)为 true 的所有 old 值。该函数返回 result + (last - first)，即目标区间的超尾值。

15. fill()

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

fill()函数将[first, last]区间中的每个元素都设置为 value。

16. fill_n()

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

fill_n()函数将从 first 位置开始的前 n 个元素都设置为 value。

17. generate()

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

generate()函数将[first, last)区间中的每个元素都设置为 gen()，其中 gen 是一个生成器函数对象，即不接受任何参数。例如，gen 可以是一个指向 rand()的指针。

18. generate_n()

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

generate_n()函数将从 first 开始的区间中前 n 个元素都设置为 gen()，其中，gen 是一个生成器函数对象，即不接受任何参数。例如，gen 可以是一个指向 rand()的指针。

19. remove()

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

remove()函数删除[first, last)区间中所有值为 value 的元素，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

注意：由于所有的 remove()和 unique()函数都不是成员函数，同时这些函数并非只能用于 STL 容器，因此它们不能重新设置容器的长度。相反，它们返回一个指示新超尾位置的迭代器。通常，被删除的元素只是被移到容器尾部。然而，对于 STL 容器，可以使用返回的迭代器和 erase()方法来重新设置 end()。

20. remove_if()

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

remove_if()函数将 pred(val)为 true 的所有 val 值从[first, last)区间删除，并返回得到的区间的超尾迭代器。该函数是稳定的，这意味着未删除的元素的顺序将保持不变。

21. remove_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

remove_copy()函数将[first, last)区间中的值复制到从 result 开始的区间中，复制时将忽略 value。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有被删除的元素的顺序将保持不变。

22. remove_copy_if()

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

remove_copy_if()函数将[first, last)区间中的值复制到从 result 开始的区间，但复制时忽略 pred(val)为 true 的 val。该函数返回得到的区间的超尾迭代器。该函数是稳定的，这意味着没有删除的元素的顺序将保持不变。

23. unique()

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

unique()函数将[first, last)区间中由两个或更多相同元素构成的序列压缩为一个元素，并返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

24. unique_copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

unique_copy()函数将[first, last)区间中的元素复制到从 result 开始的区间中，并将由两个或更多个相同元素组成的序列压缩为一个元素。该函数返回新区间的超尾迭代器。第一个版本使用值类型的= =运算符，对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素是匹配的。

25. reverse()

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

reverse()函数通过调用 swap(first, last - 1)等来反转[first, last]区间中的元素。

26. reverse_copy

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result);
```

reverse_copy()函数按相反的顺序将[first, last)区间中的元素复制到从 result 开始的区间中。这两个区间不能重叠。

27. rotate()

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

rotate()函数将[first, last)区间中的元素左旋。middle 处的元素被移到 first 处，middle + 1 处的元素被移到 first + 1 处，依此类推。middle 前的元素绕回到容器尾部，以便 first 处的元素可以紧接着 last - 1 处的元素。

28. rotate_copy()

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

rotate_copy()函数使用为 rotate()函数描述的旋转序列，将[first, last)区间中的元素复制到从 result 开始的区间中。

29. random_shuffle()

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

这个版本的 random_shuffle()函数将[first, last)区间中的元素打乱。分布是一致的，即原始顺序的每种可能排列方式出现的概率相同。

30. random_shuffle()

```
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator&& random);
```

这个版本的 random_shuffle()函数将[first, last)区间中的元素打乱。函数对象 random 确定分布。假设有 n 个元素，表达式 random(n)将返回[0, n)区间中的一个值。在 C++98 中，参数 random 是一个左值引用，而在 C++11 中是一个右值引用。

31. shuffle()

```
template<class RandomAccessIterator, class Uniform RandomNumberGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    UniformRandomNumberGenerator&& rgen);
```

函数 shuffle()将[first, last)区间中的元素打乱。函数对象 rgen 确定分布，它应满足 C++11 指定的有关均匀随机数生成器的要求。假设有 n 个元素，表达式 rgen(n)将返回[0, n]区间中的一个值。

32. is_partitioned()（C++11）

```
template<class InputIterator, class Predicate>
bool is_partitioned(InputIterator first,
                    InputIterator last, Predicate pred);
```

如果区间为空或根据 pred 进行了分区（即满足谓词 pred 的元素都在不满足该谓词的元素前面），函数 is__partitioned()将返回 true，否则返回 false。

33. partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                Predicate pred);
```

函数 partition()将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

34. stable_partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

函数 stable_partition()将其值 val 使得 pred(val)为 true 的元素都放在不满足该测试条件的所有元素之前；在这两组中，元素的相对顺序保持不变。这个函数返回一个迭代器，指向最后一个使得谓词对象函数为 true 的值的后面。

35. partition_copy()（C++11）

```
template<class InputIterator, class OutputIterator1,
         classs OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
     InputIterator first, InputIterator last,
     OutputIterator1 out_true, OutputIterator2 out_false
                                Predicate pred);
```

函数 partition_copy()将所有这样的元素都复制到从 out_true 开始的区间中，即其值 val 使得 pred(val)为 true；并将其他的元素都复制到从 out_false 开始的区间中。它返回一个 pair 对象，该对象包含两个迭代器，分别指向从 out_true 和 out_false 开始的区间的末尾。

36. partition_point()（C++11）

```
template<class ForwardIterator, class Predicate>
ForewardIterator partition_point(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

函数 partition_point()要求区间根据 pred 进行了分区。它返回一个迭代器，指向最后一个让谓词对象函数为 true 的值所在的位置。

G.5.3　排序和相关操作

表 G.15 对排序和相关操作进行了总结。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能，如果对某个函数非常感兴趣，则可以了解其细节。

表 G.15　排序和相关操作

函　　数	描　　述
sort()	对区间进行排序
stable_sort()	对区间进行排序，并保留相同元素的相对顺序
partial_sort()	对区间进行部分排序，提供完整排序的前 n 个元素
partial_sort_copy()	将经过部分排序的区间复制到另一个区间中
is_sorted()	如果对区间进行了排序，则返回 true，这是 C++11 新增的
is_sorted_until()	返回一个迭代器，指向经过排序的区间末尾，这是 C++11 新增的
nth_element()	对于给定指向区间的迭代器，找到区间被排序时，相应位置将存储哪个元素，并将该元素放到这里
lower_bound()	对于给定的一个值，在一个排序后的区间中找到第一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序
upper_bound()	对于给定的一个值，在一个排序后的区间中找到最后一个这样的位置，使得将这个值插入到这个位置前面时，不会破坏顺序
equal_range()	对于给定的一个值，在一个排序后的区间中找到一个最大的区间，使得将这个值插入其中的任何位置，都不会破坏顺序
binary_search()	如果排序后的区间中包含了与给定的值相同的值，则返回 true，否则返回 false
merge()	将两个排序后的区间合并为第三个区间
inplace_merge()	就地合并两个相邻的、排序后的区间
includes()	如果对于一个集合中的每个元素都可以在另外一个集合中找到，则返回 true
set_union()	构造两个集合的并集，其中包含在任何一个集合中出现过的元素
set_intersection()	构造两个集合的交集，其中包含在两个集合中都出现过的元素
set_difference()	构造两个集合的差集，即包含第一个集合中且没有出现在第二个集合中的所有元素
set_symmetric_difference()	构造由只出现在其中一个集合中的元素组成的集合
make_heap()	将区间转换成堆
push_heap()	将一个元素添加到堆中
pop_heap()	删除堆中最大的元素
sort_heap()	对堆进行排序
is_heap()	如果区间是堆，则返回 true，这是 C++11 新增的
is_heap_until()	返回一个迭代器，指向属于堆的区间的末尾，这是 C++11 新增的
min()	返回两个值中较小的值，如果参数为 initializer_list，则返回最小的元素（这是 C++11 新增的）
max()	返回两个值中较大的值，如果参数为 initializer_list，则返回最大的元素（这是 C++11 新增的）
minmax()	返回一个 pair 对象，其中包含按递增顺序排列的两个参数；如果参数为 initializer_list，则返回 pair 对象包含最小和最大的元素。这是 C++11 新增的
min_element()	在区间找到最小值第一次出现的位置
max_element()	在区间找到最大值第一次出现的位置
minmax_element()	返回一个 pair 对象，其中包含两个迭代器，它们分别指向区间中最小值第一次出现的位置和区间中最大值最后一次出现的位置。这是 C++11 新增的
lexicographic_compare()	按字母顺序比较两个序列，如果第一个序列小于第二个序列，则返回 true，否则返回 false
next-permutation()	生成序列的下一种排列方式
previous_permutation()	生成序列的前一种排列方式

本节中的函数使用为元素定义的<运算符或模板类型 Compare 指定的比较对象来确定两个元素的顺序。如果 comp 是一个 Compare 类型的对象，则 comp(a, b)就是 a<b 的统称，如果在排序机制中，a 在 b 之前，则返回 true。如果 a<b 返回 fasle，同时 b<a 也返回 false，则说明 a 和 b 相等。比较对象必须至少提供严格

弱排序功能（strict weak ordering）。这意味着：

- 表达式 comp(a, a)一定为 false，这是“值不能比其本身小”的统称（这是严格部分）。
- 如果 comp(a, b)为 true，且 comp(b, c)也为 true，则 comp(a, c)为 true（也就是说，比较是一种可传递的关系）。
- 如果 a 与 b 等价，且 b 与 c 也等价，则 a 与 c 等价（也就是说，等价也是一种可传递的关系）。

如果想将<运算符用于整数，则等价就意味着相等，但这一结论不能推而广之。例如，可以用几个描述邮件地址的成员来定义一个结构，同时定义一个根据邮政编码对结构进行排序的 comp 对象。则邮政编码相同的地址是等价的，但它们并不相等。

下面更详细地介绍排序及相关操作。对于每个函数，首先列出其原型，然后做简要的说明。我们将这一节分成几个小节。正如前面介绍的，迭代器对指出了区间，而选择的模板参数名指出了迭代器的类型。通常，[first, last)区间指的是从 first 到 last（不包括 last）。作为参数传递的函数是函数对象，这些函数对象可以是指针，也可以是定义了()操作的对象。正如第 16 章介绍的，谓词是接受一个参数的布尔函数，二元谓词是接受 2 个参数的布尔函数（函数可以不是 bool 类型，只要它对于 false 返回 0，对于 true 返回非 0 值）。另外，正如第 16 章介绍的，一元函数对象接受一个参数，而二元函数对象接受两个参数。

1. 排序

首先来看看排序算法。

（1）sort()

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

sort()函数将[first, last)区间按升序进行排序，排序时使用值类型的<运算符进行比较。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）stable_sort()

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

stable_sort()函数对[first, last)区间进行排序，并保持等价元素的相对顺序不变。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）partial_sort()

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
RandomAccessIterator last, Compare comp);
```

partial_sort()函数对[first, last)区间进行部分排序。将排序后的区间的前 middle - first 个元素置于[first, middle]区间内，其余元素没有排序。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）partial_sort_copy()

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                       InputIterator last,
                       RandomAccessIterator result_first,
                       RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

partial_sort_copy()函数将排序后的区间[first, last]中的前 n 个元素复制到区间[result_first, result_first + n]

中。n 的值是 last - first 和 result_last - result_first 中较小的一个。该函数返回 result - first + n。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（5）is_sorted（C++11）

```
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]是经过排序的，函数 is_sorted()将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（6）is_sorted_until（C++11）

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last
               Compare comp);
```

如果区间[first, last]包含的元素少于两个，函数 is_sorted_until 将返回 last；否则将返回迭代器 it，确保区间[first, it]是经过排序的。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（7）nth_element()

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
RandomAccessIterator last, Compare comp);
```

nth_element()函数找到将[first, last)区间排序后的第 n 个元素，并将该元素置于第 n 个位置。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

2. 二分法搜索

二分法搜索组中的算法假设区间是经过排序的。这些算法只要求正向迭代器，但使用随机迭代器时，效率最高。

（1）lower_bound()

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

lower_bound()函数在排序后的区间[first, last)中找到第一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）upper_bound()

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

upper_bound()函数在排序后的区间[first, last)中找到最后一个这样的位置，即将 value 插入到它前面时不会破坏顺序。它返回一个指向这个位置的迭代器。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）equal_range()

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(
    ForwardIterator first, ForwardIterator last, const T& value);
template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
```

```
    ForwardIterator first, ForwardIterator last, const T& value,
    Compare comp);
```

equal_range()函数在排序后的区间[first, last)区间中找到这样一个最大的子区间[it1, it2)，即将 value 插入到该区间的任何位置都不会破坏顺序。该函数返回一个由 it1 和 it2 组成的 pair。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（4）binary_search()

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

如果在排序后的区间[first, last]中找到与 value 等价的值，则 binary_search()函数返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

注意：前面说过，使用<进行排序时，如果 a<b 和 b<a 都为 false，则 a 和 b 等价。对于常规数字来说，等价意味着相等；但对于只根据一个成员进行排序的结构来说，情况并非如此。因此，在确保顺序不被破坏的情况下，可插入新值的位置可能不止一个。同样，如果使用比较对象 comp 进行排序，等价意味着 comp(a, b)和 comp（b，a）都为 false（这是“如果 a 不小于 b，b 也不小于 a，则 a 与 b 等价”的统称）。

3. 合并

合并函数假设区间是经过排序的。

（1）merge()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator,
class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

merge()函数将排序后的区间[first1, last1] 中的元素与排序后的区间[first2, last2] 中的元素进行合并，并将结果放到从 result 开始的区间中。目标区间不能与被合并的任何一个区间重叠。在两个区间中发现了等价元素时，第一个区间中的元素将位于第二个区间中的元素前面。返回值是合并的区间的超尾迭代器。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

（2）inplace_merge()

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
     BidirectionalIterator middle, BidirectionalIterator last,
     Compare comp);
```

inplace_merge()函数将两个连续的、排序后的区间——[first, middle] 和[middle, last] ——合并为一个经过排序的序列，并将其存储在[first, last] 区间中。第一个区间中的元素将位于第二个区间中的等价元素之前。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

4. 使用集合

集合操作可用于所有排序后的序列，包括集合（set）和多集合（multiset）。对于存储一个值的多个实例的容器（如 multiset）来说，定义是广义的。对于两个多集合的并集，将包含较大数目的元素实例，而交集将包含较小数目的元素实例。例如，假设多集合 A 包含了字符串“apple”7 次，多集合 B 包含该字符串 4 次。则 A 和 B 的并集将包含 7 个“apple”实例，它们的交集将包含 4 个实例。

（1）includes()

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2, Compare comp);
```

如果[first2, last2)区间中的每一个元素在[first1, last1)区间中都可以找到，则 includes()函数返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（2）set_union()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

set_union()函数构造一个由[first1, last1]区间和[first2, last2] 区间组合而成的集合，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。并集包含在任何一个集合（或两个集合）中出现的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（3）set_intersection()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1,InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_intersection()函数构造[first1, last1)区间和[first2, last2)区间的交集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。交集包含两个集合中共有的元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（4）set_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2,
      InputIterator2 last2, OutputIterator result,
      Compare comp);
```

set_difference()函数构造[first1, last1)区间与[first2, last2)区间的差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。差集包含出现在第一个集合中，但不出现在第二个集合中的所有元素。第一个版本使用<来确定顺序，而第二个版本使用比较对象 comp。

（5）set_symmetric_difference()

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(
               InputIterator1 first1, InputIterator1 last1,
```

```
                InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp);
```

set_symmetric_difference()函数构造[first1, last1)区间和[first2, last2)区间的对称（symmetric）差集，并将结果复制到 result 指定的位置。得到的区间不能与原来的任何一个区间重叠。该函数返回构造的区间的超尾迭代器。对称差集包含出现在第一个集合中，但不出现在第二个集合中，或者出现在第二个集合中，但不出现在第一个集合中的所有元素。第一个版本使用<来确定顺序，第二个版本使用比较对象 comp。

5. 使用堆

堆（heap）是一种常用的数据形式，具有这样特征：第一个元素是最大的。每当第一个元素被删除或添加了新元素时，堆都可能需要重新排列，以确保这一特征。设计堆是为了提高这两种操作的效率。

（1）make_heap()

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

make_heap()函数将[first, last)区间构造成一个堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）push_heap()

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

push_heap()函数假设[first, last – 1)区间是一个有效的堆，并将 last - 1 位置（即被假设为有效堆的区间后面的一个位置）上的值添加到堆中，使[first, last)区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）pop_heap()

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

pop_heap()函数假设[first, last)区间是一个有效堆，并将位置 last - 1 处的值与 first 处的值进行交换，使[first, last – 1]区间成为一个有效堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）sort_heap()

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

sort_heap()函数假设[first, last)区间是一个有效堆，并对其进行排序。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）is_heap()（C++11）

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last
               Compare comp);
```

如果区间[first, last]是一个有效的堆，函数 is_heap()将返回 true，否则返回 false。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）is_heap_until()（C++11）

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
            RandomAccessIterator first, RandomAccessIterator last
            Compare comp);
```

如果区间[first, last)包含的元素少于两个，则返回 last；否则返回迭代器 it，而区间[first, it)是一个有效的堆。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

6. 查找最小和最大值

最小函数和最大函数返回两个值或值序列中的最小值和最大值。

（1）min()

```
template<class T> const T& min(const T& a, const T& b);

template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

这些版本的 min()函数返回两个值中较小一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T min(initializer_list<T> t);

template<class T, class Compare>
T min(initializer_list<T> t), Compare comp);
```

这些版本的 min()函数是 C++11 新增的，它返回初始化列表 t 中最小的值。如果有多个相等的值且最小，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（2）max()

```
template<class T> const T& max(const T& a, const T& b);

template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

这些版本的 max() 函数返回这两个值中较大的一个；如果这两个值相等，则返回第一个值。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> T max(initializer_list<T> t);

template<class T, class Compare>
T max(initializer_list<T> t), Compare comp);
```

这些版本的 max()函数是 C++11 新增的，它返回初始化列表 t 中最大的值。如果有多个相等的值且最大，则返回第一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（3）minmax()（C++11）

```
template<class T>
  pair<const T&,const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
  pair<const T&,const T&> minmax(const T& a, const T& b,
                                 Compare comp);
```

如果 b 小于 a，这些版本的 minmax()函数返回 pair(b, a)，否则返回 pair(a, b)。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

```
template<class T> pair<T,T> minmax(initializer_list<T> t);

template<class T, class Compare>
pair<T,T> minmax(initializer_list<T> t), Compare comp);
```

这些版本的 minmax()函数返回初始化列表 t 中最小元素和最大元素的拷贝。如果有多个最小的元素，则返回其中的第一个；如果有多个最大的元素，则返回其中的最后一个。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（4）min_element()

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

min_element()函数返回这样一个迭代器，该迭代器指向[first, last)区间中第一个最小的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（5）max_element()

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

max_element()函数返回这样一个迭代器，该迭代器指向[first, last] 区间中第一个最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（6）minmax_element()（C++11）

```
template<class ForwardIterator>
  pair<ForwardIterator,ForwardIterator>
     minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
  pair<ForwardIterator,ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last,
                   Compare comp);
```

函数 minmax_element()返回一个 pair 对象，其中包含两个迭代器，分别指向区间[first, last)中最小和最大的元素。第一个版本使用<来确定顺序，而第二个版本使用 comp 比较对象。

（7）lexicographical_compare()

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
     InputIterator1 first1, InputIterator1 last1,
     InputIterator2 first2, InputIterator2 last2,
     Compare comp);
```

如果[first1, last1]区间中的元素序列按字典顺序小于[first2, last2]区间中的元素序列，则 lexicographical_compare()函数返回 true，否则返回 false。字典比较将两个序列的第一个元素进行比较，即对*first1 和*first2 进行比较。如果*first1 小于*first2，则该函数返回 true；如果*first2 小于*first1，则返回 fasle；如果相等，则继续比较两个序列中的下一个元素。直到对应的元素不相等或到达了某个序列的结尾，比较才停止。如果在到达某个序列的结尾时，这两个序列仍然是等价的，则较短的序列较小。如果两个序列等价，且长度相等，则任何一个序列都不小于另一个序列，因此函数将返回 false。第一个版本使用<来比较元素，而第二个版本使用 comp 比较对象。字典比较是按字母顺序比较的统称。

7. 排列组合

序列的排列组合是对元素重新排序。例如，由 3 个元素组成的序列有 6 种可能的排列方式，因为对于第一个位置，有 3 种选择；给第一个位置选定元素后，第二个位置有两种选择；第三个位置有 1 种选择。例如，数字 1、2 和 3 的 6 种排列如下：

```
123 132 213 232 312 321
```

通常，由 n 个元素组成的序列有 n*(n−1)*...*1 或 n!种排列。

排列函数假设按字典顺序排列各种可能的排列组合，就像前一个例子中的 6 种排列那样。这意味着，通常，在每个排列之前和之后都有一个特定的排列。例如，213 在 231 之前，312 在 231 之后。然而，第一个排列（如示例中的 123）前面没有其他排列，而最后一个排列（321）后面没有其他排列。

（1）next_permutation()

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

next_permutation()函数将[first, last]区间中的序列转换为字典顺序的下一个排列。如果下一个排列存在，则该函数返回 true；如果下一个排列不存在（即区间中包含的是字典顺序的最后一个排列），则该函数返回 false，并将区间转换为字典顺序的第一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp

比较对象。

（2）prev_permutation()

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

previous_permutation()函数将[first, last] 区间中的序列转换为字典顺序的前一个序列。如果前一个排列存在，则该函数返回 true；如果前一个序列不存在（即区间包含的是字典顺序的第一个排列），则该函数返回 false，并将该区间转换为字典顺序的最后一个排列。第一个版本使用<来确定顺序，而第二个版本则使用 comp 比较对象。

G.5.4　数值运算

表 G.16 对数值运算进行了总结，这些操作是由头文件 numeric 描述的。其中没有列出参数，而重载函数也只列出了一次。每一个函数都有一个使用<对元素进行排序的版本和一个使用比较函数对象对元素进行排序的版本。表后做了更详细的说明，其中包括原型。因此，可以浏览该表，以了解函数的功能；如果对某个函数非常感兴趣，则可以了解其细节。

表 G.16　　数值运算

函　　数	描　　述
accumulate()	计算区间中的值的总和
inner_product()	计算 2 个区间的内部乘积
partial_sum()	将使用一个区间计算得到的小计复制到另一个区间中
adjacent_difference()	将使用一个区间的元素计算得到的相邻差集复制到另一个区间中
iota()	将使用运算符++生成的一系列相邻的值赋给一个区间中的元素，这是 C++11 新增的

1. accumulate()

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

accumulate()函数将 acc 的值初始化为 init，然后按顺序对[first, last]区间中的每一个迭代器 i 执行 acc = acc + * i（第一个版本）或 acc = binary_op(acc, *i)（第二个版本）。然后返回 acc 的值。

2. inner_product()

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

inner_product()函数将 acc 的值初始化为 init，然后按顺序对[first1, last1]区间中的每一个迭代器 i 和[first2, first2 + (last1−first1)]区间中对应的迭代器 j 执行 acc = * i * * j（第一个版本）或 acc = binary_op(*i, *j)（第二个版本）。也就是说，该函数对每个序列的第一个元素进行计算，得到一个值，然后对每个序列中的第二个元素进行计算，得到另一个值，依此类推，直到到达第一个序列的结尾（因此，第二个序列至少应同第一个序列一样长）。然后，函数返回 acc 的值。

3. partial_sum()

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
```

```
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result,
                           BinaryOperation binary_op);
```

partial_sum()函数将*first 赋给*result，将*first + *(first + 1)赋给*(result + 1)（第一个版本），或者将 binary_op(*first, *(first + 1))赋给*(result + 1)（第二个版本），依此类推。也就是说，从 result 开始的序列的第 n 个元素将包含从 first 开始的序列的前 n 个元素的总和（或 binary_op 的等价物）。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许使用结果覆盖原来的序列。

4. adjacent_difference()

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   BinaryOperation binary_op);
```

adjacent_difference()函数将*first 赋给 result(*result = *first)。目标区间中随后的位置将被赋值为源区间中相邻位置的差集（或 binary_op 等价物）。也就是说，目标区间的下一个位置(result +1)将被赋值为*(first + 1) - * first（第一个版本）或 binary_op(*(first + 1), * first)（第二个版本），依此类推。该函数返回结果的超尾迭代器。该算法允许 result 等于 first，也就是说，如果需要，该算法允许结果覆盖原来的序列。

5. iota()（C++11）

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last,T value);
```

函数 iota()将 value 赋给*first，再将 value 递增（就像执行运算++value），并将结果赋给下一个元素，依次类推，直到最后一个元素。

附录 H　精选读物和网上资源

有很多有关 C++和编程的优秀图书和网上资源。下述清单只是其中的一些代表作而不是全部，因此还有很多优秀的图书和网站这里没有列出；然后该清单确实具有广泛的代表性。

H.1　精选读物

- Becker, Pete. The C++ Standard Library Extensions. Upper Saddle River, NJ：Addison-Wesley，2007。

本书讨论第一个 TR1（Technical Report）库。这是一个可选的 C++98 库，但 C++11 包含其大部分元素。涉及的主题包括无序集合模板、智能指针、正则表达式库、随机数库和元组。

- Booch, Grady, Robert A. Maksimchuk, Michael W. Engel, and Bobbi J.Young. Object-Oriented Analysis and Design, Third Edition. Upper Saddle River, NJ：Addison-Wesley, 2007。

本书介绍了 OOP 概念，讨论了 OOP 方法，并提供一些示例应用程序，示例是使用 C++编写的。

- Cline, Marshall, Greg Lomow and Mike Girou. *C++FAQ，Second Edition*（C++常见问题解答，第二版）。Reading，MA：Addison-Wesley，1998。

本书解答了多个经常问到的有关 C++语言的问题。

- Josuttis, Nicolai M. The C++ Standard Library: A Tutorial and Reference（C++标准库教程和参考手册）。Reading. MA：Addison-Wesley，1999。

本书介绍了标准模板库（STL）以及其他 C++库特性，如复数支持、区域和输入/输出流。

- Karlsson, Björn. Beyond the C++ Standard Library:An Introduction to Boost. Upper Saddle River，NJ：Addison-Wesley，2006。

顾名思义，本书探讨多个 Boost 库。

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*。Upper Saddle River，NJ：Addison-Wesley，2005。

本书针对的是了解 C++的程序员，提供了 55 条规定和指南。其中一些是技术性的，如解释何时应定义复制构造函数和赋值运算符；其他一些更为通用，如对 is-a 和 has-a 关系的讨论。

- Meyers, Scott. *Effctive STL: 50 Specific Ways to Improve Your Use of the Stadard Template Library*。Reading，MA：Addison-Wesley，2001。

本书提供了选择容器和算法的指南，并讨论了使用 STL 的其他方面。

- Meyers, Scott. *More Effecitve C++: 35 New Ways to Improve Your Programs and Designs*。Reading，MA：Addison-Wesley，1996。

本书秉承了《*Effecitve C++*》的传统，对语言中的一些较模糊的问题进行了解释，介绍了实现各种目标的方法，如设计智能指针，并反映了 C++程序员在过去几年中获得的其他一些经验。

- Musser, David R, Gillmer J. Derge, and Atul Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition. Reading，MA：Addison-Wesley，2001。

要介绍并演示 STL 的功能，需要一整本书，该书可满足您的需求。

- Stroustrup, Bjarne. The C++ Programming Language. Third Edition。Reading，MA：Addison-Wesley，1997。

Stroustrup 创建了 C++，因此这是一部权威作品。不过，如果对 C++有一定的了解，将可以很容易地掌握它。它不仅介绍了语言，而且提供了多个如何使用该语言的示例，同时讨论了 OOP 方法。随着语言的发展，这本书已有多个版本，该版本增加了对标准库元素的讨论，如 STL 和字符串。

- Stroustrup, Bjarne. *The Design and Evolution of C++*。Reading，MA：Addison-Wesley，1994。

如果想了解 C++的演进过程及其为何以这种方式演进，请阅读该书。

- Vandevoorde, David and Nocoli M. Jpsittos C++Templates: The Complete Guide。Reading，MA：Addison-Wesley，2003。

有关模板的内容很多，该参考手册做了详细介绍。

H.2 网上资源

- ISO/ANSI 2011 C++标准（ISO/IEC 14882:2011）。美国国家标准化组织（ANSI）和国际标准化组织（ISO）都提供。

ANSI 提供 PDF 格式的电子版下载（售价 381 美元），这通过下述网址订购：

http://webstore.ansi.org

ISO 通过下述网址提供该文档的 PDF 文件下载和光盘，售价都是 352 瑞士法郎：

www.iso.org

价格可能有变。

- C++FAQ Lite 站点可以回答常见问题（英语、汉语、法语、俄语和葡萄牙语），它是 Cline 等编著的一本图书的删节版本。当前的网址如下：

http://www.parashift.com/C++-faq-lite

- 在下面的新闻组，可以找到有关 C++问题的比较中肯的讨论：

group:comp.lang.C++.moderated

- 使用 Google、Bing 和其他搜索引擎可找到有关特定 C++主题的信息。

附录 I　转换为 ISO 标准 C++

您可能想将一些用 C 或老式 C++版本开发的程序转换为标准 C++，本附录提供了这方面的一些指南。其中的一些内容是关于从 C 转换为 C++的，另一些是关于从老式 C++转换为标准 C++的。

I.1　使用一些预处理器编译指令的替代品

C/C++预处理器提供了一系列的编译指令。通常，C++惯例是使用这些编译指令来管理编译过程，而避免用编译指令替换代码。例如，#include 编译指令是管理程序文件的重要组件。其他编译指令（如# ifndef 和# endif）使得能够控制是否对特定的代码块进行编译。# pragma 编译指令使得能够控制编译器特定的编译选项。这些都是非常有帮助（有时是必不可少）的工具。但使用# define 编译指令时应谨慎。

I.1.1　使用 const 而不是#define 来定义常量

符号常量可提高代码的可读性和可维护性。常量名指出了其含义，如果要修改它的值，只需定义修改一次，然后重新编译即可。C 使用预处理器来创建常量的符号名称。

```
#define MAX_LENGTH 100
```

这样，预处理器将在编译之前对源代码执行文本置换，即用 100 替换所有的 MAX_LENGTH。

而 C++则在变量声明使用限定符 const：

```
const int MAX_LENGTH = 100;
```

这样 MAX_LENGTH 将被视为一个只读的 int 变量。

使用 const 的方法有很多优越性。首先，声明显式指明了类型。使用# define 时，必须在数字后加上各种后缀来指出除 char、int 或 double 之外的类型。例如，使用 100L 来表明 long 类型，使用 3.14F 来表明 float 类型。更重要的是，const 方法可以很方便地用于复合类型，如下例所示：

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

最后，const 标识符遵循变量的作用域规则，因此，可以创建作用域为全局、名称空间或数据块的常量。在特定函数中定义常量时，不必担心其定义会与程序的其他地方使用的全局常量冲突。例如，对于下面的代码：

```
#define n 5
const int dz = 12;
...
void fizzle()
{
    int n;
    int dz;
    ...
}
```

预处理器将把：

```
int n;
```

替换为：

```
int 5;
```

从而导致编译错误。而 fizzle()中定义的 dz 是本地变量。另外，必要时，fizzle()可以使用作用域解析运算符（::），以::dz 的方式访问该常量。

虽然 C++借鉴了 C 语言中的关键字 const，但 C++版本更有用。例如，对于外部 const 值，C++版本有内部链接，而不是变量和 C 中 const 所使用的默认外部链接。这意味着使用 const 的程序中的每个文件都必须定义该 const。这好像增加了工作量，但实际上，它使工作更简单。使用内部链接时，可以将 const 定义

放在工程中的各种文件使用的头文件中。对于外部链接，这将导致编译错误，但对于内部链接，情况并非如此。另外，由于 const 必须在使用它的文件中定义（在该文件使用的头文件中定义也满足这样的要求），因此可以将 const 值用作数组长度参数：

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
    loads[i] = 50;
```

这在 C 语言中是行不通的，因为定义 MAX_LENGTH 的声明可能位于一个独立的文件中，在编译时，该文件可能不可用。坦白地说，在 C 语言中，可以使用 static 限定符来创建内部链接常量。也就是说，C++通过默认使用 static，让您可以少记住一件事。

顺便说一句，修订后的 C 标准（C99）允许将 const 用作数组长度，但必须将数组作为一种新式数组——变量数组，而这不是 C++标准的一部分。

在控制何时编译头文件方面，# define 编译指令仍然很有帮助：

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// code goes here
#endif
```

但对于符号常量，习惯上还是使用 const，而不是#define。另一个好方法——尤其是在有一组相关的整型常量时——是使用 enum：

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

I.1.2 使用 inline 而不是# define 来定义小型函数

在创建类似于内联函数的东西时，传统的 C 语言方式是使用一个#define 宏定义：

```
#define Cube(X) X*X*X
```

这将导致预处理器进行文本置换，将 X 替换为 Cube()的参数：

```
y = Cube(x);       // replaced with y = x*x*x;
y = Cube(x + z++); // replaced with x + z++*x + z++*x + z++;
```

由于预处理器使用文本置换，而不是真正地传递参数，因此使用这种宏可能导致意外的、错误的结果。要避免这种错误，可以在宏中使用大量的圆括号来确保正确的运算顺序：

```
#define Cube(X) ((X)*(X)*(X))
```

但即使这样做，也无法处理使用诸如 Z++等值的情况。

C++方法是使用关键字 inline 来标识内联函数，这种方法更可靠，因为它采用的是真正的参数传递。另外，C++内联函数可以是常规函数，也可以是类方法：

```
class dormant
{
private:
    int period;
    ...
public:
    int Period() const { return period; } // automatically inline
    ...
};
```

#define 宏的一个优点是，它是无类型的，因此将其用于任何类型，运算都是有意义的。在 C++中，可以创建内联模板来使函数独立于类型，同时传递参数。

总之，请使用 C++内联技术，而不是 C 语言中的#define 宏。

I.2 使用函数原型

实际上，您没有选择的余地。虽然在 C 语言中，原型是可选的，但在 C++中，它确实是必不可少的。请注意，在使用之前定义的函数（如内联函数）是其原型。

应尽可能在函数原型和函数头中使用 const。具体地说，对于表示不可修改的数据的指针参数和引用参数，应使用 const。这不仅使编译器能够捕获修改数据的错误，也使函数更为通用。也就是说，接受 const 指针或引用的函数能够同时处理 const 数据和非 const 数据，而不使用 const 指针或引用的函数只能处理非

const 数据。

I.3　使用类型转换

Stroustrup 对 C 语言的抱怨之一是其无规律可循的类型转换运算符。确实，类型转换通常是必需的，但标准类型转换太不严格。例如，对于下面的代码：

```
struct Doof
{
    double feeb;
    double steeb;
    char sgif[10];
};
Doof leam;
short * ps = (short *) & leam; // old syntax
int * pi = int * (&leam); // new syntax
```

C 语言不能防止将一种类型的指针转换为另一种完全不相关的类型的指针。

从某种意义上看，这种情况与 goto 语句相似。goto 语句的问题太灵活了，导致代码混乱。解决方法是提供更严格的、结构化程度更高的 goto 版本，来处理需要使用 goto 语句的常见任务，诸如 for 循环、while 循环和 if else 语句等语言元素应运而生。对于类型转换不严格的问题，标准 C++提供了类似的解决方案，即用严格的类型转换来处理最常见的、需要进行类型转换的情况。下面是第 15 章介绍的类型转换运算符：

- dynamic_cast；
- static_cast；
- const_cast；
- reinterpret_cast。

因此，在执行涉及指针的类型转换时，应尽可能使用上述运算符之一。这样做不但可以指出类型转换的目的，并可以检查类型转换是否是按预期那样使用的。

I.4　熟悉 C++特性

如果使用的是 malloc()和 free()，请改用 new 和 delete；如果是使用 setjmp()和 longjmp()处理错误，则请改用 try、throw 和 catch。另外，对于表示 true 和 false 的值，应将其类型声明为 bool。

I.5　使用新的头文件

C++标准指定了头文件的新名称，请参见第 2 章。如果使用的是老式头文件，则应当改用新名称。这样做不仅仅是形式上的改变，因为新版本有时新增了特性。例如，头文件 ostream 提供了对宽字符输入和输出的支持，还提供了新的控制符，如 boolalpha 和 fixed（请参见第 17 章）。对于众多格式化选项的设置来说，这些控制符提供的接口比使用 setf()或 iomanip 函数更简单。如果确实使用的是 setf()，则在指定常量时，请使用 ios_base 而不是 ios，即使用 ios_base::fixed 而不是 ios::fixed。另外，新的头文件包含名称空间。

I.6　使用名称空间

名称空间有助于组织程序中使用的标识符，避免名称冲突。由于标准库是使用新的头文件组织实现的，它将名称放在 std 名称空间中，因此使用这些头文件需要处理名称空间。

出于简化的目的，本书的示例通常使用编译指令 using 来使 std 名称空间中的名称可用：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;  // a using-directive
```

然而，不管需要与否，都导出名称空间中的所有名称，是与名称空间的初衷背道而驰的。

稍微要好些的方法是，在函数中使用 using 编译指令，这将使名称在该函数中可用。

更好也是推荐的方法是，使用 using 声明或作用域解析运算符（::），只使程序需要的名称可用。例如，下面的代码使 cin、cout 和 end1 可用于文件的剩余部分：

```
#include <iostream>
using std::cin;      // a using-declaration
using std::cout;
using std::endl;
```

但使用作用域解析运算符只能使名称在使用该运算符的表达式中可用：

```
cout << std::fixed << x << endl;  //using the scope resolution operator
```

这样做可能很麻烦，但可以将通用的 using 声明放在一个头文件中：

```
// mynames — a header file
using std::cin;           // a using-declaration
using std::cout;
using std::endl;
```

还可以将通用的 using 声明放在一个名称空间中：

```
// mynames — a header file
#include <iostream>

namespace io
{
    using std::cin;
    using std::cout;
    using std::endl;
}

namespace formats
{
    using std::fixed;
    using std::scientific;
    using std:boolalpha;
}
```

这样，程序可以包含该文件，并使用所需的名称空间：

```
#include "mynames"
using namespace io;
```

I.7 使用智能指针

每个 new 都应与 delete 配对使用。如果使用 new 的函数由于引发异常而提前结束，将导致问题。正如第 15 章介绍的，使用 autoptr 对象跟踪 new 创建的对象将自动完成 delete 操作。C++11 新增的 unique_ptr 和 shared_ptr 提供了更佳的替代方案。

I.8 使用 string 类

传统的 C 风格字符串深受不是真正的类型之苦。可以将字符串存储在字符数组中，也可以将字符数组初始化为字符串。但不能使用赋值运算符将字符串赋给字符数组，而必须使用 strcpy()或 strncpy()。不能使用关系运算符来比较 C 风格字符串，而必须使用 strcmp()（如果忘记了这一点，使用了>运算符，将不会出现语法错误，程序将比较字符串的地址，而不是字符串的内容）。

而 string 类（参见第 16 章和附录 F）使得能够使用对象来表示字符串，并定义了赋值运算符、关系运算符和加法运算符（用于拼接）。另外，string 类还提供了自动内存管理功能，因此通常不用担心字符串被保存前，有人可能会跨越数组边界或将字符串截短。

String 类提供了许多方便的方法。例如，可以将一个 string 对象追加到另一个对象的后面，也可以将 C 风格的字符串，甚至 char 值追加到 string 对象的后面。对于接受 C 风格字符串参数的函数，可以使用 c_str()方法来返回一个适当的 char 指针。

string 类不仅提供了一组设计良好的方法来处理与字符串相关的工作（如查找子字符串），而且与 STL 兼容，因此，可以将 STL 算法用于 string 对象。

I.9　使用 STL

标准模板库（请参见第 16 章和附录 G）为许多编程需要提供了现成的解决方案，应使用它。例如，与其声明一个 double 或 string 对象数组，不如创建 vector<double>对象或 vector<string>对象。这样做的好处与使用 string 对象（而不是 C 风格字符串）相似。赋值运算符已被定义，因此可以使用赋值运算符将一个 vector 对象赋给另一个 vector 对象。可以按引用传递 vector 对象，接收这种对象的函数可以使用 size()方法来确定 vector 对象中元素数目。内置的内存管理功能使得当使用 pushback()方法在 vector 对象中添加元素时，其大小将自动调整。当然，还可以根据实际需要来使用其他有用的类方法和通用算法。在 C++11 中，如果长度固定的数组是更好的解决方案，可使用 array<double>或 array<string>。

如果需要链表、双端队列（或队列）、栈、常规队列、集合或映射，应使用 STL，它提供了有用的容器模板。算法库使得可以将矢量的内容轻松地复制到链表中，或将集合的内容同矢量进行比较。这种设计使得 STL 成为一个工具箱，它提供了基本部件，可以根据自己的需要进行装配。

在设计内容广泛的算法库时，效率是一个主要的设计目标，因此只需要完成少量的编程工作，便可以得到最好的结果。另外，实现算法时使用了迭代器的概念，这意味着这些算法不仅可用于 STL 容器。具体地说，它们也可用于传统数组。

附录 J　复习题答案

第 2 章复习题答案

1. 它们叫作函数。
2. 这将导致在最终的编译之前，使用 iostream 文件的内容替换该编译指令。
3. 它使得程序可以使用 std 名称空间中的定义。
4. `cout << "Hello, world\n";`

或

```
cout << "Hello, world" << endl;
```

5. `int cheeses;`
6. `cheeses = 32;`
7. `cin >> cheeses;`
8. `cout << "We have " << cheeses << " varieties of cheese\n";`
9. 调用函数 froop()时，应提供一个参数，该参数的类型为 double，而该函数将返回一个 int 值。例如，可以像下面这样使用它：

```
int gval = froop(3.14159);
```

函数 rattle()接受一个 int 参数且没有返回值。例如，可以这样使用它：

```
rattle(37);
```

函数 prune()不接受任何参数且返回一个 int 值。例如，可以这样使用它：

```
int residue = prune();
```

10. 当函数的返回类型为 void 时，不用在函数中使用 return。然而，如果不提供返回值，则可以使用它：

```
return;
```

第 3 章复习题答案

1. 有多种整型类型，可以根据特定需求选择最适合的类型。例如，可以使用 short 来存储空格，使用 long 来确保存储容量，也可以寻找可提高特定计算的速度的类型。
2.

```
short rbis = 80;        // or short int rbis = 80;
unsigned int q = 42110; // or unsigned q = 42110;
unsigned long ants = 3000000000;
// or long long ants = 3000000000;
```

注意：不要指望 int 变量能够存储 3000000000；另外，如果系统支持通用的列表初始化，可使用它：

```
short rbis = {80};         // = is optional
unsigned int q {42110};  // could use = {42110}
long long ants {3000000000};
```

3. C++没有提供自动防止超出整型限制的功能，可以使用头文件 climits 来确定限制情况。
4. 常量 33L 的类型为 long，常量 33 的类型为 int。
5. 这两条语句并不真正等价，虽然对于某些系统来说，它们是等效的。最重要的是，只有在使用 ASCII 码的系统上，第一条语句才将得分设置为字母 A，而第二条语句还可用于使用其他编码的系统。其次，65 是一个 int 常量，而‘A’是一个 char 常量。
6. 下面是 4 种方式：

```
char c = 88;
cout << c << endl;         // char type prints as character

cout.put(char(88));        // put() prints char as character
```

```
cout << char(88) << endl;  // new-style type cast value to char

cout << (char)88 << endl;  // old-style type cast value to char
```

7. 这个问题的答案取决于这两个类型的长度。如果 long 为 4 个字节，则没有损失。因为最大的 long 值将是 20 亿，即有 10 位数。由于 double 提供了至少 13 位有效数字，因而不需要进行任何舍入。long long 类型可提供 19 位有效数字，超过了 double 保证的 13 位有效数字。

8.
```
a. 8 * 9 + 2 is 72 + 2 is 74
b. 6 * 3 / 4 is 18 / 4 is 4
c. 3 / 4 * 6 is 0 * 6 is 0
d. 6.0 * 3 / 4 is 18.0 / 4 is 4.5
e. 15 % 4 is 3
```

9. 下面的代码都可用于完成第一项任务：

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

要将它们作为 double 类型相加，再进行转换，可采取下述方式之一：

```
int pos = (int) (x1 + x2);
int pos = int(x1 + x2);
```

10.
```
a. int
b. float
c. char
d. char32_t
e. double
```

第 4 章复习题答案

1.
```
a. char actors[30];
b. short betsie[100];
c. float chuck[13];
d. long double dipsea[64];
```

2.
```
a. array<char,30> actors;
b. array<short, 100> betsie;
c. array<float, 13> chuck;
d. array<long double, 64> dipsea;
```

3.
```
int oddly[5] = {1, 3, 5, 7, 9};
```

4.
```
int even = oddly[0] + oddly[4];
```

5.
```
cout << ideas[1] << "\n"; // or << endl;
```

6.
```
char lunch[13] = "cheeseburger";// number of characters + 1
```

或者

```
char lunch[] = "cheeseburger"; // let the compiler count elements
```

7.
```
string lunch = "Waldorf Salad";
```

如果没有 using 编译指令，则为：

```
std::string lunch = "Waldorf Salad";
```

8.
```
struct fish {
    char kind[20];
    int weight;
    float length;
};
```

9.
```
fish petes =
{
    "trout",
    12,
    26.25
};
```

10.
```
enum Response {No, Yes, Maybe};
```

11.
```
double * pd = &ted;
cout << *pd << "\n";
```

12.
```
float * pf = treacle; // or = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
            // or use *pf and *(pf + 9)
```

13. 这里假设已经包含了头文件 iostream 和 vector，并有一条 using 编译指令：

```
unsigned int size;
cout << "Enter a positive integer: ";
```

```
cin >> size;
int * dyn = new int [size];
vector<int> dv(size);
```

14. 是的，它是有效的。表达式“home of the jolly bytes”是一个字符串常量，因此，它将判定为字符串开始的地址。cout 对象将 char 地址解释为打印字符串，但类型转换(int *)将地址转换为 int 指针，然后作为地址被打印。总之，该语句打印字符串的地址，只要 int 类型足够宽，能够存储该地址。

15.
```
struct fish
{
    char kind[20];
    int weight;
    float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

16. 使用 cin >>address 将使得程序跳过空白，直到找到非空白字符为止。然后它将读取字符，直到再次遇到空白为止。因此，它将跳过数字输入后的换行符，从而避免这种问题。另一方面，它只读取一个单词，而不是整行。

17.
```
#include <string>
#include <vector>
#include <array>
const int Str_num {10}; // or = 10
...
std::vector<std::string> vstr(Str_num);
std::array<std::string, Str_num> astr;
```

第 5 章复习题答案

1. 输入条件循环在进入输入循环体之前将评估测试表达式。如果条件最初为 false，则循环不会执行其循环体。退出条件循环在处理循环体之后评估测试表达式。因此，即使测试表达式最初为 false，循环也将执行一次。for 和 while 循环都是输入条件循环，而 do while 循环是退出条件循环。

2. 它将打印下面的内容：

```
01234
```

注意，cout<< endl;不是循环体的组成部分，因为没有大括号。

3. 它将打印下面的内容：

```
0369
12
```

4. 它将打印下面的内容：

```
6
8
```

5. 它将打印下面的内容：

```
k = 8
```

6. 使用*= 运算符最简单：

```
for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
```

7. 将语句放在一对大括号中将形成一个复合语句或代码块。

8. 当然，第一条语句是有效的。表达式 1，024 由两个表达式组成——1 和 024，用逗号运算符连接。值为右侧表达式的值。这是 024，八进制为 20，因此该声明将值 20 赋给 X。第二条语句也是有效的。然而，运算符优先级将导致它被判定成这样：

```
(y = 1), 024;
```

也就是说，左侧表达式将 y 设置成 1，整个表达式的值（没有使用）为 024 或 20（八进制）。

9. cin>> ch 将跳过空格、换行符和制表符，其他两种格式将读取这些字符。

第 6 章复习题答案

1. 这两个版本将给出相同的答案，但 if else 版本的效率更高。例如，考虑当 ch 为空格时的情况。版本 1 对空格加 1，然后看它是否为换行符。这将浪费时间，因为程序已经知道 ch 为空格，因此它不是换行符。在这种情况下，版本 2 将不会查看字符是否为换行符。

2. ++ch 和 ch + 1 得到的数值相同。但++ch 的类型为 char，将作为字符打印，而 ch + 1 是 int 类型（因为将 char 和 int 相加），将作为数字打印。

3. 由于程序使用 ch = '$'，而不是 ch = ='$'，因此输入和输出将如下：

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

在第二次打印前，每个字符都被转换为$字符。另外，表达式 ch=$的值为$字符的编码，因此它是非 0 值，因而为 true;所以每次 ct2 将被加 1。

4.
```
a. weight >= 115 && weight < 125
b. ch == 'q' || ch == 'Q'
c. x % 2 == 0 && x != 26
d. x % 2 == 0 && !(x % 26 == 0)
e. donation >= 1000 && donation <= 2000 || guest == 1
f. (ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')
```

5. 不一定。例如，如果 x 为 10，则!x 为 0，!!x 为 1。然而，如果 x 为 bool 变量，则!!x 为 x。

6.
```
(x < 0)? -x : x
```

或

```
(x >= 0)? x : -x;
```

7.
```
switch (ch)
{
   case 'A': a_grade++;
             break;
   case 'B': b_grade++;
             break;
   case 'C': c_grade++;
             break;
   case 'D': d_grade++;
             break;
   default: f_grade++;
             break;
}
```

8. 如果使用整数标签，且用户输入了非整数（如 q），则程序将因为整数输入不能处理字符而挂起。但是，如果使用字符标签，而用户输入了整数（如 5），则字符输入将 5 作为字符处理。然后，switch 语句的 default 部分将提示输入另一个字符。

9. 下面是一个版本：

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
    if (ch == '\n')
        line++;
}
```

第 7 章复习题答案

1. 这 3 个步骤是定义函数、提供原型、调用函数。

2.
```
a. void igor(void);   // or void igor()
b. float tofu(int n); // or float tofu(int);
c. double mpg(double miles, double gallons);
d. long summation(long harray[], int size);
e. double doctor(const char * str);
f. void ofcourse(boss dude);
g. char * plot(map *pmap);
```

3.
```
void set_array(int arr[], int size, int value)
{
   for (int i = 0; i < size; i++)
       arr[i] = value;
}
```

4.
```
void set_array(int * begin, int * end, int value)
{
   for (int * pt = begin; pt != end; pt++)
       pt* = value;
}
```

5.
```
double biggest (const double foot[], int size)
{
    double max;
    if (size < 1)
    {
       cout << "Invalid array size of " << size << endl;
       cout << "Returning a value of 0\n";
       return 0;
}
else     // not necessary because return terminates program
{
       max = foot[0];
       for (int i = 1; i < size; i++)
          if (foot[i] > max)
              max = foot[i];
       return max;
   }
}
```

6. 将 const 限定符用于指针，以防止指向的原始数据被修改。程序传递基本类型（如 int 或 double）时，它将按值传递，以便函数使用副本。这样，原始数据将得到保护。

7. 字符串可以存储在 char 数组中，可以用带双引号的字符串来表示，也可以用指向字符串第一个字符的指针来表示。

8.
```
int replace(char * str, char c1, char c2)
{
    int count = 0;
    while (*str) // while not at end of string
    {
        if (*str == c1)
        {
            *str = c2;
            count++;
        }
        str++; // advance to next character
    }
    return count;
}
```

9. 由于 C++将“pizza”解释为其第一个元素的地址，因此使用*运算符将得到第一个元素的值，即字符 p。由于 C++将“taco”解释为第一个元素的地址，因此它将“taco”[2]解释为第二个元素的值，即字符 c。换句话来说，字符串常量的行为与数组名相同。

10. 要按值传递它，只要传递结构名 glitz 即可。要传递它的地址，请使用地址运算符&glitz。按值传递将自动保护原始数据，但这是以时间和内存为代价的。按地址传递可节省时间和内存，但不能保护原始数据，除非对函数参数使用了 const 限定符。另外，按值传递意味着可以使用常规的结构成员表示法，但传递指针则必须使用间接成员运算符。

11.
```
int judge (int (*pf)(const char *));
```

12. a. 注意，如果 ap 是一个 applicant 结构，则 ap.credit_ratings 就是一个数组名，而 ap.credit_ratings[i]是一个数组元素：

```
void display(applicant ap)
{
    cout << ap.name << endl;
    for (int i = 0; i < 3; i++)
        cout << ap.credit_ratings[i] << endl;
}
```

b. 注意，如果 pa 是一个指向 applicant 结构的指针，则 pa->credit_ratings 就是一个数组名，而 pa->credit_ratings[i]是一个数组元素：

```
void show(const applicant * pa)
{
   cout << pa->name << endl;
   for (int i = 0; i < 3; i++)
       cout << pa->credit_ratings[i] << endl;
}
```

13.
```
typedef void (*p_f1)(applicant *);
p_f1 p1 = f1;
typedef const char * (*p_f2)(const applicant *, const applicant *);
```

```
    p_f2 p2 = f2;
    p_f1 ap[5];
    p_f2 (*pa)[10];
```

第 8 章复习题答案

1. 只有一行代码的小型、非递归函数适合作为内联函数。
2. a. void song(const char * name，int times = 1);
   b. 没有。只有原型包含默认值的信息。
   c. 是的，如果保留 times 的默认值：

```
void song(char * name = "O, My Papa", int times = 1);
```

3. 可以使用字符串"\""或字符'"'来打印引号，下面的函数演示了这两种方法。

```
#include <iostream.h>
void iquote(int n)
{
    cout << "\"" << n << "\"";
}

void iquote(double x)
{
    cout << '"' << x << '"';
}

void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4. a. 该函数不应修改结构成员，所以使用 const 限定符。

```
 void show_box(const box & container)
{
    cout << "Made by " << container. maker << endl;
    cout << "Height = " << container.height << endl;
    cout << "Width = " << container.width << endl;
    cout << "Length = " << container.length << endl;
    cout << "Volume = " << container.volume << endl;
}
b. void set_volume(box & crate)
   {
      crate.volume = crate.height * crate.width * crate.length;
   }
```

5. 首先，将原型修改成下面这样：

```
// function to modify array object
void fill(std::array<double, Seasons> & pa);
// function that uses array object without modifying it
void show(const std::array<double, Seasons> & da);
```

注意，show()应使用 const，以禁止修改对象。

接下来，在 main()中，将 fill()调用改为下面这样：

```
fill(expenses);
```

函数 show()的调用不需要修改。

接下来，新的 fill()应类似于下面这样：

```
void fill(std::array<double, Seasons> & pa) // changed
{
    using namespace std;
    for (int i = 0; i < Seasons; i++)
    {
        cout << "Enter " << Snames[i] << " expenses: ";
        cin >> pa[i]; // changed
    }
}
```

注意到(*pa)[i]变成了更简单的 pa[i]。

最后，修改 show()的函数头：

```
void show(std::array<double, Seasons> & da)
```

6. a. 通过为第二个参数提供默认值：

```
double mass(double d, double v = 1.0);
```

也可以通过函数重载：

```
double mass(double d, double v);
double mass(double d);
```

b. 不能为重复的值使用默认值，因为必须从右到左提供默认值。可以使用重载：

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

c. 可以使用函数重载：

```
int average(int a, int b);
double average(double x, double y);
```

d. 不能这样做，因为两个版本的特征标将相同。

7.
```
template<class T>
T max(T t1, T t2) // or T max(const T & t1, const T & t2)
{
    return t1 > t2? t1 : t2;
}
```

8.
```
template<> box max(box b1, box b2)
{
    return b1.volume > b2.volume? b1 : b2;
}
```

9. v1 的类型为 float，v2 的类型为 float &，v3 的类型为 float &，v4 的类型为 int，v5 的类型为 double。字面值 2.0 的类型为 double，因此表达式 2.0 * m 的类型为 double。

第 9 章复习题答案

1. a. homer 将自动成为自动变量。

b. 应该在一个文件中将 secret 定义为外部变量，并在第二个文件中使用 extern 来声明它。

c. 可以在外部定义前加上关键字 static，将 topsecret 定义为一个有内部链接的静态变量。也可在一个未命名的名称空间中进行定义。

d. 应在函数中的声明前加上关键字 static，将 beencalled 定义为一个本地静态变量。

2. using 声明使得名称空间中的单个名称可用，其作用域与 using 所在的声明区域相同。using 编译指令使名称空间中的所有名称可用。使用 using 编译指令时，就像在一个包含 using 声明和名称空间本身的最小声明区域中声明了这些名称一样。

3.
```
#include <iostream>
int main()
{
    double x;
    std::cout << "Enter value: ";
    while (! (std::cin >> x) )
    {
        std::cout << "Bad input. Please enter a number: ";
        std::cin.clear();
       while (std::cin.get() != '\n')
           continue;
    }
    std::cout << "Value = " << x << std::endl;
    return 0;
}
```

4. 下面是修改后的代码：

```
#include <iostream>
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
    double x;
    cout << "Enter value: ";
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get() != '\n')
            continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5. 可以在每个文件中包含单独的静态函数定义。或者每个文件都在未命名的名称空间中定义一个合适

的 average()函数。

6.
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7.
```
1
4, 1, 2
2
2
4, 1, 2
2
```

第 10 章复习题答案

1. 类是用户定义的类型的定义。类声明指定了数据将如何存储，同时指定了用来访问和操纵这些数据的方法（类成员函数）。

2. 类表示人们可以类方法的公有接口对类对象执行的操作，这是抽象。类的数据成员可以是私有的（默认值），这意味着只能通过成员函数来访问这些数据，这是数据隐藏。实现的具体细节（如数据表示和方法的代码）都是隐藏的，这是封装。

3. 类定义了一种类型，包括如何使用它。对象是一个变量或其他数据对象（如由 new 生成的），并根据类定义被创建和使用。类和对象之间的关系同标准类型与其变量之间的关系相同。

4. 如果创建给定类的多个对象，则每个对象都有其自己的数据内存空间;但所有的对象都使用同一组成员函数（通常，方法是公有的，而数据是私有的，但这只是策略方面的问题，而不是对类的要求）。

5. 这个示例使用 char 数组来存储字符数据，但可以使用 string 类对象。

```
// #include <cstring>

// class definition
class BankAccount
{
private:
     char name[40];    // or std::string name;
     char acctnum[25]; // or std::string acctnum;
     double balance;
public:
     BankAccount(const char * client, const char * num, double bal = 0.0);
//or BankAccount(const std::string & client,
//               const std::string & num, double bal = 0.0);
     void show(void) const;
     void deposit(double cash);
     void withdraw(double cash);
};
```

6. 在创建类对象或显式调用构造函数时，类的构造函数都将被调用。当对象过期时，类的析构函数将被调用。

7. 有两种可能的解决方案（要使用 strncpy()，必须包含头文件 cstring 或 string.h;要使用 string 类，必须包含头文件 string）：

```
BankAccount::BankAccount(const char * client, const char * num, double bal)
{
    strncpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

或者：

```
BankAccount::BankAccount(const std::string & client,
                         const std::string & num, double bal)
{
    name = client;
    acctnum = num;
    balance = bal;
}
```

请记住，默认参数位于原型中，而不是函数定义中。

8. 默认构造函数是没有参数或所有参数都有默认值的构造函数。拥有默认构造函数后，可以声明对象，而不初始化它，即使已经定义了初始化构造函数。它还使得能够声明数组。

9.
```
// stock30.h
#ifndef STOCK30_H_
#define STOCK30_H_

class Stock
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();    // default constructor
    Stock(const std::string & co, long n, double pr);
    ~Stock() {} // do-nothing destructor
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    const string & co_name() const { return company; }
};
```

10. this 指针是类方法可以使用的指针，它指向用于调用方法的对象。因此，this 是对象的地址，*this 是对象本身。

第 11 章复习题答案

1. 下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
Stonewt operator*(double mult);

// definition – let constructor do the work
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2. 成员函数是类定义的一部分，通过特定的对象来调用。成员函数可以隐式访问调用对象的成员，而无需使用成员运算符。友元函数不是类的组成部分，因此被称为直接函数调用。友元函数不能隐式访问类成员，而必须将成员运算符用于作为参数传递的对象。请比较复习题 1 和复习题 4 的答案。

3. 要访问私有成员，它必须是友元，但要访问公有成员，可以不是友元。

4. 下面是类定义文件的原型和方法文件的函数定义：

```
// prototype
friend Stonewt operator*(double mult, const Stonewt & s);

// definition – let constructor do the work
Stonewt operator*(double mult, const Stonewt & s)
{
    return Stonewt(mult * s.pounds);
}
```

5. 下面的 5 个运算符不能重载：

- sizeof。
- .。
- .*。
- ::。
- ?:。

6. 这些运算符必须使用成员函数来定义。

7. 下面是一个可能的原型和定义：

```
// prototype and inline definition
operator double () {return mag;}
```

但请注意，使用 magval()方法比定义该转换函数更符合逻辑。

第 12 章复习题答案

1. a. 语法是正确的，但该构造函数没有将 str 指针初始化。该构造函数应将指针设置成 NULL 或使用 new [ ]来初始化它。

b. 该构造函数没有创建新的字符串，而只是复制了原有字符串的地址。它应当使用 new [ ]和 strcpy()。

c. 它复制了字符串，但没有给它分配存储空间，应使用 new char[len + 1]来分配适当数量的内存。

2. 首先，当这种类型的对象过期时，对象的成员指针指向的数据仍将保留在内存中，这将占用空间，同时不可访问，因为指针已经丢失。可以让类析构函数删除构造函数中 new 分配的内存，来解决这种问题。其次，析构函数释放这种内存后，如果程序将这样的对象初始化为另一个对象，则析构函数将试图释放这些内存两次。这是因为将一个对象初始化为另一个对象的默认初始化，将复制指针值，但不复制指向的数据，这将使两个指针指向相同的数据。解决方法是，定义一个复制构造函数，使初始化复制指向的数据。第三，将一个对象赋给另一个对象也将导致两个指针指向相同的数据。解决方法是重载赋值运算符，使之复制数据，而不是指针。

3. C++自动提供下面的成员函数：

- 如果没有定义构造函数，将提供默认构造函数。
- 如果没有定义复制构造函数，将提供复制构造函数。
- 如果没有定义赋值运算符，将提供赋值运算符。
- 如果没有定义析构函数，将提供默认析构函数。
- 如果没有定义地址运算符，将提供地址运算符。

默认构造函数不完成任何工作，但使得能够声明数组和未初始化的对象。默认复制构造函数和默认赋值运算符使用成员赋值。默认析构函数也不完成任何工作。隐式地址运算符返回调用对象的地址（即 this 指针的值）。

4. 应将 personality 成员声明为字符数组或 car 指针，或者将其声明为 String 对象或 string 对象。该声明没有将方法设置为公有的，因此会有几个小错误。下面是一种可能的解决方法，修改的地方以粗体显示：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char personality[40]; // provide array size
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    friend ostream & operator<<(ostream & os, const nifty & n);
};  // note closing semicolon

nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}

nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
```

```
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

下面是另一种解决方案：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char * personality; // create a pointer
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    nifty(const nifty & n);
    ~nifty() { delete [] personality; }
    nifty & operator=(const nifty & n) const;
    friend ostream & operator<<(ostream & os, const nifty & n);
};   // note closing semicolon

nifty::nifty()
{
    personality = NULL;
    talents = 0;
}

nifty::nifty(const char * s)
{
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

5. a.

```
Golfer nancy; // default constructor
Golfer lulu("Little Lulu"); // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer; // default constructor
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker"; // Golfer(const char * name, int g)
*par = nancy; // default assignment operator
nancy = "Nancy Putter";// Golfer(const char * name, int g), then
                       // the default assignment operator
```

注意，对于语句 5 和 6，有些编译器还将调用默认的赋值运算符。

b. 类应定义一个复制数据（而不是地址）的赋值运算符。

第 13 章复习题答案

1. 基类的公有成员成为派生类的公有成员。基类的保护成员成为派生类的保护成员。基类的私有成员被继承，但不能直接访问。复习题 2 的答案提供了这些通用规定的特例。

2. 不能继承构造函数、析构函数、赋值运算符和友元。

3. 如果返回的类型为 void，仍可以使用单个赋值，但不能使用连锁赋值：

```
baseDMA magazine("Pandering to Glitz", 1);
baseDMA gift1, gift2, gift3;
gift1 = magazine;       // ok
gift 2 = gift3 = gift1; // no longer valid
```

如果方法返回一个对象，而不是引用，则该方法的执行速度将有所减慢，这是因为返回语句需要复制对象。

4. 按派生的顺序调用构造函数，最早的构造函数最先调用。调用析构函数的顺序正好相反。

5. 是的，每个类都必须有自己的构造函数。如果派生类没有添加新成员，则构造函数可以为空，但必须存在。

6. 只调用派生类方法。它取代基类定义。仅当派生类没有重新定义方法或使用作用域解析运算符时，才会调用基类方法。然而，应把将所有要重新定义的函数声明为虚函数。

7. 如果派生类构造函数使用 new 或 new[ ]运算符来初始化类的指针成员，则应定义一个赋值运算符。更普遍地说，如果对于派生类成员来说，默认赋值不正确，则应定义赋值运算符。

8. 当然，可以将派生类对象的地址赋给基类指针;但只有通过显式类型转换，才可以将基类对象的地址赋给派生类指针（向下转换），而使用这样的指针不一定安全。

9. 是的，可以将派生类对象赋给基类对象。对于派生类中新增的数据成员都不会传递给基类对象。然而，程序将使用基类的赋值运算符。仅当派生类定义了转换运算符（即包含将基类引用作为唯一参数的构造函数）或使用基类为参数的赋值运算符时，相反方向的赋值才是可能的。

10. 它可以这样做，因为 C++允许基类引用指向从该基类派生而来的任何类型。

11. 按值传递对象将调用复制构造函数。由于形参是基类对象，因此将调用基类的复制构造函数。复制构造函数以基类引用为参数，该引用可以指向作为参数传递的派生对象。最终结果是，将生成一个新的基类对象，其成员对应于派生对象的基类部分。

12. 按引用（而不是按值）传递对象，这样可以确保函数从虚函数受益。另外，按引用（而不是按值）传递对象可以节省内存和时间，尤其对于大型对象。按值传递对象的主要优点在于可以保护原始数据，但可以通过将引用作为 const 类型传递，来达到同样的目的。

13. 如果 head()是一个常规方法，则 ph->head()将调用 Corporation::head();如果 head()是一个虚函数，则 ph->head()将调用 PublicCorporation::head()。

14. 首先，这种情况不符合 is-a 模型，因此公有继承不适用。其次，House 中的 area()定义隐藏了 area()的 Kitchen 版本，因为这两个方法的特征标不同。

第 14 章复习题答案

1.

Class Bear	class PolarBear	公有，北极熊是一种熊
class Kitchen	class Home	私有，家里有厨房
class Person	class Programmer	公有，程序员是一种人
class Person	class HorseAndJockey	私有，马和驯马师的组合中包含一个人
class Person class Automobile	class Driver	人是公有的，因为司机是一个人；汽车是私有的，因为司机有一辆汽车

2.
```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    fb.tell();
    cout << glip << endl;
}
```

3.
```
Gloam::Gloam(int g, const char * s)
              : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
              : glip(g), Frabjous(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    Frabjous::tell();
    cout << glip << endl;
}
```

4.
```
class Stack<Worker *>
{
private:
    enum {MAX = 10};     // constant specific to class
    Worker * items[MAX]; // holds stack items
    int top;             // index for top stack item
```

```
    public:
        Stack();
        Boolean isempty();
        Boolean isfull();
        Boolean push(const Worker * & item); // add item to stack
        Boolean pop(Worker * & item);        // pop top into item
    };
```

5.
```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

程序清单 14.18 生成 4 个模板：ArrayTP<int, 10>、ArrayTP<double, 10>、ArrayTP<int, 5>和 Array<ArrayTP <int, 5>, 10>。

6. 如果两条继承路线有相同的祖先，则类中将包含祖先成员的两个拷贝。将祖先类作为虚基类可以解决这种问题。

第 15 章复习题答案

1. a. 友元声明如下：

```
friend class clasp;
```

b. 这需要一个前向声明，以便编译器能够解释 void snip（muff&）：

```
    snip(muff &):
    class muff; // forward declaration
    class cuff {
    public:
        void snip(muff &) { ... }
        ...
    };
    class muff {
        friend void cuff::snip(muff &);
        ...
};
```

c. 首先，cuff 类声明应在 muff 类之前，以便编译器可以理解 cuff::snip()。其次，编译器需要 muff 的一个前向声明，以便可以理解 snip(muff &)。

```
    class muff; // forward declaration
    class cuff {
    public:
        void snip(muff &) { ... }
        ...
    };
    class muff {
        friend void cuff::snip(muff &);
        ...
};
```

2. 不。为使类 A 拥有一个本身为类 B 的成员函数的友元，B 的声明必须位于 A 的声明之前。一个前向声明是不够的，因为这种声明可以告诉 A：B 是一个类;但它不能指出类成员的名称。同样，如果 B 拥有一个本身是 A 的成员函数的友元，则 A 的这个声明必须位于 B 的声明之前。这两个要求是互斥的。

3. 访问类的唯一方法是通过其有接口，这意味着对于 Sauce 对象，只能调用构造函数来创建一个。其他成员（soy 和 sugar）在默认情况下是私有的。

4. 假设函数 f1()调用函数 f2()。f2()中的返回语句导致程序执行在函数 f1()中调用函数 f2()后面的一条语句。throw 语句导致程序沿函数调用的当前序列回溯，直到找到直接或间接包含对 f2()的调用的 try 语句块为止。它可能在 f1()中、调用 f1()的函数中或其他函数中。找到这样的 try 语句块后，将执行下一个匹配的 catch 语句块，而不是函数调用后的语句。

5. 应按从子孙到祖先的顺序排列 catch 语句块。

6. 对于示例#1，如果 pg 指向一个 Superb 对象或从 Superb 派生而来的任何类的对象，则 if 条件为 true。具体地说，如果 pg 指向 Magnificent 对象，则 if 条件也为 true。对于示例#2，仅当指向 Superb 对象时，if 条件才为 true，如果指向的是从 Superb 派生出来的对象，则 if 条件不为 true。

7. Dynamic_cast 运算符只允许沿类层次结构向上转换，而 static_cast 运算符允许向上转换和向下转换。static_cast 运算符还允许枚举类型和整型之间以及数值类型之间的转换。

第 16 章复习题答案

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
    string st; // a string object
public:
    RQ1() : st("") {}
    RQ1(const char * s) : st(s) {}
    ~RQ1() {};
// more stuff
};
```

不再需要显式复制构造函数、析构程序和赋值运算符，因为 string 对象提供了自己的内存管理功能。

2. 可以将一个 string 对象赋给另一个。string 对象提供了自己的内存管理功能，所以一般不需要担心字符串超出存储容量。

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia= new int[20]; // wrong, use with new, not new[]
auto_ptr<string>(new string);   // wrong, no name for pointer
int rigue = 7;
auto_ptr<int>(&rigue);           // wrong, memory not allocated by new
auto_ptr dbl (new double);       // wrong, omits <double>
```

5. 栈的 LIFO 特征意味着可能必须在到达所需要的球棍（club）之前删除很多球棍。

6. 集合将只存储每个值的一个拷贝，因此，5 个 5 分将被存储为 1 个 5 分。

7. 使用迭代器使得能够使用接口类似于指针的对象遍历不以数组方式组织的数据，如双向链表中的数据。

8. STL 方法使得可以将 STL 函数用于指向常规数组的常规指针以及指向 STL 容器类的迭代器，因此提高了通用性。

9. 可以将一个 vector 对象赋给另一个。vector 管理自己的内存，因此可以将元素插入到矢量中，并让它自动调整长度。使用 at()方法，可以自动检查边界。

10. 这两个 vector 函数和 random_shuffle()函数要求随机访问迭代器，而 list 对象只有双向迭代器。可以使用 list 模板类的 sort()成员函数（参见附录 G），而不是通用函数来排序，但没有与 random_shuffle()等效的成员函数。然而，可以将链表复制到矢量中，然后打乱矢量，并将结果重新复制到链表中。

第 17 章复习题答案

1. iostream 文件定义了用于管理输入和输出的类、常量和操纵符，这些对象管理用于处理 I/O 的流和缓冲区。该文件还创建了一些标准对象（cin、cout、cerr 和 clog 以及对应的宽字符对象），用于处理与每个程序相连的标准输入和输出流。

2. 键盘输入生成一系列字符。输入 121 将生成 3 个字符，每个字符都由一个 1 字节的二进制码表示。要将这个值存储为 int 类型，则必须将这 3 个字符转换为 121 值的二进制表示。

3. 在默认情况下，标准输出和标准错误都将输出发送给标准输出设备（通常为显示器）。然而，如果要求操作系统将输出重定向到文件，则标准输出将与文件（而不是显示器）相连，但标准错误仍与显示器相连。

4. ostream 类为每种 C++基本类型定义了一个 operator <<()函数的版本。编译器将下面的表达式：

```
cout << spot
```

解释为：

```
cout.operator<<(spot)
```

这样，它便能够将该方法调用与具有相同参数类型的函数原型匹配。

5. 可以将返回 ostream &类型的输出方法拼接。这样，通过一个对象调用方法时，将返回该对象。然后，返回对象将可以调用序列中的下一个方法。

6.
```
//rq17-6.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << "Enter an integer: ";
    int n;
    cin >> n;
    cout << setw(15) << "base ten" << setw(15)
           << "base sixteen" << setw(15) << "base eight" << "\n";
    cout.setf(ios::showbase); // or cout << showbase;
    cout << setw(15) << n << hex << setw(15) << n
           << oct << setw(15) << n << "\n";
    return 0;
}
```

7.
```
//rq17-7.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
// or cout << showpoint << fixed << right;
    cout << "First format:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    return 0;
}
```

8. 下面是输出：

```
ct1 = 5; ct2 = 9
```

该程序的前半部分忽略空格和换行符，而后半部分没有。注意，程序的后半部分从第一个 q 后面的换行符开始读取，将换行符计算在内。

9. 如果输入行超过 80 个字符，ignore()将不能正常工作。在这种情况下，它将跳过前 80 个字符。

第 18 章复习题答案

1.
```
class Z200
{
private:
    int j;
    char ch;
    double z;
public:
```

```
    Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
...
};

double x {8.8}; // or = {8.8}
std::string s {"What a bracing effect!"};
int k{99};
Z200 zip{200,'Z',0.67});
std::vector<int> ai {3, 9, 4, 7, 1};
```

2. r1(w)合法，形参 rx 指向 w。

r1(w+1)合法，形参 rx 指向一个临时变量，这个变量被初始化为 w+1。

r1(up(w))合法，形参 rx 指向一个临时变量，这个变量被初始化为 up(w)的返回值。

一般而言，将左值传递给 const 左值引用参数时，参数将被初始化为左值。将右值传递给函数时，const 左值引用参数将指向右值的临时拷贝。

r2(w)合法，形参 rx 指向 w。

r2(w+1)非法，因为 w+1 是一个右值。

r2(up(w))非法，因为 up(w)的返回值是一个右值。

一般而言，将左值传递给非 const 左值引用参数时，参数将被初始化为左值；但非 const 左值形参不能接受右值实参。

r3(w)非法，因为右值引用不能指向左值（如 w）。

r3(w+1)合法，rx 指向表达式 w+1 的临时拷贝。

r3(up(w))合法，rx 指向 up(w)的临时返回值。

3. a.
```
double & rx
const double & rx
const double & rx
```

非 const 左值引用与左值实参 w 匹配。其他两个实参为右值，const 左值引用可指向它们的拷贝。

b.
```
double & rx
double && rx
double && rx
```

左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

c.
```
const double & rx
double && rx
double && rx
```

const 左值引用与左值实参 w 匹配，而右值引用与两个右值实参匹配。

总之，非 const 左值形参与左值实参匹配，非 const 右值形参与右值实参匹配；const 左值形参可与左值或右值形参匹配，但编译器优先选择前两种方式（如果可供选择的话）。

4. 它们是默认构造函数、复制构造函数、移动构造函数、析构函数、复制赋值运算符和移动赋值运算符。这些函数之所以特殊，是因为编译器将根据情况自动提供它们的默认版本。

5. 在转让数据所有权（而不是复制数据）可行时，可使用移动构造函数，但对于标准数组，没有转让其所有权的机制。如果 Fizzle 使用指针和动态内存分配，则可将数据的地址赋给新指针，以转让其所有权。

6.
```
#include <iostream>
#include <algorithm>
template<typename T>
  void show2(double x, T fp) {std::cout << x << " -> " << fp(x) << '\n';}
int main()
{
    show2(18.0, [](double x){return 1.8*x + 32;});
    return 0;
}
```

7.
```
#include <iostream>
#include <array>
#include <algorithm>
const int Size = 5;
template<typename T>
  void sum(std::array<double,Size> a, T& fp);
int main()
```

```
{
    double total = 0.0;
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c, [&total](double w){total += w;});
    std::cout << "total: " << total << '\n';
    std::cin.get();
    return 0;
}
template<typename T>
  void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
           fp(*pt);
    }
}
```

§2. 基础知识

2.1. 计算机的组成

CPU = Central Processing Unit

★ 程序：让CPU(运算器和控制器)工作的一串指令

- 高级语言：人能理解的程序
- 机器语言：CPU能理解并执行的程序

★ 高级语言程序设计：用高级语言编写程序的一套语法和规则

§2. 基础知识

2.2. 二进制

2.2.1. 进制的基本概念

★ 进制：按进位原则进行记数的方法叫做进位记数制，简称为进制

十进制　　：0-9，逢十进一

二十四进制：0-23，逢二十四进一

六十进制　：0-59，逢六十进一

=> N进制，0 ~ N-1，逢N进一

★ 基数：指各种位制中允许选用基本数码的个数

十进制　　：0-9　=> 基数为10

二十四进制：0-23 => 基数为24

六十进制　：0-59 => 基数为60

★ 位权：在N进制数中，每个数码所表示的数值等于该数码乘以一个与数码所在位置相关的常数，这个常数叫做位权。

其大小是以基数为底、数码所在位置的序号为指数的整数次幂

$215 = 2*10^2 + 1*10^1 *5*10^0$

2的位权是10^2（百位，基数10为底，序号为2）

1的位权是10^1（十位，基数10为底，序号为1）

5的位权是10^0（个位，基数10为底，序号为0）

§2. 基础知识

2.2. 二进制

2.2.2. 二进制的基本概念

★ 二进制：基数为2，只有0、1两个数码，逢二进一

★ 二进制是计算机内部表示数据的方法，因为计算机就其本身来说是一个电器设备，为了能够快速存储/处理/传递信息，其内部采用了大量电子元件；在这些电子元件中，电路的通和断、电压的高与低，这两种状态最容易实现，也最稳定、也最容易实现对电路本身的控制。我们将计算机所能表示这样的状态，用0、1来表示、即用二进制数表示计算机内部的所有运算和操作

★ 位与字节：每个二进制位只能表示0/1两种状态，当表示较大数据时，所用位数就比较长，为便于管理，每8位称为一个字节

(位：bit 字节：byte)

8 bit = 1 byte

bit ：计算机内表示数据的最小单位

byte ：计算机表示数据的基本单位（数据表示为1-n个byte）

§2. 基础知识

2.2.2. 二进制的基本概念

★ 位与字节：每个二进制位只能表示0/1两种状态，当表示较大数据时，所用位数就比较长，为便于管理，每8位称为一个字节

(位：bit 字节：byte)

8 bit = 1 byte

bit ：计算机内表示数据的最小单位

byte ：计算机表示数据的基本单位（数据表示为1-n个byte）

● 二进制的大数表示及不同单位的换算如表所示

二进制大数的表示单位：

2^{10}	= 1024	=1 KB	(KiloByte)
2^{20}	= 1024 KB	=1 MB	(MegaByte)
2^{30}	= 1024 MB	=1 GB	(GigaByte)
2^{40}	= 1024 GB	=1 TB	(TeraByte)
2^{50}	= 1024 TB	=1 PB	(PeraByte)
2^{60}	= 1024 PB	=1 EB	(ExaByte)
2^{70}	= 1024 EB	=1 ZB	(ZettaByte)
2^{80}	= 1024 ZB	=1 YB	(YottaByte)
2^{90}	= 1024 YB	=1 BB	(BrontoByte)
2^{100}	= 1024 BB	=1 NB	(NonaByte)
2^{110}	= 1024 NB	=1 DB	(DoggaByte)
2^{120}	= 1024 DB	=1 CB	(CorydonByte)

● 简写的时候，b=bit/B=byte

例：某智能手机，存储空间 128GB
某智能电视，存储空间 128Gb

● 实际表述中，K/M/G既可以表示$2^{10}/2^{20}/2^{30}$，也可以是$10^3/10^6/10^9$，因此部分表述有二义性，折算时有误差，要根据语境理解（计算机内一般按二进制理解，其余十进制）

例：某程序猿工资：18K = 18000
宽带速率：20Mbps = 20 x 10^6 (bit per second)
笔记本内存：8GB = 8 x 2^{30}
硬盘：1TB = 1 x 10^{12} （厂商标注）
= 1 x 2^{40} （计算机理解）

⇨ 约9%误差

§2. 基础知识

2.2. 二进制

2.2.2. 二进制的基本概念

★ 计算机内数据的表示：因为采用二进制，只有两个数码（0/1），任何复杂的数据都是由0/1的基本表示组成的

- 数字(有数值含义的数字序列)
- 文本(包括数码，即无数值含义的数字序列)
- 静态图像
- 动态视频
- 声音

课上会有简单的介绍，了解即可；
听不懂也不影响后续学习

§2. 基础知识

2.2.3. 二进制与十进制的互相转换

★ 数制转换：计算机内部采用二进制，但用计算机解决实际问题时，数据的输入输出通常使用十进制(人易于理解)。因此，使用计算机进行数据处理时必须先将十进制转换为二进制，处理完成后再将二进制转换为十进制，这种将数字由一种数制转换成另一种数制的方法称为数制转换

★ 十进制转二进制(整数)：除2取余法

基本方法：
(1) 要转换的整数除以2，得商和余数(整除)
(2) 继续用商除以2，得新的商和余数(整除)
(3) 重复(2)直到商为零时为止
(4) 把所有余数逆序排列，即为转换的二进制数

$(25)_{10} = (11001)_2$

★ 二进制转十进制(整数)：按权相加法

基本方法：
(1) 把二进制数写成加权系数展开式
(2) 按十进制加法规则求和

$$11001 = 1x2^4 + 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0$$
$$= 16 + 8 + 0 + 0 + 1$$
$$= 25$$

★ 暂不考虑负数(后续讲)

★ 二进制/十进制小数的相互转换

§2. 基础知识

2.2.4. 八进制、十六进制

★ 八进制：基数为8，数码为0-7，逢八进一

★ 十六进制：基数为16，数码为0-15，逢十六进一，为避免歧义，用A-F(a-f)替代10-15表示

★ 十进制转八、十六进制：除8/16取余法

8	11418	
8	1427	2
8	178	3
8	22	2
8	2	6
	0	2

$(11418)_{10} = (26232)_8$

16	11418	
16	713	10
16	44	9
16	2	12
	0	2

$(11418)_{10} = (2C9A)_{16}$

★ 八、十六进制转十进制：按权相加法

$$(26232)_8 = 2x8^4 + 6x8^3 + 2x8^2 + 3x8^1 + 2x8^0$$
$$= 2x4096 + 6x512 + 2x64 + 3x8 + 2x1$$
$$= 11418$$

$$(2C9A)_{16} = 2x16^3 + 12x16^2 + 9x16^1 + 10x16^0$$
$$= 2x4096 + 12x256 + 9x16 + 10x1$$
$$= 11418$$

§2. 基础知识

2.2.4. 八进制、十六进制

★ 二进制转八、十六进制：低位开始，每3/4位转1位

$(1011100101011)_2 = 1\ 011\ 100\ 101\ 011 = (13453)_8$

$(1011100101011)_2 = 1\ 0111\ 0010\ 1011 = (172B)_{16}$

★ 八、十六进制转二进制：每1位转3/4位，不足补0

$(13453)_8 = 001\ 011\ 100\ 101\ 011 = (1011100101011)_2$

$(172B)_{16} = 0001\ 0111\ 0010\ 1011 = (1011100101011)_2$

★ 暂不考虑负数(后续讲)

★ 八进制和十六进制的相互转换

§2. 基础知识

2.3. C++程序简介

2.3.1. 最简单的C++程序

例1

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World." << endl;
    return 0;
}
```

功能：在屏幕上输出"Hello, World."

例2

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,sum;
    cin >> a >> b;
    sum=a+b;
    cout << "a+b=" << sum << endl;
    return 0;
}
```

功能：从键盘上输入两个整数(第1行，空格分开)
在屏幕上输出和(第2行)

例3

```
#include <iostream>
using namespace std;
int my_max(int x, int y)
{
    int z;
    if (x>y)
        z=x;
    else
         z=y;
    return z;
}
int main()
{
    int a,b,m;
    cin >> a >> b;
    m = my_max(a,b);
    cout << "max=" << m << '\n';
    return 0;
}
```

功能：从键盘上输入两个整数(第1行)
在屏幕上输出大的那个(第2行)

§2. 基础知识

2.3. C++程序简介

2.3.2. 程序结构的基本形式

包含的头文件 →
命名空间 →

暂 常量定义
函数的定义
无 全局变量的定义

函数1 →
...
...
函数n

```
#include <iostream>
using namespace std;

int main()
{
    int a,b,sum;
    cin >> a >> b;
    sum=a+b;
    cout<<"a+b="<<sum<< endl;
    return 0;
}
```

★ C++程序由函数组成，函数是C++程序的基本单位 →

例 3

```
#include <iostream>
using namespace std;

int my_max(int x, int y)
{
    int z;
    if (x>y)
       z=x;
    else
        z=y;
    return (z);
}
```

函数my_max

```
int main()
{
    int a,b,m;
    cin >> a >> b;
    m = my_max(a,b);
    cout<<"max="<<m<<'\n';
    return 0;
}
```

函数main

★ 一个C++程序可由若干源程序文件（*.cpp）组成，每个源程序文件可以包含若干函数 →

通过工程文件将多个源程序文件集合在一个程序中

★ 有且仅有一个名为main()的函数，称为主函数，程序的执行从它开始

★ C++提供许多库函数

§2. 基础知识

2.3. C++程序简介

2.3.3. 函数的组成

函数返回类型　函数名（形式参数表）

{

函数体（局部变量的定义、函数体的可执行部分）

}

```
int my_max(int x, int y)
{
    int z;
    if (x>y) z=x;
      else z=y;
    return (z);
}
```

```
#include <iostream>
using namespace std;
int my_max(int x, int y)
{ ...
}
int main()
{ ...
}
```

```
#include <iostream>
using namespace std;

//此处需补函数my_max的声明(暂略)
int main()
{ ...
}
int my_max(int x, int y)
{ ...
}
```

★ 从main()开始执行，函数相互间的位置不影响程序的正确性

§2. 基础知识

2.3. C++程序简介

2.3.3. 函数的组成

★ 从main()开始执行，函数相互间的位置不影响程序的正确性

★ 函数平行定义，嵌套调用

★ 函数由语句组成，一个语句以;结尾（必须有），语句分为定义语句和执行语句，定义语句用于声明某些信息，执行语句用于完成特定的操作

★ 书写格式自由，可以一行多个语句，也可以一个语句多行（以\表示分行）

例3

```
#include <iostream>
using namespace std;
int my_max(int x, int y)
{
    int z;
    if (x>y) z=x;
    else z=y;
    return (z);
}

int main()
{
    int a,b,m;
    cin >> a >> b;
    m = my_max(a,b);
    cout<<"max="<<m<<'\n';
    return 0;
}
```

定义my_max

调用my-max

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World." << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{ cout << "Hello, World." << endl; return 0;}
```

```
#include <iostream>
using namespace std;
int main()
{   cout    \
    <<      \
    "Hello, World." \
    << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{   cout
    <<
    "Hello, World."
    << endl;
    return 0;
}
```

同一个程序，这些格式编译器都认为是正确的

§2. 基础知识

2.3. C++程序简介

2.3.3. 函数的组成

★ 书写格式自由，可以一行多个语句，也可以一个语句多行（以\表示分行）

- 不同书写格式，编译器都正确，但对人的阅读而言，友好程度不同
- 建议：一句一行（本课程强制要求：一句一行）

  => 违反格式规定的，本题分数扣20%

✓
`#include <iostream>`
`using namespace std;`
`int main()`
`{`
`  cout << "Hello, World." << endl;`
`  return 0;`
`}`
	✗
	`#include <iostream>`
	`using namespace std;`
	`int main()`
	`{ cout << "Hello, World." << endl;return 0;}`
		✓
		`int max(int x, int y)`
		`{`
		`  int z;`
		`  if (x>y)`
		`  z=x;`
		`  else`
		`  z=y;`
		`  return z;`
		`}`
			✗
			`int max(int x, int y)`
			`{`
			`  int z;`
			`  if (x>y) z=x;`
			`  else z=y;`
			`  return z;`
			`}`

✗
`#include <iostream>`
`using namespace std;`
`int main()`
`{`
`  cout  \`
`  <<  \`
`  "Hello, World." \`
`  << endl;`
`  return 0;`
`}`
	✗
	`#include <iostream>`
	`using namespace std;`
	`int main()`
	`{`
	`  cout`
	`  <<`
	`  "Hello, World."`
	`  << endl;`
	`  return 0;`
	`}`
		✓
		`int max(int x, int y)`
		`{`
		`  return 0;`
		`}`
			✗
			`int max(int x, int y) {`
			`  return 0;`
			`}`

§2. 基础知识

2.3. C++程序简介

2.3.3. 函数的组成

★ 可以用/* … */(多行)或 //…(单行)两种方式加入注释，注释中的内容是为了增加程序的可读性，因此不需要符合C++的语法及规定

```
#include <iostream>
using namespace std;     //命名空间

int main()               /* function main */
{
    cout << "Hello, World." << endl; //Ausgabe(德文-输出)
    return 0;            /* fanhui :-) ^-^ */
}
```

```
#include <iostream>
using namespace std;     //命名空间
/* 下面是主函数，仅用于输出一个字符串，
   输出后程序即运行结束    */

int main()               /* function main */
{
    cout << "Hello, World." << endl;
    return 0;            /* fanhui :-) ^-^ */
}
```

★ 系统提供的库函数和自己编写的函数调用方法相同

§2.基础知识

2.4.编译器的安装与使用、编程的基本步骤

2.4.1.VS2022的安装与基本使用

2.4.2.DevC++的安装与基本使用

2.4.3.编程的基本过程

§2. 基础知识

2.5. C++的数据类型

2.5.1. 整型数的符号位

问题：如果用2字节（16bit）表示一个整数，则数据范围为多少？

00000000 00000000 = 0

11111111 11111111 = $2^{16}-1$ = 65535

答：数据范围为0-65535

=> 这种表示方法不考虑负数，称为无符号数

续问：如何能同时表示正负数（称为有符号数），数据范围是多少？

前提：计算机内任何数据都表示为0/1，包括正负号，假设0/1表示+/-

方案1：单独用额外的一个bit表示+/-，再加16bit数据

例：0 00000000 00000001 = +1 （max = +65535）

1 00000000 00000001 = -1 （min = -65535）

=> 无法用两字节表示正负数（放弃）

方案2：最高(最左)bit位表示+/-，再加15bit数据

例： 00000000 00000001 = +1 （max = +32767）

10000000 00000001 = -1 （min = -32767）

=> 可用两字节表示正负数，但数据范围减半（选）

§2. 基础知识

2.5. C++的数据类型

2.5.1. 整型数的符号位

★ 数据的二进制原码

将表示一个整型数的若干字节的最高bit位(最左)的0/1理解为正负号，（该bit位称为符号位），后续的所有bit理解为数值的方式称为数据的二进制原码表示

00000000 00000001 = +1
01111111 11111111 = +32767
10000000 00000001 = -1
11111111 11111111 = -32767

结论：原码不适合在计算机内表示整数

● 原码的缺陷1：+0与-0的二义性问题

10000000 00000000 = + 0
00000000 00000000 = - 0

计算机思维方式的引申：

计算机内部如何判断两个数据相等？

方法1：对应字节完全一致(选)

=> 缺陷1需要解决

方法2：某些数据做特判(放弃)

● 原码的缺陷2：含负数的情况下无法正常运算

仅含正数运算：1 + 1

二进制	十进制
00000000 00000001	1
+) 1	+) 1
00000000 00000010 (+2)	2

含负数运算：-1 + 1

二进制	十进制
10000000 00000001	-1
+) 1	+) 1
10000000 00000010 (-2)	0

§2.基础知识

2.5.C++的数据类型

2.5.2.补码的基本概念

★ 补码：解决±0的二义性及含负数运算的问题，是计算机内整型数值的真正表示方法

- 正数与原码相同
- 负数：绝对值的原码取反+1
  - Step1：求绝对值的原码
  - Step2：绝对值的原码取反
  - Step3：取反后再+1

例：以2字节（16bit）整数为例

数值	绝对值的二进制表示	原码	补码
100	1100100	0000000001100100	0000000001100100
-10	1010（绝对值）	0000000000001010	1111111111110101 +) 1 ---------- 1111111111110110
0	0(正) 0(负)	0000000000000000 0000000000000000	0000000000000000 1111111111111111 +) 1 ---------- 1 0000000000000000 （高位溢出，舍去）
-1	1（绝对值）	0000000000000001	1111111111111110 +) 1 ---------- 1111111111111111
-32768	1000000000000000 （1表示2^{15}）	1000000000000000	0111111111111111 +) 1 ---------- 1000000000000000
-32767	0111111111111111	0111111111111111	1000000000000000 +) 1 ---------- 1000000000000001

原码方式的 -32767

原码方式的 -0

原码方式的 -1

问题1：计算机的整数用原码表示，负数用补码表示，该说法是否正确？

问题2：现在看到一个以1开始的二进制整数(1***)，应该是无符号的正数，还是有符号的负数？

0 0000000 00000000
0 1111111 11111111 原码理解：+0 ~ + 32767
补码理解：+0 ~ + 32767
==========
1 0000000 00000000
1 1111111 11111111 原码理解：-0 ~ - 32767
补码理解：-1 ~ - 32768
=> 2字节（16bit）的整数范围为 -32768 ~ +32767
-2^{15} ~ $+2^{15}$

§2. 基础知识

2.5. C++的数据类型

2.5.3. 数据类型分类

- 数据类型
  - 基本类型
    - 整型
      - 短整型 (short int)
      - 整型 (int)
      - 长整型 (long int)
      - 长长整型(long long int)
    - 字符型(char)
    - 浮点型
      - 单精度型 (float)
      - 双精度型 (double)
      - 长双精度型(long double)
    - 布尔型(bool)
  - 派生类型
    - 指针类型 (*)
    - 枚举类型 (enum)
    - 数组类型 ([])
    - 结构体类型(struct)
    - 共用体类型(union)
    - 类类型 (class)
  - 空类型(void)

每种int都有signed与unsigned(包括char型)

可以用基本数据类型来构造派生类型

§2. 基础知识

2.5. C++的数据类型

2.5.4. 各种数据类型所占字节及表示范围(以VS2022 x86/32bit为基准)

类型	类型标识符	字节	数值范围	数值范围
整型	[signed] int	4	-2147483648 ～ +2147483647	-2^{31} ～ $+2^{31}-1$
无符号整型	unsigned [int]	4	0 ～ 4294967295	0 ～ $+2^{32}-1$
短整型	short [int]	2	-32768 ～ +32767	-2^{15} ～ $+2^{15}-1$
无符号短整型	unsigned short [int]	2	0 ～ 65535	0 ～ $+2^{16}-1$
长整型	long [int]	4	-2147483648 ～ +2147483647	-2^{31} ～ $+2^{31}-1$
无符号长整型	unsigned long [int]	4	0 ～ 4294967295	0 ～ $+2^{32}-1$
长长整型	long long [int]	8	-9223372036854775808 ～ 9223372036854775807	-2^{63} ～ $+2^{63}-1$
无符号长长整型	unsigned long long [int]	8	0 ～ 18446744073709551616	0 ～ $+2^{64}-1$
字符型	[signed] char	1	-128 ～ +127	-2^{7} ～ $+2^{7}-1$
无符号字符型	unsigned char	1	0 ～ +255	0 ～ $+2^{8}-1$
单精度型	float	4	-3.4×10^{38} ～ 3.4×10^{38}	-3.4×10^{38} ～ 3.4×10^{38}
双精度型	double	8	-1.7×10^{308} ～ 1.7×10^{308}	-1.7×10^{308} ～ 1.7×10^{308}
长双精度型	long double	8	-1.7×10^{308} ～ 1.7×10^{308}	-1.7×10^{308} ～ 1.7×10^{308}

[]表示可省略

例：signed int, int 等价

long, long int 等价

§2. 基础知识

2.5. C++的数据类型

2.5.4. 各种数据类型所占字节及表示范围(以VS2022 x86/32bit为基准)

★ 如何确定数据类型所占的字节：sizeof(类型)

```
#include <iostream>
using namespace std;

int main()
{
    cout << sizeof(int)                << endl;
    cout << sizeof(unsigned int)       << endl;
    cout << sizeof(long)               << endl;
    cout << sizeof(unsigned short)     << endl;
    cout << sizeof(unsigned long long) << endl;
    cout << sizeof(float)              << endl;
    cout << sizeof(long double)        << endl;

    return 0;
}
```

用VS2022的32位编译器
64位编译器
DevC++的32位编译器
64位编译器
分别编译运行并观察结果

注意：VS2022和DevC++如何
切换32位和64位编译器？

§2. 基础知识

2.5. C++的数据类型

2.5.4. 各种数据类型所占字节及表示范围(以VS2022 x86/32bit为基准)

★ 如何确定数据类型的上下限：

```
#include <iostream>
#include <climits>
using namespace std;

int main()
{
    cout << INT_MIN   << endl;
    cout << UINT_MAX  << endl;
    cout << LLONG_MAX << endl;
    cout << SHRT_MAX  << endl;

    return 0;
}
```

方法1：根据sizeof的结果自行推算

方法2：打印系统预定义的值

- 可能需要包含climits头文件
  (目前所用双编译器均不需要)

思考题

前提条件：

1、不准用INT_MIN等预置值(或不知道)

2、知道sizeof()

3、不准用其它工具(书/网络/计算器)

问题：想知道long long型(或其它)数据的最大值，怎么办？

§2. 基础知识

2.5. C++的数据类型

2.5.4. 各种数据类型所占字节及表示范围(以VS2022 x86/32bit为基准)

★ 对于整型数(含char)，均有signed及unsigned的区别，缺省为signed

★ 不同的编译系统中，不同数据类型的所占字节/表示范围可能不同

- 本课程中若不加以特别说明，均认为：

  short　　：2字节
  int/long　：4字节
  long long　：8字节
  long double ：8字节

- 所有C/C++编译器中，不同整型数据的相互关系遵循的基本准则

  以int型数据为基准大小，short不大于int，long不小于int，longlong不小于long

★ 因为数据必须占用一定字节，因此计算机不可能表示数学中的无穷概念

(可以有特殊定义表示无穷大，但无法体现增量)

★ 对于整型数，存储为二进制数形式，占满对应字节长度

```
例：85(十进制) = 1010101(二进制)
则：int型  ：00000000 00000000 00000000 01010101

    long型 ：00000000 00000000 00000000 01010101

    short型：                   00000000 01010101

    char型 ：                            01010101
```

★ 浮点型数有有效位数的限定，可能存在一定的误差

例：	short	int	long	long long
Turbo C（1987年）	2	2	4	无
VS2022	2	4	4	8
RockyLinux	2	4	8	8

§2. 基础知识

2.6. 常量

2.6.1. 基本概念

常量：在程序运行过程中值不能改变的量称为常量

- 字面常量(直接常量)：直接字面形式表示
- 符号常量：通过标识符表示

2.6.2. 数值常量

2.6.2.1. 整型常量

整型常量在C/C++源程序中的四种表示方法：

- 十进制：　正常方式
- 二进制：　0b+0~1
- 八进制：　0+0~7
- 十六进制：　0x/0X+0~9, A-F, a-f

123　　10进制表示 0b1111011　　2进制表示 0173　　8进制表示 0x7B　　16进制表示 四个值相等，都是10进制的123	请把下列三个数从大到小排列： (1) 123 (2) 0123 (3) 0x123　　?

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.1. 整型常量

整型常量在C/C++源程序中的四种表示方法：

- 十进制：　正常方式
- 二进制：　0b+0~1
- 八进制：　0+0~7
- 十六进制：　0x/0X+0~9, A-F, a-f

内部表示
输出显示
额外修饰
} 注意区分

```
#include <iostream>
using namespace std;
int main()
{
    cout << 119 << endl;
    cout << 0b1110111 << endl;
    cout << 0167 << endl;
    cout << 0x77 << endl;

    return 0;
}
```

输出？

```
#include <iostream>
using namespace std;
int main()
{
    cout << hex << 119 << endl;
    cout << hex << 0b1110111 << endl;
    cout << hex << 0167 << endl;
    cout << hex << 0x77 << endl;

    return 0;
}
```

输出？

```
#include <iostream>
using namespace std;
int main()
{
    cout << dec << 119 << endl;
    cout << "0x" << hex << 0b1110111 << endl;
    cout << "(" << oct << 0167 << ")8" << endl;

    return 0;
}
```

输出？

三个例题的结论：

★ 无论在源程序表示为何种进制，在内存中只有二进制补码一种

★ 无论在源程序表示为何种进制，和输出无关，输出缺省为十进制，可加前导进制转换(hex/oct/dec)改为其它进制

★ 输出只负责最基本的内容(可能引发二义性)，其余需要自行按需添加

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.1. 整型常量

整型常量在C/C++源程序中的四种表示方法：

- 十进制：　正常方式
- 二进制：　0b+0~1
- 八进制：　0+0~7
- 十六进制：　0x/0X+0~9, A-F, a-f

★ 无直接的二进制输出方法

★ 老版本编译器无源程序中的二进制表示方法

★ 二进制方式表示不方便，今后不再讨论(以后也可能会说三种表示方法)

★ 整型常量缺省是int型，long型通常加l(L)表示，unsigned通常加u(U)，short型无特殊后缀

下列整型常量分别是什么类型？

123　：

123L ：

123U ：

123UL：

123LU：

?

另：L不建议小写，为什么？

问题：这几种在vs下都是4字节，如何证明？

答：C++提供typeid运算符，基本用法为typeid(数据).name

```
#include <iostream>
using namespace std;

int main()
{
    cout << typeid(123).name() << endl;
    cout << typeid(123L).name() << endl;
    cout << typeid(123U).name() << endl;
    cout << typeid(123UL).name() << endl;
    cout << typeid(123LU).name() << endl;

    return 0;
}
```

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.1. 整型常量

```
#include <iostream>
using namespace std;

int main()
{
    cout << typeid(bool).name() << endl;
    cout << typeid(char).name() << endl;
    cout << typeid(unsigned char).name() << endl;
    cout << typeid(signed short).name() << endl;
    cout << typeid(unsigned short).name() << endl;
    cout << typeid(int).name() << endl;
    cout << typeid(unsigned).name() << endl;
    cout << typeid(unsigned long).name() << endl;
    cout << typeid(long long).name() << endl;
    cout << typeid(unsigned long long).name() << endl;
    cout << typeid(float).name() << endl;
    cout << typeid(double).name() << endl;
    cout << typeid(long double).name() << endl;
    return 0;
}
```

VS打印完整信息，Dev打印单字母，后续不再解释Dev，有兴趣自行研究即可

VS

Microsoft Visual Studio 调试控制台

```
bool
char
unsigned char
short
unsigned short
int
unsigned int
unsigned long
__int64
unsigned __int64
float
double
long double
```

Dev

D:\We

```
b
c
h
s
t
i
j
m
x
y
f
d
e
```

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.2. 浮点型常量

★ 只有十进制这一种进制，但可以表示为两种形式：

- 十进制数（带小数点的数字）
  - 0.123 123.456 123.0
- 指数形式（科学记数法）
  - ● e前面为尾数部分，必须有数，e后为指数部分,必须为整数

1.23e4	✓
1.23e-4	✓
-1.23e-4	✓
e4	×
1.23e4.5	×

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.2. 浮点型常量

★ 只有十进制这一种进制，但可以表示为两种形式：

- 十进制数（带小数点的数字）
- 指数形式（科学记数法）

★ 浮点数在内存中的存储分为三部分，分别是符号位、指数部分和尾数部分

float单精度：

31	30 … 23	22 … 0
0/1	8bit指数	23bit尾数

double双精度：

63	62 … 52	51 … 0
0/1	11bit指数	52bit尾数

- long double 的具体位数分配暂忽略，原理相同
- 浮点数的存储遵从 IEEE 754 规范

§2. 基础知识

2.6. 常量

2.6.2. 数值常量

2.6.2.2. 浮点型常量

★ 只有十进制这一种进制，但可以表示为两种形式：

- 十进制数（带小数点的数字）
- 指数形式（科学记数法）

★ 浮点数在内存中的存储分为三部分，分别是符号位、指数部分和尾数部分

★ 浮点数有指定有效位数(float:6位/double:15位)，超出有效位数则舍去(四舍五入)，因此会产生误差

例1：常量1：123456.7890123456e5

常量2：123456.7890123457e5

内部存储都是 0.1234567890123456e11

例2：1.0/3*3

不同编译系统，可能是0.999999或1（本课程的双系统都是1）

● 后续通过作业理解

★ 浮点常量缺省为double型，如需表示为float型，可以在后面加f(F)

1.23 ：double型，占8个字节

1.23F ：float型，占4个字节

可自行写测试程序来证明

```
cout << sizeof(1.23)  << endl;
cout << typeid(1.23).name()  << endl;
cout << sizeof(1.23F) << endl;
cout << typeid(1.23F).name()  << endl;
```

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.1. ASCII码

★ P.845 附录C ASCII码表

★ ASCII码占用一个字节，共可表示256个字符

0XXXXXXX ：基本ASCII码 128个(0-127)

1xxxxxxx ：扩展ASCII码 128个(128-255)

★ 基本ASCII码分图形字符和控制字符

0-32，127：控制字符，34个

33-126 ：图形字符，94个(键盘上都能找到)

★ 几个基本的ASCII码值

0 - 48/0x30 空格 - 32/0x20

A - 65/0x41 a - 97/0x61

★ 汉字的表示：

GB2312-80 ：用两个字节表示一个汉字，共6733个，两字节的高位为1(与扩展ASCII码冲突)

GBK ：1995年公布，2字节表示，收录汉字20000+

GB18030-2005：2-4个字节表示，收录汉字70000+

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.2. 普通字符常量与转义字符常量

★ 直接表示　　’空格或大部分可见的图形字符’

★ 转义符表示　’\字符、八、十六进制数’

● ’\字符’：P.43 表3.2

在表格后加：’\ddd’ ：000-377 （0-255）：8进制值对应的ASCII码

’\xhh’ ：00-ff/FF（0-255）：16进制值对应的ASCII码

注意：\x的x必须小写，否则warning(’\X41’错)，后面字符大小写不限(’\x1A’/’x1a’均可)

相似概念：整型常量的16进制，大小写均可0x41 / 0X41

```
#include <iostream>
using namespace std;
int main()
{
    cout << ’\377’ << endl;
    cout << ’\477’ << endl;
    cout << ’\x41’ << endl;
    cout << ’\X41’ << endl;
    return 0;
}
```

VS2022/DevC++：分别提示什么信息？

体会编译器的差异

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.2. 普通字符常量与转义字符常量

★ 一个字符常量可有几种表示形式

A（ASCII=65）	'A' '\101' '\x41'（'\X41'错!!!）	ESC(ASCII=27)	'\33' '\033' '\x1b' '\x1B'
换行(ASCII=10)	'\n' '\12' '\012' '\xA' '\x0A' '\xa' '\x0a'	双引号(ASCII=34)	'\"' '\42' '\042' '\x22'

★ '0'与'\0'的区别（非常重要!!!）

'0' - ASCII 48 '\60' '\060' '\x30'

'\0' - ASCII 0 '\00' '\000' '\x0' '\x00'

★ 控制字符中，除空格外，都不能直接表示，\ ' " 等特殊图形字符也不能直接表示

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.3. 字符在内存中的存储

★ 一个字符常量只能表示一个字符

★ 一个字符在内存中占用一个字节

★ 字节的值为该字符的ASCII码

```
#include <iostream>
using namespace std;
int main()
{
    cout << 'A' << endl;
    cout << sizeof('A') << endl;
    cout << typeid('A').name() << endl;
    //打印A的ASCII码值的方法(强制类型转换，后面会专题讲到)
    cout << int('A') << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
A
1
char
65
```

=> 推论：汉字不能表示为字符常量形式

- 为什么？
- 样例的结论不再讨论，初学者不建议深究

```
#include <iostream>
using namespace std;
int main()
{
    cout << '汉' << endl;
    cout << sizeof('汉') << endl;
    cout << typeid('汉').name() << endl;
    return 0;
}
```

输出

显示输出来源(S): 生成

```
已启动生成…
1>------ 已启动生成: 项目: demo-cpp, 配置: Debug Win32 ------
1>demo.cpp
1>demo-cpp.vcxproj -> D:\WorkSpace\VS2022-demo\Debug\demo-cpp.exe
========== 生成: 成功 1 个，失败 0 个，最新 0 个，跳过 0 个 ==========
```

Microsoft Visual Studio 调试控制台

```
47802
4
int
```

VS ： 0 error， 0 warning

Dev： 0 error， 3 warning

编译器 (3) | 资源 | 编译日志 | 调试 | 搜索结果 | 关闭

行	列	单元	信息
5	13	D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp	[Warning] multi-character character constant [-Wmultichar]
6	20	D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp	[Warning] multi-character character constant [-Wmultichar]
7	20	D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp	[Warning] multi-character character constant [-Wmultichar]

D:\WorkSpace\

```
47802
4
i
```

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.4. 字符串常量

含义：连续多个字符组成的字符序列

表示："字符串"

★ 可以是图形字符，也可以转义符

字符串的长度：字符序列中字符的个数

"abc123*#" = ?

"\x61\x62\x63\061\62\063\x2a\043" = ?

"\r\n\t\\A\\t\x1b\"\1234\xft\x2f\33" = ?

"\r\n\t\\A\\t\x1b\"\9234\xft\x2f\33" = ?

```
#include <iostream>
using namespace std;

int main()
{
    cout << strlen("abc123*#") << endl;
    cout << strlen("\x61\x62\x63\061\62\063\x2a\043") << endl;
    cout << strlen("\r\n\t\\A\\t\x1b\"\1234\xft\x2f\33") << endl;

    return 0;
}
```

strlen是系统函数，打印字符串的长度

思考：（课上未讲，也不再深入讨论，有兴趣可自行查找资料）

C/C++在编译8/16进制转义符时有区别：

1、转义符后的合法8/16进制数若多于3/2个

"\1234"：8进制，编译不报错，长度为2

"\x2fa"：16进制，编译报error错

2、转义符后跟非法字符

"\9234"：8进制，编译报warning错，长度为4

"*123"：同上

"\xg123"：16进制，编译报error错

"\x*123"：同上

§2. 基础知识

2.6. 常量

2.6.3. 字符常量与字符串常量

2.6.3.4. 字符串常量

在内存中的存放：每个字符的ASCII码+字符串结束标志 '\0' (ASCII 0、尾0)

★ ""与" "的区别

"" - 空字符串，长度为0

" " - 含一个空格的字符串，长度为1

★ 'A'与"A"的区别

'A' - 字符常量，内存中占一个字节

"A" - 字符串常量，内存中占两个字节

★ 暂不讨论字符串中含尾0的情况(后续模块再讨论)

"Hello\0ABC"

★ 字符串常量方式可表示汉字

\0	
32	\0
65	
65	\0

```
#include <iostream>
using namespace std;
int main()
{
    cout << "中" << endl;
    cout << sizeof("中") << endl;

    cout << "同济" << endl;
    cout << sizeof("同济") << endl;

    return 0;
}
```

Microsoft Visual

```
中
3
同济
5
```

```
#include <iostream>
using namespace std;
int main()
{
    cout << sizeof("")  << endl;
    cout << strlen("")  << endl;
    cout << sizeof(" ") << endl;
    cout << strlen(" ") << endl;

    cout << sizeof('A') << endl;
    cout << strlen('A') << endl;
    cout << sizeof("A") << endl;
    cout << strlen("A") << endl;

    return 0;
}
```

问题

1、上述8个中哪句编译错？

2、其余正确的语句，输出是什么？

3、sizeof和strlen差异？

§2. 基础知识

2.6. 常量

2.6.4. 符号常量

用一个标识符代表的常量称为符号常量

```
#define    pi   3.14159
```

★ 优点：含义清晰，修改方便

```
#include <iostream>
using namespace std;

int main()
{   ...;
    3.14159 * ...;
    ...;
    3.14159 * ...;
    ...;
    3.14159 * ...;
    ...;
    return 0;
}
```

```
#include <iostream>
using namespace std;
#define pi 3.14159
int main()
{   ...;
    pi * ...;
    ...;
    pi * ...;
    ...;
    pi * ...;
    ...;
    return 0;
}
```

假设共10000处使用π值

1、要求降低π的精度为3.14

2、要求提高π的精度为3.1415926

那种方法方便？易于修改？

在VS2022下编译运行下面的代码，观察结果

```
#define 喵喵喵 main
#define 喵喵 int
#define 喵 (
#define 喵喵呜 )
#define 喵喵喵喵 {
#define 喵呜喵 }
#define 呜喵喵 <<
#define 呜 cout
#define 呜呜 endl
#define 呜呜呜 "喵喵喵!"
#define 呜呜呜呜 return
#define 呜喵 0
#define 喵呜 ;
#include <iostream>
using namespace std;
喵喵 喵喵喵 喵 喵喵呜
喵喵喵喵
呜 呜喵喵 呜呜呜 呜喵喵 呜呜
喵呜
呜呜呜呜 呜喵 喵呜
喵呜喵
```

§2. 基础知识

2.7. 变量

2.7.1. 基本概念

变量：在程序运行中，值能够改变的量（有名字、值）

问：为什么要用变量？

=> 最开始的例2，从键盘输入两个整数到a、b中（cin >> a >> b;），因为每次输入不同，a、b必须是变量

2.7.2. 标识符

标识符：用来标识变量名、符号常量名、函数名、数组名、结构体名、类名等的有效字符序列，称为标识符

C++的标识符命名规则：由字母或下划线开头，由字母、数字、下划线组成

合理： A a a1 _a _a1 a1_b abc （前例：#define pi 3.14159 中，pi 也是标识符）

不合理： 1a a-b

★ 标识符区分大小写(也称为大小写敏感)

★ 长度<=32(或其它值，每种编译器还有参数可设，不再讨论)

★ 取名时通常按变量的含义

★ 必须先定义、后使用(某些语言不定义可直接使用，即如果第一次使用不存在，会自动创建) ⇨ 各有利弊，遵循各语言的规则即可：
例：定义时 cno1(字母1)
使用时 cno1(数字1)

★ 同级不能同名(不同级的同名问题后续讨论)

★ 标识符不能与关键字(int/float等)同名

例：int float; ×

§2. 基础知识

2.7. 变量

2.7.3. 定义变量

数据类型　标识符名，标识符名，…；

多个变量间用逗号分隔，最后有分号

```
int  a;
unsigned int  b,c;
long e,f;
short i,j;
float f1,f2;
double d1,d2;
char c1;
```

定义一个变量：
变量名为_____
存放一个_____类型的数据
在内存中占用_____个字节

§2. 基础知识

2.7. 变量

2.7.3. 定义变量

★ VS2022允许用中文做变量名，通用性差，不建议(Dev C++编译报错)

```
#include <iostream>
using namespace std;

int main()
{
    int 分数 = 78;
    cout << 分数 << endl;
    return 0;
}
```

★ C++11 标准支持auto自动定义类型，由初值决定类型，易错，
对使用者的基本概念清晰程度要求很高，初学者不建议(本课程禁用)

```
#include <iostream>
using namespace std;
int main()
{
    auto x1 = 12;
    auto x2 = 12U;
    cout << x1 - 13 << endl;
    cout << x2 - 13 << endl;

    auto y = 1.2F;
    auto z = 1.2;
    cout << sizeof(y) << endl;
    cout << sizeof(z) << endl;
    return 0;
}
```

本课程双编译器：
直接支持

某些编译器：
缺省设置编译出错，加-std=c++11后正确
● 具体方法不做要求，了解即可

§2. 基础知识

2.7. 变量

2.7.4. 对变量赋初值

变量可以在定义的同时赋初值

```
int a=10;        →   int a;
                     a=10;
char b='B';      →   char b;
                     b='B';
```

```
int a=10, b=15*2, c=30;   →   int a,b,c;
                              a=10;
                              b=15*2;
                              c=30;
```

```
int a=10, b=10, c=10;     →   int a,b,c;
                              a=10;
                              b=10;
                              c=10;
```

★ 对多个变量赋同一初值，要分开进行

int a=b=c=10; （错误）

int a=10, b=a, c=b+1; （可用已定义变量赋初值）

```
#include <iostream>
using namespace std;

int main()
{
    int a = b = c = 10;
    return 0;
}
```

```
error C2065: "b": 未声明的标识符
error C2065: "c": 未声明的标识符
```

```
#include <iostream>
using namespace std;

int main()
{
    int c, b, a = b = c = 10;
    return 0;
}
```

编译器思维培养：

1、左侧报什么编译错误？

2、右侧为什么是正确的？

3、编译器的检查流程是怎样的？

§2. 基础知识

2.7. 变量

2.7.4. 对变量赋初值

★ 若变量定义后未赋值即访问，不同编译器表现不同

VS　　：error

Dev C++：不报错，输出不可预知值 → 不可预知值：不同编译系统表现不一样，有些每次都一样，有些每次不同

```
#include <iostream>
using namespace std;

int main()
{
    int k;
    cout << k << endl;
    k=10;
    return 0;
}
```

§2. 基础知识

2.7. 变量

2.7.4. 对变量赋初值

例：在VS和DevC++中分别运行这几个程序，观察error及warning信息

```
#include <iostream>
using namespace std;
int main()
{
    float a, b=5.78*3.5, c=2*sin(2.0);
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    float b=5.78F * 3.5F, c=2*sin(2.0);
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

再次强调：

不同编译器可能在细节处理上有差异，前几个都是简单的例子，后面限于时间关系，均以VS为准，完成多编译器作业时，要有能力去解决差异

```
#include <iostream>
using namespace std;
int main()
{
    float a=0, b=5.78*3.5, c=2*sin(2.0);
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    double b=5.78*3.5, c=2*sin(2.0);
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

```
#include <iostream>
#include <cmath> //数学函数对应头文件，VS可省略
using namespace std;
int main()
{   float a, b=5.78*3.5, c=2*sin(2.0);
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

```
#include <iostream>
#include <cmath> //数学函数对应头文件，VS可省略
using namespace std;
int main()
{   double a=0, b=5.78*3.5 , c=2*sin(2.0);
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

只有提交这个版本，得分才能超过50% !!!

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.1. 整型变量的使用

★ 整型常量缺省是int型，long型后通常加l(L)表示，unsigned通常加u(U)，short型无特殊后缀
例：123 123L 123U 123UL 123LU

★ 数据的溢出在C++中不认为是错误

★ 同长度的signed与unsigned相互赋值时，可能会有不正确的结果
(机内二进制相同，但十进制表现不同)

★ 不同长度的整型数据相互赋值时，遵循如下规则：
短=>长：低位赋值，高位填充符号位(短为signed)
填充0 (短为unsigned)
长=>短：低位赋值，高位丢弃
(赋值按机内二进制进行，可能导致不正确)

★ 数据的溢出在C++中不认为是错误

```
short a=32767, b=a+1;
a=    0111111111111111 = 32767
  +)  0000000000000001
  ----------------------------
b=    1000000000000000 = -32768
```

含计算过程

注：学过2.9.5 转换优先级后，再来思考本例忽略了什么步骤!!!

```
short a=-32768, b=a-1;
a= 1000000000000000 = -32768
b= 0111111111111111 = 32767
```

计算过程略

```
unsigned short a=65535, b=a+1;
a=    1111111111111111 = 65535
b= 1 0000000000000000 = 0
```

（高位溢出，舍去）

计算过程略

```
unsigned short a=0, b=a-1;
a=    0000000000000000 = 0
b=    1111111111111111 = 65535
```

（借位不够，虚借一位）

计算过程略

```
#include <iostream>
using namespace std;

int main()
{
    short a1=32767, b1=a1+1;
    cout  << a1 << ' ' << b1 << endl;

    short a2=-32768, b2=a2-1;
    cout  << a2 << ' ' << b2 << endl;

    unsigned short a3=65535, b3=a3+1;
    cout  << a3 << ' ' << b3 << endl;

    unsigned short a4=0, b4=a4-1;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

从十进制角度理解，
相当于以几为模？

?

★ 同长度的signed与unsigned相互赋值时，可能会有不正确的结果

（机内二进制相同，但十进制表现不同）

```
short a=100; unsigned short b=a;
a= 0000000001100100 = 100
b= 0000000001100100 = 100
```

```
short a=-10; unsigned short b=a;
a= 1111111111110110 = -10
b= 1111111111110110 = 65526
```

```
unsigned short a=100; short b=a;
a= 0000000001100100 = 100
b= 0000000001100100 = 100
```

```
unsigned short a=40000; short b=a;
a= 1001110001000000 = 40000
b= 1001110001000000 = -25536
```

```
#include <iostream>
using namespace std;

int main()
{
    short a1=100;
    unsigned short b1=a1;
    cout  << a1 << ' ' << b1 << endl;

    short a2=-10;
    unsigned short b2=a2;
    cout  << a2 << ' ' << b2 << endl;

    unsigned short a3=100;
    short b3=a3;
    cout  << a3 << ' ' << b3 << endl;

    unsigned short a4=40000;
    short b4=a4;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

?

```
//P.23 趣味思考题的解法
#include <iostream>
#include <climits>
using namespace std;
int main()
{
    //学完P.26的解法
    cout << 0x7FFFFFFFFFFFFFFF << endl;

    //学完本页知识点后的解法
    unsigned long long x = -1LL;
    cout << x/2 << endl;
    return 0;
}
```

思考题的两种解法
要想明白为什么!!!

★ 短=>长：低位赋值，高位填充符号位(短为signed)
填充0　　(短为unsigned)

short a=100; long b=a;

a= 填充符号位 0 0000000001100100 =100

b= 0000000000000000 0000000001100100 =100

unsigned short a=100; long b=a;

a= 填充 0 0000000001100100 =100

b= 0000000000000000 0000000001100100 =100

short a=32767; long b=a;

a= 填充符号位 0 0111111111111111 =32767

b= 0000000000000000 0111111111111111 =32767

unsigned short a=32767; long b=a;

a= 填充 0 0111111111111111 =32767

b= 0000000000000000 0111111111111111 =32767

```
#include <iostream>
using namespace std;

int main()
{
    short a1=100;
    long b1=a1;
    cout  << a1 << ' ' << b1 << endl;

    unsigned short a2=100;
    long b2=a2;
    cout  << a2 << ' ' << b2 << endl;

    short a3=32767;
    long b3=a3;
    cout  << a3 << ' ' << b3 << endl;

    unsigned short a4=32767;
    long b4=a4;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

?

★ 短=>长：低位赋值，高位填充符号位(短为signed)
填充0　　(短为unsigned)

short a=-10; long b=a;

a= 填充符号位 1 1111111111110110 =-10

b= 1111111111111111111111111110110 =-10

short a=-10; unsigned long b=a;

a= 填充符号位 1 1111111111110110 =-10

b= 1111111111111111111111111110110 =4294967286

unsigned short a=40000; long b=a;

a= 填充0 1001110001000000 =40000

b= 00000000000000001001110001000000 =40000

unsigned short a=40000; unsigned long b=a;

a= 填充0 1001110001000000 =40000

b= 00000000000000001001110001000000 =40000

```
#include <iostream>
using namespace std;

int main()
{
    short a1=-10;
    long b1=a1;
    cout  << a1 << ' ' << b1 << endl;

    short a2=-10;
    unsigned long b2=a2;
    cout  << a2 << ' ' << b2 << endl;

    unsigned short a3=40000;
    long b3=a3;
    cout  << a3 << ' ' << b3 << endl;

    unsigned short a4=40000;
    unsigned long b4=a4;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

?

★ 长=>短：低位赋值，高位丢弃

```
long a=100;  short b=a;
a= 0000000000000000000000001100100 =100
b=                 0000000001100100 =100
```

```
long a=70000;short b=a;
a= 0000000000000001000100010111000 0 =70000
b=                 0001000101110000 =4464
```

```
long a=100000;short b=a;
a= 0000000000000001100001101010000 0 =100000
b=                 1000011010100000 =-31072
```

```
long a=100000;unsigned short b=a;
a= 0000000000000001100001101010000 0 =100000
b=                 1000011010100000 =34464
```

绝对值之和=?
为什么？

```
#include <iostream>
using namespace std;

int main()
{
    long a1=100;
    short b1=a1;
    cout  << a1 << ' ' << b1 << endl;

    long a2=70000;
    short b2=a2;
    cout  << a2 << ' ' << b2 << endl;

    long a3=100000;
    short b3=a3;
    cout  << a3 << ' ' << b3 << endl;

    long a4=100000;
    unsigned short b4=a4;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

?

★ 长=>短：低位赋值，高位丢弃

```
long a=-10;  short b=a;
a= 11111111111111111111111111110110 =-10
b=                 1111111111110110 =-10
```

```
long a=-10;  unsigned short b=a;
a= 11111111111111111111111111110110 =-10
b=                 1111111111110110 =65526
```

```
unsigned long a=4294967286;short b=a;
a= 11111111111111111111111111110110 =4294967286
b=                 1111111111110110 =-10
```

```
unsigned long a=4294967286;unsigned short b=a;
a= 11111111111111111111111111110110 =4294967286
b=                 1111111111110110 =65526
```

```
#include <iostream>
using namespace std;

int main()
{
    long a1=-10;
    short b1=a1;
    cout  << a1 << ' ' << b1 << endl;

    long a2=-10;
    unsigned short b2=a2;
    cout  << a2 << ' ' << b2 << endl;

    unsigned long a3=4294967286;
    short b3=a3;
    cout  << a3 << ' ' << b3 << endl;

    unsigned long a4=4294967286;
    unsigned short b4=a4;
    cout  << a4 << ' ' << b4 << endl;

    return 0;
}
```

?

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.2. 浮点型变量的使用

★ 浮点数在内存中的存储分为三部分，分别是符号位、指数部分和尾数部分

★ 浮点数有指定的有效位数，有效位数以外的数字被舍去，会产生误差

★ 浮点常量缺省为double型，如需表示为float型，可以在后面加f(F)

1.23　1.23F

★ float赋值给double一定正确，double赋值给float不一定正确，且不同于整型的高位丢弃，VS2022会出现 inf (infinity)的形式(有warning)，具体原因不作要求

```
#include <iostream>
using namespace std;

int main()
{
    double d=1.23456e38;
    float  f1=d;
    float  f2=d*10;
    cout << d << endl;
    cout << f1 << endl;
    cout << f2 << endl;
    return 0;
}
```

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.3. 字符型变量的使用

★ 直接表示　　'空格或大部分可见的图形字符'

★ 转义符表示　'\字符、八、十六进制数'

★ 一个字符常量可有几种表示形式

★ '0'与'\0'的区别

★ 控制字符中，除空格外，都不能直接表示，\ ' " 等特殊图形字符也不能直接表示

★ 存储形式：一个字符常量只能表示一个字符，
一个字符在内存中占用一个字节，
字节的值为该字符的ASCII码

★ 注意变量与常量的区别

```
char a,b;
a='a';        //左边是变量，右边是常量
b='\101';
```

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.3. 字符型变量的使用

★ 注意变量与常量的区别

★ 与整数的互通性：可当作1字节的整数参与运算(signed/unsigned)

```
#include <iostream>
using namespace std;
int main()
{
    char a,b,c,d,e;
    a=65;
    b=0b1000001;
    c=0101;
    d=0x41;
    e=0X41;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;
    cout << e << endl;
    return 0;
}
```

右侧为各种数制的int型常量

Microsoft

```
A
A
A
A
A
```

```
#include <iostream>
using namespace std;
int main()
{
    char a,b,c;
    a='A';
    b='\101';
    c='\x41'; //不可以'\X41'
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

右侧为各种形式的char型常量

Microsoft

```
A
A
A
```

结果虽相同，但过程不同

a=65 ： 4字节赋值给1字节，丢弃24bit 0

a='A'： 1字节赋值给1字节

输出形式：字符(cout根据输出变量类型决定)

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.3. 字符型变量的使用

★ 注意变量与常量的区别

★ 与整数的互通性：可当作1字节的整数参与运算(signed/unsigned)

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,c,d,e;
    a=65;
    b=0b1000001;
    c=0101;
    d=0x41;
    e=0X41;
    cout  << a << endl;
    cout  << b << endl;
    cout  << c << endl;
    cout  << d << endl;
    cout  << e << endl;
    return 0;
}
```

```
Microsoft
65
65
65
65
65
```

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,c;
    a='A';
    b='\101';
    c='\x41'; //不可以 '\X41'

    cout  << a << endl;
    cout  << b << endl;
    cout  << c << endl;
    return 0;
}
```

```
Microsoft
65
65
65
```

结果虽相同，但过程不同

a=65 ： 4字节赋值给4字节

a='A'： 1字节赋值给4字节，补高24bit

输出形式：数字(cout根据输出变量类型决定)

§2. 基础知识

2.7. 变量

2.7.5. 各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.3. 字符型变量的使用

★ 注意变量与常量的区别

★ 与整数的互通性：可当作1字节的整数参与运算(signed/unsigned)

```
#include <iostream>
using namespace std;

int main()
{
    char c1,c2;
    c1 = 'a';
    c2 = 'b';
    c1 = c1-32;
    c2 = c2-32;
    cout << c1 <<' '<< c2 <<endl;

    return 0;
}
```

换为int，输出什么？

1、___字节常量赋值给___变量
2、___字节变量 - ___常量(不同字节相减规则待定)，
再赋值给___字节变量
3、输出为_____形式(字符/数字)

Microsoft Visual Studio 调试控制台

```
A B
```

§2.基础知识

2.7.变量

2.7.5.各种类型变量的使用(归纳+补充+重点+难点)

2.7.5.4.字符串型变量的使用

★ C++中无字符串变量，可用一维字符数组来表示字符串变量（后续再讨论）

§2. 基础知识

2.7. 变量

2.7.6. 常变量

含义：在程序执行过程中值不能改变的变量

形式：const 数据类型 变量名=初值;

数据类型 const 变量名=初值;

例：const int a=10;

double pi=3.14159;

★ 常变量必须在定义时赋初值，且在执行过程中不能再次赋值，否则编译错误

```
#include <iostream>
using namespace std;

int main()
{
    const int a;

    return 0;
}
```

error C2734: “a”: 如果不是外部的，则必须初始化常量对象

```
#include <iostream>
using namespace std;

int main()
{
    const int a=10;
    a=15;

    return 0;
}
```

error C3892: “a”: 不能给常量赋值

§2. 基础知识

2.7. 变量

2.7.6. 常变量

★ 常变量与符号常量使用方法相似，但本质有区别（具体不做要求，推荐使用常变量）

- 常变量：有类型、有空间、有初始值（除了值不可变，其余同变量）
- 符号常量：一个标识符替代一串字符

```
#include <iostream>
using namespace std;
#define pi 3.14159    //①无分号
int main()
{
    double r=3, s, v;
    s = pi*r*r;        //②
    v = pi*r*r*r*4/3;  //③
    cout << s << endl;
    cout << v << endl;
    return 0;
}
```

1、将①改为#define pi 3.14159;（有分号）报什么错？
2、在①有分号的基础上，②改为s=r*r*pi;
③改为v=r*r*r*4/3*pi;(均有分号)，
能否正确？如何解释？
3、在①有分号的基础上，②改为s=r*r*pi
③改为v=r*r*r*4/3*pi(均无分号)，
能否正确？如何解释？

```
#include <iostream>
using namespace std;
int main()
{
    const double pi=3.14159; //①有分号
    double r=3, s, v;
    s = pi*r*r;        //②
    v = pi*r*r*r*4/3;  //③
    cout << s << endl;
    cout << v << endl;
    return 0;
}
```

1、将①改为const double pi=3.14159（无分号），报什么错？
2、在①无分号的基础上，②改为s=r*r*pi
③改为v=r*r*r*4/3*pi（均无分号），
报什么错？

回忆课上所讲，自行测试

§2. 基础知识

2.8. C++的运算符

2.8.1. C++的运算符的种类

总：下发的附录（D-运算符优先级）

★ 运算符根据参与运算的操作数的个数，分为一元/二元/三元运算符，也称为单目/双目/三目运算符

★ 唯一的三目运算符是第15组的":?"

★ 勘误 => 附录D P.850 优先级第5组

*

/

% （书上印为^）

2.8.2. 运算符的优先级和结合性

优先级：不同运算符进行混合运算时，按优先级的高低依次执行

结合性：同级运算符进行混合运算时，按结合性的方向依次进行处理

左结合(L-R)：从左至右进行同级运算符的混合运算

右结合(R_L)：从右至左进行同级运算符的混合运算

附录D（1最高 18最低 熟记!!!）

=> 随着学习过程慢慢加入新的运算符

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.1. 基本的算术运算符

+　　-　　*　　/　　%

★ 字符型可以参与算术运算，当作1字节的整型数，值为其ASCII码

'A' * 10 => 65 * 10 => 650

```
#include <iostream>
using namespace std;
int main()
{
    cout << 'A' * 10 << endl;
    return 0;
}
```

Microsoft Visual

```
650
```

★ %的使用(求模，整数相除求余数)，%两侧为整型

(char, short, int, long, long long及对应的unsigned)

10 % 3　　1 10 % 7　　3 7 % 3　　1 3 % 7　　3	3) 10 = 3, 9, 1	10 % -3　　1 10 % -7　　3 7 % -3　　1 3 % -7　　3	-3) 10 = -3, 9, 1
-10 % 3　　-1 -10 % 7　　-3 -7 % 3　　-1 -3 % 7　　-3	3) -10 = -3, -9, -1	-10 % -3　　-1 -10 % -7　　-3 -7 % -3　　-1 -3 % -7　　-3	-3) -10 = 3, -9, -1

★ 整数相除（ / ）的使用

结果为整数(舍去小数, 非四舍五入)

```
//比较容易犯的错误
#include <iostream>
using namespace std;
int main()
{
    double pi=3.14159, v1, v2;
    int   r=3, h=5;
    v1=1/3*pi*r*r*h;
    v2=4/3*pi*r*r*r;
    cout << v1 << endl;
    cout << v2 << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0
84.8229
```

实际	期望
0	47.1238
84.8229	113.097

4/3更有欺骗性!!!

§2.基础知识

2.9.算术运算符与算术表达式

2.9.2.算术运算符的优先级与结合性

优先级：* / % 5级

+ - 6级

都是双目运算符

结合性：左结合(L-R)

2.9.3.算术表达式

含义：用算术运算符及括号将操作数（运算对象：常量、变量、函数）连接起来，组成符合C++语法规则的算式

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.4. C++的表达式求值(补充，重要!!!)

表达式：由若干操作数和操作符构成的符合C++语法的算式

表达式的求值：整个表达式从左到右依次分析，分析原则如下

★ 若只有一个运算符，则求值

a+b

★ 若有两个运算符，若

① 左边运算符的优先级高于右边运算符　或

② 左边运算符的优先级等于右边运算符，且该级别运算符是左结合，则对左边运算符求值，其值再参与后续的运算

a*b+c

a+b+c

★ 若有两个运算符，若

① 左边运算符的优先级低于右边运算符　或

② 左边运算符的优先级等于右边运算符，且该级别运算符是右结合，则先忽略左边运算符，继续向后分析，直到右边运算符被求值后再次分析左边运算符

a+=a-=a*a

a+b*c

a+b*c-d

a+b*c*d

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.4. C++的表达式求值(补充，重要!!!)

★ 用栈(LIFO)的形式理解表达式求值过程

LIFO：Last In First Out

运算数栈：存放运算数

运算符栈：存放运算符

规则：(1) 运算数/运算符分别进各自的栈

(2) 若欲进栈的运算符级别高于或等于(右结合)栈顶运算符，则进栈

(3) 若欲进栈的运算符级别低于或等于(左结合)栈顶运算符，则先将栈顶运算符计算完成，计算数为运算数栈的两个元素，运算符及运算数出栈，运算结果进栈

(4) 重复上述步骤至运算符栈为空，运算数栈只有一个元素，否则认为语法错

例:10+’a’+i*f-d/e

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

初始：两栈均为空

例:10+'a'+i*f-d/e

说明：10 - 整型常量
'a' - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

10进栈

例:10+'a'+i*f-d/e

说明：10 - 整型常量
'a' - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

+进栈

例:10+'a'+i*f-d/e

说明：10 - 整型常量
'a' - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

'a'进栈

例:10+’a’+i*f-d/e

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

要进栈的(+)等于栈顶(+)，且左结合，求值

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

要进栈的(+)等于栈顶(+)，且左结合，求值

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

+进栈

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

i进栈

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

*进栈 （要进栈的*高于栈顶的+）

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

f进栈

例:10+’a’+i*f-d/e

步骤① 10+’a’

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

f
i
10+’a’

*
+

要进栈的(-)低于栈顶的(*)，先计算

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

要进栈的(-)低于栈顶的(*)，先计算

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

要进栈的(-)等于栈顶的(+)，左结合，先计算

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

要进栈的(-)等于栈顶的(+)，左结合，先计算

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

-进栈

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

d进栈

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

/进栈 （要进栈的/高于栈顶的-）

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

e进栈

例:10+'a'+i*f-d/e

步骤① 10+'a'

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

步骤④ d/e

说明：10 - 整型常量
'a' - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

计算 d/e

例:10+'a'+i*f-d/e

步骤① 10+'a'

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

步骤④ d/e

步骤⑤ ③-④ （步骤③的和 - 步骤④的商）

说明：10 - 整型常量
'a' - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

计算 ③-④

例:10+’a’+i*f-d/e

步骤① 10+’a’

步骤② i*f

步骤③ ①+② （步骤①的和 + 步骤②的积）

步骤④ d/e

步骤⑤ ③-④ （步骤③的和 - 步骤④的商）

说明：10 - 整型常量
’a’ - 字符型常量
i - 某个int型变量
f - 某个float型变量
d - 某个double型变量
e - 某个long型变量

说明：
本例只是一个简单的说明，表述也不够准确，且未考虑到括号、单目、三目运算符等，真实的表达式求值比这个复杂得多，待后续课程再进行深入学习

- 做为初学者理解，够用了

表达式分析并求值完成(运算符栈为空，运算数栈只有一个数)

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.5. 字符型、整型、实型的混合运算

★ 运算的方法

10+'a'+1.5-8765.1234*'b'

如果某个运算符涉及到两个数据不是同一类型，则需要先转换成同一类型，再进行运算

★ 转换的优先级（阅读：P.53~57 3.4.4 类型转换）

高 ↑ long double
double
float
unsigned long long
long long
unsigned long
long
unsigned [int]
低 int <- char, u_char, short, u_short

```
char a;
short b;

a+b    (int + int)
```

<-整型提升(表示必定的转换)

整型提升的等价说法：
参与算术运算的最小数据类型是int

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.5. 字符型、整型、实型的混合运算

★ 转换的优先级（阅读：P.53~57 3.4.4 类型转换）

思考：怎么证明说法的正确性？（引申：如何解决学习过程中碰到的含义不清的问题？）

```
#include <iostream>
using namespace std;
int main()
{
    short a = 1;
    short b = 32767;

    cout << a+b << endl;

    return 0;
}
```

思考：b=a+1中
问1：a+1=?
　　什么类型?
问2：b为什么是-32768?

已知：short a=32767, b=a+1;
　则：b是-32768
验证：int <- char / short是必定的转换
预测：结果为-32768 则验证不正确
　　结果为 32768 则验证正确

```
#include <iostream>
using namespace std;
int main()
{
    short a = 32767;
    short b = 1;
    cout << sizeof(a + 'A') << endl;
    cout << typeid(a + b).name() << endl;

    return 0;
}
```

其它两种可用的方法

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.5. 字符型、整型、实型的混合运算

例：10+'a'+i*f-d/e（刚才用栈解释表达式求值的例子）

① 10+'a'　　int + int

② i*f　　float * float

③ ①+②　　float + float

④ d/e　　double / double

⑤ ③-④　　double - double

注意：不是将整个算式中优先级最高的最先计算，而是分步进行

★ 同级的signed与unsigned混合时，以unsigned为准

```
#include <iostream>
using namespace std;
int main()
{   int a=2;
    int b=3;
    cout << a-b << endl;
    return 0;
}                          -1
```

```
#include <iostream>
using namespace std;
int main()
{   int a=2;
    unsigned int b=3;
    cout << a-b << endl;
    return 0;
}                          4294967295
```

★ 类型转换由系统隐式进行

★ 类型转换时，不是一次全部转换成最高级，而是依次转换

'a'+10+15L+1.2

'a'+10	：char=>int	int+int
'a'+10+15L	：int=>long	long+long
'a'+10+15L+1.2	：long=>double	double+double

比较容易犯的错误：

```
float pi=3.14159, v1, v2;
int   r=3, h=5;
圆锥体积：v1=1/3*pi*r*r*h;
球体积：   v2=4/3*pi*r*r*r;
```

	实际	期望
圆锥体积	0	47.1238
球体积	84.8229	113.097

此例正因为是依次转换，才得到上述结果

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.5. 字符型、整型、实型的混合运算

★ 运算的方法

10+'a'+1.5-8765.1234*'b'

如果某个运算符涉及到两个数据不是同一类型，则需要先转换成同一类型，再进行运算

★ 转换的优先级（阅读：P.53~57 3.4.4 类型转换）

特别说明：转换优先级的规则很复杂，这里只是一个适合初学者的简易规则，
不完全正确，如果与实际编译器的表现有不一致的地方，以编译器为准
https://zh.cppreference.com/w/c/language/conversion
https://zh.cppreference.com/w/cpp/language/operator_arithmetic

高 ↑ long double
double
float
unsigned long long
long long
unsigned long
long
unsigned [int]
低 int <- char, u_char, short, u_short
<- 整型提升(表示必定的转换)

4) 否则两个操作数均为整数。此情况下，
首先，两个操作数都会经历*整数提升*（见后述）。然后

- 若两类型在提升后相同，则该类型即为共用类型
- 否则，若两操作数在提升后有相同的符号性（均为有符号或均为无符号），则拥有较低*转换等级*（见后述）者会隐式转换成拥有较高*转换等级*的操作数的类型
- 否则，两者符号性不同：若无符号类型操作数拥有大于或等于有符号类型操作数的*转换等级*，则有符号类型操作数会隐式转换成无符号类型
- 否则，两者符号性不同且有符号操作数的等级大于无符号操作数的等级。此情况中，若有符号类型可以表达无符号类型的所有值，则有无符号类型的操作数被隐式转换成有符号操作数的类型。
- 否则，两个操作数都会经历隐式转换，到有符号类型的无符号类型对应者。

```
1.f + 20000001; // int 被转换成 float ，给出 20000000.00
                // 相加后舍入到 float ，给出20000000.00
(char)'a' + 1L; // 首先，char 被提升回 int。
                // 这是有符号+有符号的情形，等级不同
                // int 被转换成 long ，结果是 signed long 的 98
2u - 10; // 无符号/有符号，等级相同
         // 10 被转换成无符号，无符号数学运算为模 UINT_MAX+1
         // 对于 32 位 int ，结果是 unsigned int 类型的 4294967288 （即 UINT_MAX-7 ）
0UL - 1LL; // 无符号/有符号，相异等级，有符号的等级较大。
           // 若 sizeof(long) == sizeof(long long) ，则有符号数不能表示所有无符号数
           // 这是最后一种情况：两个操作数都被转换成 unsigned long long
           // 结果是 unsigned long long 类型的 18446744073709551615 （ ULLONG_MAX ）
```

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.6. 自增与自减运算符

2.9.6.1. 形式

++（--）变量名　　先自增/减1，后使用(前缀，优先级第3组)

变量名++（--）　　先使用，后自增/减1(后缀，优先级第2组)

```
#include <iostream>
using namespace std;
int main()
{   int i=3, j;
    j = ++i;   //前缀
    cout << i << endl;   4
    cout << j << endl;   4
    return 0;
}
```

① ++优先级高于=，先计算++
② 因为++是前缀，先++i成为4
③ 再将i值4赋值给j

```
#include <iostream>
using namespace std;
int main()
{   int i=3, j;
    j = i++;   //后缀
    cout << i << endl;   4
    cout << j << endl;   3
    return 0;
}
```

① ++优先级高于=，先计算++
② 后缀++,将i的原值3保留到某个中间变量中
接下来，因为不同的编译器(例如VS系列和gcc系列)，可能有两种执行顺序

③ 先将原值赋给j	③ 先进行++,i值为4
④ 再进行++,i值为4	④ 再将原值赋给j

无论哪种顺序，都是j=3 i=4

```
#include <iostream>
using namespace std;
int main()
{
    int i = 3;
    i = i++;
    cout << i << endl;
    return 0;
}
```

无聊的做法，仅为了理解

顺序1	顺序2
i值为4	i值为3
VS	DevC++

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.6. 自增与自减运算符

2.9.6.1. 形式

2.9.6.2. 使用

★ ++/--的前/后缀对变量自身无影响，影响的是参与运算的表达式

```
int i=3;
i++;
++i;
```

单独成为语句时，前后缀等价

思考：在不考虑CPU硬件优化，编译器软件优化的情况下，纯语句角度考虑，哪种效率更高？为什么？

★ 前后缀的优先级和结合性均不相同

后缀：优先级(2)　前缀：优先级(3)

左结合　　右结合

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.6. 自增与自减运算符

2.9.6.1. 形式

2.9.6.2. 使用

★ ++/--的前/后缀对变量自身无影响，影响的是参与运算的表达式

★ 前后缀的优先级和结合性均不相同

★ ++/--后，变量的值改变(不能对常量、表达式)

int i=3, j=4;

const int k=3;

(i+j)++：编译器认为和是只读的 (错)

10++：常量值不能被改变 (错)

k++：常变量的值不能被改变 (错)

```
#include <iostream>
using namespace std;
int main()
{
    int i = 3, j = 4;
    const int k = 3;
    (i + j)++;
    10++;
    k++;
    return 0;
}
```

```
error C2105: "++" 需要左值
error C2105: "++" 需要左值
error C2105: "++" 需要左值
```

★ 不主张对同一个变量的多个++/--出现在同一个表达式中

例：(i++)+(i++)+(i++)

(不同编译系统处理方式不同，不深入讨论)

```
//用不同编译器测试本程序
#include <iostream>
using namespace std;
int main()
{
    int  i = 3, k;
    k = (i++) + (i++) + (i++);
    cout << i << ' ' << k << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
6 9
```

VS

D:\WorkSpace\VS2019-demo\demo-CPP\demo.exe

```
6 12
```

Dev

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.7. 强制类型转换

(类型名)(表达式) / 类型名(表达式) / static_cast<类型名>(表达式)

(int)(a+b)	int(a+b)	static_cast<int>(a+b)
(int)a	int(a)	static_cast<int>(a)
C形式	C++形式	C++形式

C方式

```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    double b=3.5, c=3.6;

    cout << sizeof(a + (int)b) << endl;
    cout << sizeof(a + (int)b + c) << endl;
    cout << sizeof(a + (int)(b+c)) << endl;
    cout << endl;

    cout << a + (int)b << endl;
    cout << a + (int)b + c << endl;
    cout << a + (int)(b + c) << endl;
    cout << endl;

    cout << typeid(a + (int)b).name() << endl;
    cout << typeid(a + (int)b + c).name() << endl;
    cout << typeid(a + (int)(b + c)).name() << endl;

    return 0;
}
```

C++方式

```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    double b=3.5, c=3.6;

    cout << sizeof(a + int(b)) << endl;
    cout << sizeof(a + int(b) + c) << endl;
    cout << sizeof(a + int(b+c)) << endl;
    cout << endl;

    cout << a + int(b) << endl;
    cout << a + int(b) + c << endl;
    cout << a + int(b + c) << endl;

    cout << endl;
    cout << typeid(a + int(b)).name() << endl;
    cout << typeid(a + int(b) + c).name() << endl;
    cout << typeid(a + int(b + c)).name() << endl;

    return 0;
}
```

```
Microsoft Visual
4
8
4

13
16.6
17

int
double
int
```

三个程序输出一致

C++方式

```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    double b=3.5, c=3.6;

    cout << sizeof(a + static_cast<int>(b)) << endl;
    cout << sizeof(a + static_cast<int>(b) + c) << endl;
    cout << sizeof(a + static_cast<int>(b+c)) << endl;
    cout << endl;

    cout << a + static_cast<int>(b) << endl;
    cout << a + static_cast<int>(b) + c << endl;
    cout << a + static_cast<int>(b + c) << endl;
    cout << endl;

    cout << typeid(a + static_cast<int>(b)).name() << endl;
    cout << typeid(a + static_cast<int>(b) + c).name() << endl;
    cout << typeid(a + static_cast<int>(b + c)).name() << endl;

    return 0;
}
```

§2. 基础知识

2.9. 算术运算符与算术表达式

2.9.7. 强制类型转换

(类型名)(表达式) / 类型名(表达式) / static_cast<类型名>(表达式)

```
(int)(a+b)     /  int(a+b)      /  static_cast<int>(a+b)
(int)a         /  int(a)        /  static_cast<int>(a)
```

C形式　　C++形式　　C++形式

```
int a;
double b,c;
  a+(int)b      / a+int(b)    / a+static_cast<int>(b)      int型
  a+(int)b+c    / a+int(b)+c  / a+static_cast<int>(b)+c    double型
  a+(int)(b+c)  / a+int(b+c)  / a+static_cast<int>(b+c)    int型
```

C形式　　C++形式　　C++形式

★ 必须在程序中显式使用

★ 强制转换后，原变量的值、类型不变(强制转换的结果放在一个中间变量中)

```
int a = -3;
unsigned int b = 2;
cout << a+b << endl;
```

?

```
int a = -3;
unsigned int b = 2;
cout << int(a+b) << endl;
```

?

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.1. 形式

变量 = 数据（常量、变量、表达式）

a=b

a=15

a=b*c-d

★ 赋值运算符左边必须是变量名

★ 若赋值运算符的左右类型不同，则以左值类型为准进行转换

```
float a;
a = 1.2;
```

（右值double转换为左值float型，VS有warning）

```
warning C4305: “=”: 从“double”到“float”截断
```

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.2. 字符型、整型、实型间相互赋值的类型转换

★ float/double => char/short/int/long/long long时，取整

● 保证合理范围，否则可能溢出(C++语法不错)

f=123.0 => char=123

f=12345.67 => short 12345

=> char 溢出

f=1234567.89 => long 1234567

=> char 溢出

=> short 溢出

```
#include <iostream>
using namespace std;
int main()
{
    float f1 = 123.0F;
    char c1 = f1;
    cout << int(c1) << endl;

    double f2 = 12345.67;
    char c2 = f2;
    short s2 = f2;
    cout << int(c2) << ' ' << s2 << endl;

    double f3 = 1234567.89;
    char c3 = f3;
    short s3 = f3;
    long l3 = f3;
    cout <<int(c3)<<' '<< s3 <<' '<<l3<<endl;

    return 0;
}
```

1、观察运行结果
2、int(c1)的目的是什么？
3、溢出的结果与长字节整数赋短字节的规律是否相同？

Microsoft Visual Studio 调试控制台

```
123
57 12345
-121 -10617 1234567
```

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.2. 字符型、整型、实型间相互赋值的类型转换

★ char/short/int/long/long long => float/double，不变

后面补零，不会溢出，但精度受影响

```
char 123        => float  123
int  12345      => double 12345
long 1234567    => float  1.23457e+06
long 123456700  => float  1.23457e+08
long 123456789  => float  1.23457e+08
```

自行构造测试程序并理解

● 实型数运算时要四舍五入

★ float/double相互赋值时

float -> double ：不会错

double -> float ：可能溢出（VS中是 inf 形式）

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.2. 字符型、整型、实型间相互赋值的类型转换

★ char/short/int/long/long long相互赋值时

少字节->多字节：低位赋值，高位补符号位/0(十进制的表示形式可能不同)

```
char  123 => long 123
short -10 => unsigned int 4294967286
```

多字节->少字节：低位赋值，高位自动截断(可能溢出)

```
long 1234  => short 1234
long 70000 => short 4464
```

★ 同类型signed与unsigned相互赋值时，值的二进制形式不变，但十进制表示形式可能不同

```
unsigned short a=65535;
short b;
b=a;  b=-1
```

(unsigned)65535 = 1111 1111 1111 1111 = (signed)-1

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.3. 复合的赋值运算符

含义：将算术运算符/位运算符和赋值组合在一起，同时完成计算并赋值

形式：变量 复合赋值运算符 表达式(常量、变量、表达式)

附录D 优先级第16组 共10种(目前要求：+ - * / %，位运算符暂略)

a+=b => a=a+b

a*=b+1 => a=a*(b+1)

★ 优先级相同（注意：+、*优先级不同，但+=、*=优先级相同）

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.4. 赋值表达式

2.10.4.1. 含义

将一个变量和一个表达式用赋值运算符连接起来

2.10.4.2. 形式

变量 = 表达式（常量、变量）

a=b

a=5

a=b+c

a=(b=5)　　如何理解？

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.4. 赋值表达式

2.10.4.3. 赋值表达式的值(含复合赋值表达式)

和变量的值相等

a=b=c=5 => a=(b=(c=5))

理解：

1、将常量5赋值给c，赋值表达式 c=5 的值也是5

2、将赋值表达式c=5的值5赋值给b，赋值表达式 b=(c=5) 的值也是5

3、将赋值表达式b=(c=5)的值5赋值给a，赋值表达式 a=(b=(c=5)) 的值也是5

★ 变量赋初值时，int a=b=c=5;不行

int a=5,b=5,c=5; 正确	int a,b,c;
int a=b=c=5; 错误	a=b=c=5; 正确

```
error C2065: “b”: 未声明的标识符
error C2065: “c”: 未声明的标识符
```

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.4. 赋值表达式

2.10.4.3. 赋值表达式的值(含复合赋值表达式)

★ 变量赋初值时，int a=b=c=5;不行

★ 赋值表达式的值可以参与其它表达式的运算

```
int a=12;
a+=a-=a*a;
```

步骤	原因	执行的表达式	a的值
① a*a	* 优先级最高	12*12	a=12
② a-=a*a	+=、-=相同，右结合	a-=12*12	a=12
③ a=a-a*a	-=展开	a=12-12*12	a=-132
④ a+=(a=a-a*a)	+=最后	a+=-132	a=-132
⑤ a=a+(a=a-a*a)	+=展开	a=(-132)+(-132)	a=-264

```
#include <iostream>
using namespace std;

int main()
{
    int a = 12;
    a += a -= a * a;
    cout << a << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
-264
```

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.4. 赋值表达式

2.10.4.3. 赋值表达式的值(含复合赋值表达式)

★ 变量赋初值时，int a=b=c=5;不行

★ 赋值表达式的值可以参与其它表达式的运算

思考：观察下面的程序，想明白为什么左侧是正确而右侧是错误的，错在哪个=上？

(a=3*5)=4*3 3*5 a=3*5 a=15 4*3 a=4*3 a=12 虽然=的左边是一个表达式，不是要求的变量，但先计算该表达式，使=执行时形式为变量	a=3*5=4*3 3*5 4*3 4*3 => 3*5 (4*3的值赋给表达式3*5,错)

`#include <iostream>`
`using namespace std;`
`int main()`
`{`
`int a;`
`(a = 3 * 5) = 4 * 3;`
`cout << a << endl;`
`return 0;`
`}`
 	
Microsoft Visual Studio 调试控制台
12
	`#include <iostream>`
	`using namespace std;`
	`int main()`
	`{`
	`int a;`
	`a = 3 * 5 = 4 * 3;`
	`cout << a << endl;`
	`return 0;`
	`}`
	
	error C2106: “=”: 左操作数必须为左值

§2. 基础知识

2.10. 赋值运算符与赋值表达式

2.10.4. 赋值表达式

2.10.4.3. 赋值表达式的值(含复合赋值表达式)

★ 变量赋初值时，int a=b=c=5;不行

★ 赋值表达式的值可以参与其它表达式的运算

★ 不讨论相同变量的多个赋值表达式同时运算的情况

例：cout << (a=10)+(a=20) << endl;

不同编译器可能20、30、40（VS是40，RedFlag5是30）

```
#include <iostream>
using namespace std;

int main()
{
    int a;
    cout << (a = 10) + (a = 20) << endl;
    return 0;
}
```

不同编译器取值不同：

VS/DevC++ : 40

RedFlag5 gcc : 30

```
[root@RF5-X64 ~]# cat test.cpp
#include <iostream>
using namespace std;
int main()
{
        int a;
        cout << (a = 10) + (a = 20) << endl;
        return 0;
}

[root@RF5-X64 ~]# c++ -o test test.cpp
[root@RF5-X64 ~]# ./test
30
[root@RF5-X64 ~]#
```

§2. 基础知识

2.11. 逗号运算符和逗号表达式

2.11.1. 形式

表达式1，表达式2，...，表达式n

★ C++中级别最低的运算符(又称为顺序求值运算符)

2.11.2. 逗号表达式的值

顺序求表达式1，2，...，n的值，整个逗号表达式的值为第n个表达式的值

a=3*5, a*4 式1(赋值表达式)：a=15 式2(算术表达式)：15*4=60 整个逗号表达式的值为60	b=(a=3*5, a*4) b = 60（赋值表达式,将逗号表达式的值赋给b）
(a=3*5, a*4), a+5 式1(逗号表达式) 式1-1(赋值表达式)=15 (a=15) 式1-2(算术表达式)=60 式1 =60 式2(算术表达式)=20 整个逗号表达式的值为20	b = ((a=3*5, a*4), a+5) b=20（赋值表达式，将逗号表达式的值赋给b）
int a=5, b=4, c=3; cout << a << b << c; cout << (a,b) << (a,c) << (a,b,c);	543433 为什么？ 再次强调：只输出最基本的信息

§2. 基础知识

2.12. 关于C++表达式的总结

★ C++中任意类型的表达式均有值

- 算术表达式
- 赋值表达式、复合赋值表达式
- 逗号表达式

★ 表达式按照优先级/结合性逐步求值(借助栈理解)，运算过程中可能还会涉及到类型转换

★ 表达式类型由最后一个运算决定

b = (a=3*5, a*4) ：赋值表达式

b = a=3*5, a*4 ：逗号表达式

§2. 基础知识

2.13. 位运算

2.13.1. 位运算的基本概念

2.13.1.1. 字节和位

字节：byte，计算机中数据表示的基本单位

位　：bit，　计算机中数据表示的最小单位

1 byte = 8 bit

2.13.1.2. 位运算

以bit为单位进行数据的运算

再次强调：
有符号数：符号位是最高位(0/1)
无符号数：符号位是0

2.13.1.3. 位运算的基本方法

★ 按位进行（只有0、1）

★ 要求运算数据长度相等，若不等，则右对齐，按符号位补齐左边

char a=0x37;	0000 0000 0011 0111
short b=0x1234;	0001 0010 0011 0100
char a=0xA7;	1111 1111 1010 0111
short b=0x8341;	1000 0011 0100 0001
unsigned char a=0xA7;	0000 0000 1010 0111
short b=0x8341;	1000 0011 0100 0001

★ 数在计算机内是用补码表示的

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.1. 与(&)

运算规则：遇0得0

例：char a=3,b=5; 求a&b

```
  0000 0011
& 0000 0101
  0000 0001              a&b=1
```

例：char a=0x87; short b=0x9c52; 求a&b

```
  1111 1111 1000 0111
& 1001 1100 0101 0010
  1001 1100 0000 0010    a&b=0x9c02 (-25598)
```

例：unsigned char a=0x87; short b=0x9c52;求a&b

```
  0000 0000 1000 0111
& 1001 1100 0101 0010
  0000 0000 0000 0010    a&b=0x2
```

例：char a=0xb6,b=0xc2; 求a&b

```
  1011 0110
& 1100 0010
  1000 0010              a&b=0x82  (-126)
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a1 = 3, b1 = 5;
    cout << "a=" << (int)a1 << " b=" << (int)b1 << " a&b=" << (a1&b1) << endl;

    char a2 = 0x87;
    short b2 = 0x9c52;
    cout << "a=0x" << hex << (int)a2 << " b=0x" << b2  << " a&b=0x" << (a2&b2)
         << " a&b=0x" << short(a2&b2) << " " << dec << " a&b=" << (a2&b2) << endl;

    unsigned char a3 = 0x87;
    short b3 = 0x9c52;
    cout << "a=0x" << hex << (int)a3 << " b=0x" << b3 << " a&b=0x" << (a3&b3) << endl;

    char a4 = 0xb6, b4 = 0xc2;
    cout << "a=0x" << hex << (int)a4 << " b=0x" << (int)b4 << " a&b=0x" << (a4&b4)
         << " " << dec << "a&b=" << (a4&b4) << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台　读懂运行结果!!!

```
a=3 b=5 a&b=1
a=0xffffff87 b=0x9c52 a&b=0xffff9c02 a&b=0x9c02  a&b=-25598
a=0x87 b=0x9c52 a&b=0x2
a=0xffffffb6 b=0xffffffc2 a&b=0xffffff82 a&b=-126
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.1. 与(&)

运算规则：遇0得0

应用：

★ 清零

例：char a=0xb6;现要求将该数清零，则：

1011 0110

& 0?00 ?00?　　要清零数为1的位，本数对应位为0

0000 0000

a&0x0　a&0x1　a&0x8　a&0x9

a&0x40　a&0x41　a&0x48　a&0x49

★ 取指定位

例：char a=0xb6;现要求只保留低4位，

而高4位清0，则：

1011 0110

& 0000 1111　　要保留的位，本数对应位为1

0000 0110

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    /* &的应用：清0 */
    char a1=0xb6;
    cout << "char a=" << hex << (int)a1 << endl
         << "     a&0x0 =" << dec << (a1&0x0)  << endl
         << "     a&0x1 =" << dec << (a1&0x1)  << endl
         << "     a&0x8 =" << dec << (a1&0x8)  << endl
         << "     a&0x9 =" << dec << (a1&0x9)  << endl
         << "     a&0x40=" << dec << (a1&0x40) << endl
         << "     a&0x41=" << dec << (a1&0x41) << endl
         << "     a&0x48=" << dec << (a1&0x48) << endl
         << "     a&0x49=" << dec << (a1&0x49) << endl;

    /* &的应用：取指定位 */
    char a2=0xb6;
    cout << "char a=0x" << hex << (int)a2
         << " a&0x0F="  << dec << (a2&0x0F)
         << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
char a=ffffffb6
     a&0x0 =0
     a&0x1 =0
     a&0x8 =0
     a&0x9 =0
     a&0x40=0
     a&0x41=0
     a&0x48=0
     a&0x49=0
char a=0xffffffb6 a&0x0F=6
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.2. 或(|)

运算规则：遇1得1

例：char a=3,b=5; 求a|b

```
  0000 0011
| 0000 0101
  0000 0111     a|b=7
```

例：char a=3; short b=5;求a|b

```
  0000 0000 0000 0011
| 0000 0000 0000 0101
  0000 0000 0000 0111     a|b=7
```

例：char a=0xb6,b=0xc2; 则a|b

```
  1011 0110
| 1100 0010
  1111 0110     a|b=0xF6
```

有符号10进制：-10

应用：★ 设定某些位为1

例：char a=0xb6;要求1,4位设为1，其它不变

```
  1011 0110
| 0000 1001
  1011 1111
```

要设置的位，本数对应位为1

(0xBF)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a1=3, b1=5;
    cout << "a=" << (int)a1 << " b=" << (int)b1 << " a|b=" << (a1|b1) << endl;

    char a2=3;
    short b2=5;
    cout << "a=" << (int)a2 << " b=" << b2 << " a|b=" << (a2|b2) << endl;

    char a3=0xb6, b3=0xc2;
    cout << "a=" << hex << (int)a3 << " b=" << (int)b3;
    cout << " a|b=0x" << hex << (a3|b3) << " " << dec << (a3|b3) << endl;

    /* |的应用，将1、4 bit位设为1，其它不变 */
    char a4=0xb6;
    cout << "a=" << hex << (int)a4 << " a|0x9=0x" << (a4|0x9) << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=3 b=5 a|b=7
a=3 b=5 a|b=7
a=ffffffb6 b=ffffffc2 a|b=0xfffffff6 -10
a=ffffffb6 a|0x9=0xffffffbf
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.3. 异或(^)

运算规则：相同为0，不同为1

例：char a=3,b=5; 求a^b

```
  0000 0011
^ 0000 0101
  0000 0110      a^b=6
```

例：char a=3; short b=5;求a^b

```
  0000 0000 0000 0011
^ 0000 0000 0000 0101
  0000 0000 0000 0110      a^b=6
```

例：char a=0xb6,b=0xc2; 则a^b

```
  1011 0110
^ 1100 0010
  0111 0100      a^b=0x74
```

有符号10进制：116

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a1=3, b1=5;
    cout << "a=" << (int)a1 << " b=" << (int)b1 << " a^b=" << (a1^b1) << endl;

    char a2=3;
    short b2=5;
    cout << "a=" << (int)a2 << " b=" << b2 << " a^b=" << (a2^b2) << endl;

    char a3=0xb6, b3=0xc2;
    cout << "a=" << hex << (int)a3 << " b=" << (int)b3;
    cout << " a^b=0x" << hex << (a3^b3) << " " << dec << (a3^b3) << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=3 b=5 a^b=6
a=3 b=5 a^b=6
a=ffffffb6 b=ffffffc2 a^b=0x74 116
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.3. 异或(^)

运算规则：相同为0，不同为1

应用：

★ 特定位翻转（0，1互换）

例：char a=0xb6；高4位翻转，低4位不变

```
  1011 0110
^ 1111 0000      要翻转的位，本数对应位为1
  0100 0110
```

★ 两数交换

例：char a=0xb6,b=0xc2；要求a，b互换

三步：a=a^b　　b=b^a　　a=a^b

(1) a=1011 0110
b=1100 0010
a=0111 0100　　a=a^b=0x74

(2) b=1100 0010
a=0111 0100
b=1011 0110　　b=b^a=0xb6

(3) a=0111 0100
b=1011 0110
a=1100 0010　　a=a^b=0xc2

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    /* ^的应用：特定位翻转 */
    char a1=0xb6;
    cout << "a=" << hex << (int)a1 << " a^0xF0=0x" << (a1^(char)0xF0) << endl;

    /* ^的应用：两数交换 */
    char a2=0xb6, b2=0xc2;
    cout << "a=" <<hex<<(int)a2<<" b=" <<(int)b2<< endl;
    a2 = a2^b2;
    b2 = b2^a2;
    a2 = a2^b2;
    cout << "a=" <<hex<<(int)a2<<" b=" <<(int)b2<< endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=ffffffb6 a^0xF0=0x46
a=ffffffb6 b=ffffffc2
a=ffffffc2 b=ffffffb6
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.3. 异或(^)

运算规则：相同为0，不同为1

应用：

★ 简单密码传送

甲：持有secret_key
乙：持有secret_key
第三方：无法知道secret_key

甲：要发送的情报
encrypt(msg, secret_key, encryped_msg);
得到的 encryped_msg 用文件/明码等各种形式传输

乙：收到公共方式传输得到的 encryped_msg 后
decrypted(encryped_msg, secret_key, decryped_msg);
得到decryped_msg

第三方：收到 encryped_msg 后，看不懂

```
#include <iostream>
using namespace std;
void encrypt(const char* msg, const char* secret_key, char *encryped_msg)
{
    const char* p1 = msg, * p2 = secret_key;
    char* p3 = encryped_msg;
    /* 加密 */
    for (; *p1; p1++, p2++, p3++)
        *p3 = *p1 ^ *p2;
    *p3 = 0;
}
void decrypted(const char* encryped_msg, const char* secret_key, char* decryped_msg)
{
    const char* p1 = encryped_msg, * p2 = secret_key;
    char* p3 = decryped_msg;
    /* 解密(与解密操作完全一致) */
    for (; *p1; p1++, p2++, p3++)
        *p3 = *p1 ^ *p2;
    *p3 = 0;
}
int main()
{   const char* msg = "This is my student";
    const char* secret_key = "周伯通黄药师郭靖黄蓉";
    char encryped_msg[80], decryped_msg[80];

    cout << "原始信息：" << msg << endl;
    encrypt(msg, secret_key, encryped_msg);
    cout << "加密后的信息：" << encryped_msg << endl; //这个信息允许公共传播
    decrypted(encryped_msg, secret_key, decryped_msg);
    cout << "解密后的信息：" << decryped_msg << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
原始信息：This is my student
加密后的信息：偞圯砹孺啃暾蛯谳膏
解密后的信息：This is my student
```

原始信息、密钥串、加密信息，任意两个可以还原出第三个，因此要注意保护密钥串

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.4. 取反(~)

运算规则：0/1互反

例：char a=0x5c; 求~a

a=0101 1100

~a=1010 0011　　~a=0xa3

有符号10进制：-93

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x5c;
    cout << "a=" << hex << (int)a
         << " ~a=0x" << (~a) << " " << dec << (~a) << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=5c ~a=0xffffffa3 -93
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.5. 左移(<<)

运算规则：左移数据，右补0

例：char a=0x12;

a=0001 0010

0010 0100	a<<1=0x24	0x12 = 18
0100 1000	a<<2=0x48	0x24 = 36
1001 0000	a<<3=0x90	0x48 = 72
		0x90 = -112
		无符号:144

例：int b=0x12;

a<<1=0x24	
a<<2=0x48	0x24 = 36
a<<3=0x90	0x48 = 72
	0x90 = 144

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    /* char型 */
    char a=0x12;
    cout << "a=0x" << hex << int(a) << " " << dec << int(a) << endl;
    cout << "a<<1=0x" << hex << (int)(char)(a<<1) << " "
         << dec << (int)(char)(a<<1) << endl;
    cout << "a<<2=0x" << hex << (int)(char)(a<<2) << " "
         << dec << (int)(char)(a<<2) << endl;
    cout << "a<<3=0x" << hex << (int)(char)(a<<3) << " "
         << dec << (int)(char)(a<<3) << endl;
    cout << endl;

    /* 直接是int型的情况 */
    int b=0x12;
    cout << "b=0x" << hex << b << " " << dec << b << endl;
    cout << "b<<1=0x" << hex << (b<<1) << " " << dec << (b<<1) << endl;
    cout << "b<<2=0x" << hex << (b<<2) << " " << dec << (b<<2) << endl;
    cout << "b<<3=0x" << hex << (b<<3) << " " << dec << (b<<3) << endl;

    return 0;
}
```

为什么是(int)(char)(a<<1)？
先 a<<1
转为 char，此时若有溢出，则会丢弃
再转 int， 以int方式输出

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=0x12 18
a<<1=0x24 36
a<<2=0x48 72
a<<3=0xffffff90 -112

b=0x12 18
b<<1=0x24 36
b<<2=0x48 72
b<<3=0x90 144
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.5. 左移(<<)

运算规则：左移数据，右补0

例：char a=0x12；求a<<3

a=0001 0010

1001 0000 a<<3=0x90 有符号 -112

无符号144

★ 在不溢出(1不被舍去)的情况下，左移n位等于乘2的n次方(当做无符号数理解)

例：char a=0x12；求a<<4

a=0001 0010

1 0010 0000 a<<4=0x20 0x12=18 0x20=32

32+256(2^8)=288=18*16(2^4)

例：char a=0x9c；求a<<2

a=1001 1100

10 0111 0000 a<<2=0x70 0x9c=156 0x70=112

112+512(2^9)=624=156*4(2^2)

例：char a=0xc2；求a<<2

a=1100 0010

11 0000 1000 a<<2=0x8 0xc2=194 0x8=8

8+512(2^9)+256(2^8)=776=194*4(2^2)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    char a1=0x12;
    cout << "a<<4=0x" << hex << (int)(char)(a1<<4) <<" "
         << dec << (int)(char)(a1<<4) << endl;

    char a2=0x9c;
    cout << "a<<2=0x" << hex << (int)(char)(a2<<2) <<" "
         << dec << (int)(char)(a2<<2) << endl;

    char a3=0xc2;
    cout << "a<<2=0x" << hex << (int)(char)(a3<<2) <<" "
         << dec << (int)(char)(a3<<2) << endl;
    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a<<4=0x20 32
a<<2=0x70 112
a<<2=0x8 8
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.6. 右移(>>)

运算规则：右移数据，左补0（逻辑右移）

右移数据，左补符号位（算术右移）<= C/C++的位运算时算术右移

★ 算术右移，无符号数仍补0

例：char a=0x18;

a=0001 1000		0x18 = 24
0000 1100	a>>1=0xc	0xc = 12
0000 0110	a>>2=0x6	0x6 = 6
0000 0011	a>>3=0x3	0x3 = 3
0000 0001	a>>4=0x1	0x1 = 1

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x18;
    cout << "a>>1=0x" << hex << (int)(a>>1)
         << " " << dec << (int)(a>>1) << endl;
    cout << "a>>2=0x" << hex << (int)(a>>2)
         << " " << dec << (int)(a>>2) << endl;
    cout << "a>>3=0x" << hex << (int)(a>>3)
         << " " << dec << (int)(a>>3) << endl;
    cout << "a>>4=0x" << hex << (int)(a>>4)
         << " " << dec << (int)(a>>4) << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a>>1=0xc 12
a>>2=0x6 6
a>>3=0x3 3
a>>4=0x1 1
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.6. 右移(>>)

运算规则：右移数据，左补0（逻辑右移）

右移数据，左补符号位（算术右移）<= C/C++的位运算时算术右移

★ 算术右移，无符号数仍补0

★ 在不溢出(1不被舍去)的情况下，右移n位等于除2的n次方

(当作有符号数理解)

例：char a=0x84；求a>>1

a=1000 0100

1100 0010 a>>1=0xc2

最高位为1，若作为符号位，则表示负数

0x84 = -124
无符号：132
0xc2 = -62
无符号：194

a=1000 0100
-) 1
1000 0011
0111 1100

补码 => 原码
(1)减1
(2)取反
(3)绝对值
|a|=124
|a>>1|=62

a=1100 0010
-) 1
1100 0001
0011 1110

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x84; //有符号数补1 !!!
    cout << "a=0x" << hex << int(a) << " " << dec << int(a) << endl;
    cout << "a>>1=0x" << hex << (int)(a>>1) << " " << dec << (int)(a>>1) << endl;
    cout << endl;

    unsigned char b=0x84; //无符号数补0 !!!
    cout << "b=0x" << hex << int(b) << " " << dec << int(b) << endl;
    cout << "b>>1=0x" << hex << (int)(b>>1)<<" "   << dec << (int)(b>>1) << endl;

    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a=0xffffff84 -124
a>>1=0xffffffc2 -62

b=0x84 132
b>>1=0x42 66
```

§2. 基础知识

2.13. 位运算

2.13.2. 常用的位运算

2.13.2.6. 右移(>>)

例：char a=0x18;

```
a=0001 1000                (24)
  0000 1100   a>>1=0xc     (12)
  0000 0110   a>>2=0x6     (6)
  0000 0011   a>>3=0x3     (3)
  0000 0001   a>>4=0x1     (1)  溢出舍去了1
  0000 0000   a>>5=0x0     (0)  再次溢出舍去1
  0000 0000   a>>6=0x0     (0)  >>6以上都是0
```

例：char a=0x84;

```
a=1000 0100                (-124)
  1100 0010   a>>1=0xc2    (-62)
  1110 0001   a>>2=0xe1    (-31)
  1111 0000   a>>3=0xf0    (-16)  溢出舍去了1
  1111 1000   a>>4=0xf8    (-8)
  1111 1100   a>>5=0xfc    (-4)
  1111 1110   a>>6=0xfe    (-2)
  1111 1111   a>>7=0xff    (-1)
  1111 1111   a>>8=0xff    (-1)  >>8以上都是-1
```

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x18;
    int i;

    for(i=1; i<=6; i++) {
        a = a>>1;
        cout << "a>>" << i << "=0x" << hex << int(a) << " " << dec << int(a) << endl;
        }
    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a>>1=0xc 12
a>>2=0x6 6
a>>3=0x3 3
a>>4=0x1 1
a>>5=0x0 0
a>>6=0x0 0
```

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x84;
    int i;

    for(i=1; i<=8; i++) {
        a = a>>1;
        cout << "a>>" << i << "=0x" << hex << int(a) << " " << dec << int(a) << endl;
        }
    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a>>1=0xffffffc2 -62
a>>2=0xffffffe1 -31
a>>3=0xfffffff0 -16
a>>4=0xfffffff8 -8
a>>5=0xfffffffc -4
a>>6=0xfffffffe -2
a>>7=0xffffffff -1
a>>8=0xffffffff -1
```

§ 2. 基础知识

2.13. 位运算

2.13.3. 复合位运算符

&=　|=　^=　<<=　>>=

★ 将上例中 a = a>>1;

改为 a >>= 1; 结果相同

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x18;
    int i;

    for(i=1; i<=6; i++) {
        a >>= 1;
        cout << "a>>" << i << "=0x" << hex << int(a) << " " << dec << int(a) << endl;
        }
    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a>>1=0xc 12
a>>2=0x6 6
a>>3=0x3 3
a>>4=0x1 1
a>>5=0x0 0
a>>6=0x0 0
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    char a=0x84;
    int i;

    for(i=1; i<=8; i++) {
        a >>= 1;
        cout << "a>>" << i << "=0x" << hex << int(a) << " " << dec << int(a) << endl;
        }
    return 0;
}
```

读懂运行结果!!!

Microsoft Visual Studio 调试控制台

```
a>>1=0xffffffc2 -62
a>>2=0xffffffe1 -31
a>>3=0xfffffff0 -16
a>>4=0xfffffff8 -8
a>>5=0xfffffffc -4
a>>6=0xfffffffe -2
a>>7=0xffffffff -1
a>>8=0xffffffff -1
```

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.1. 算法的基本概念

3.1.1.1. 算法的定义

广义定义：为解决某个特定问题而采用的具体的方法和步骤

计算机的定义：对特定问题求解步骤的一种描述，是指令的有限序列，每个指令包含一个或几个基本操作

★ 一个问题可以有多种算法

3.1.1.2. 算法的分类

数值算法：　求解数学值(方程/函数)

大数据量计算，体现运算复杂性(逻辑相对简单)

非数值算法：除数学值外的其它领域(一般用于事务管理领域)

大数据量管理，体现逻辑复杂性(运算相对简单)

3.1.1.3. 算法的基本特征

★ 输入：有0-n个输入

★ 输出：有1-n个输出

★ 确定性：每条指令有确切含义，不产生二义性；对相同输入只能得到相同输出

★ 有穷性：每个算法在有穷步骤内完成；每个步骤都在有穷时间内完成(时间要在合理范围内)

★ 有效性：算法中所有操作都可以通过已实现的基本运算执行有限次数来实现

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.2. 程序的含义及组成

含义：可以在计算机执行的一组相关的指令及数据的集合，用来完成某一特定的任务及功能

组成：

- 数据的描述（静态）（数据结构）
  - 在程序中要指定的数据的类型及数据的组织形式
- 操作的描述（动态）（算法）
  - 对动作的描述(操作步骤)

★ 操作的对象是数据，即操作要依赖于数据

★ 数据的描述+操作的描述 = 程序

★ 数据结构 +算法 = 程序

★ 数据结构+算法+程序设计方法+语言工具+开发环境 = 程序

3.1.3. 程序的三种基本结构

顺序结构：程序按语句排列的先后次序顺序执行

选择结构：程序根据某个条件的逻辑值(真/假)来决定是否执行某些语句

循环结构：反复执行某些语句

特点：

★ 仅有一个入口

★ 仅有一个出口

★ 每一部分均可能被执行

★ 不存在死循环

§3.结构化程序设计基础

3.1.面向过程的程序设计和算法

3.1.4.算法的表示

例1：（顺序结构）

输出一个数字的平方

例2：（单分支结构）

当一个成绩小于60分时，输出"不合格"

例3：（双分支结构）

当一个成绩小于60分时，输出"不合格"，

否则输出"合格"

例4：（循环结构）

输出10个数字的平方

例5：（综合应用）

100个学生，对每个学生，当成绩小于60分时，输出"不合格"，否则输出"合格"

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.4. 算法的表示

3.1.4.1. 自然语言表示（课上未讲，简单体会一下即可）

用自然文字进行描述

例1：（顺序结构）输出一个数字的平方
步骤1：从键盘读入一个数字
步骤2：求该数的平方
步骤3：输出该数的平方

例2：（单分支结构）当一个成绩小于60分时，输出"不合格"
步骤1：从键盘读入一个数字作为成绩
步骤2：若该数小于60，转步骤3，否则直接转步骤4
步骤3：输出"不合格"
步骤4：结束

例3：（双分支结构）当一个成绩小于60分时，输出"不合格"，否则输出"合格"
步骤1：从键盘读入一个数字作为成绩
步骤2：若数小于60，转步骤3，否则转步骤4
步骤3：输出"不合格"，转步骤5
步骤4：输出"合格"
步骤5：结束

例4：（循环结构）输出10个数字的平方
步骤1：计数器置0
步骤2：若计数器大于等于10，转步骤7
步骤3：从键盘读入一个数字
步骤4：求该数的平方
步骤5：输出该数的平方
步骤6：计数器加1，转步骤2
步骤7：结束

例5：（综合应用）100个学生，对每个学生，当成绩小于60分时，输出"不合格"，否则输出"合格"
步骤1：计数器置0
步骤2：若计数器大于等于100，转步骤8
步骤3：从键盘读入一个数字作为成绩
步骤4：若该数小于60，转步骤5，否则转步骤6
步骤5：输出"不合格"，转步骤7
步骤6：输出"合格"
步骤7：计数器加1，转步骤2
步骤8：结束

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.4. 算法的表示

3.1.4.2. 流程图表示

★ 基本图形表示（参考其它书籍）

★ 缺陷：对较大的程序，过于复杂，难以阅读和修改

例1：（顺序结构）输出一个数字的平方

例2：（单分支结构）当一个成绩小于60分时吗，输出"不合格"

例3：（双分支结构）当一个成绩小于60分时，输出"不合格"，否则输出"合格"

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.4. 算法的表示

3.1.4.2. 流程图表示

例4：（循环结构）输出10个数字的平方

当型循环

思考：当i的初值为20时，左右两侧的执行结果是否一致？

直到型循环

例5：（综合应用）100个学生，对每个学生，当成绩小于60分时，输出"不合格"，否则输出"合格"

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.4. 算法的表示

3.1.4.3. 伪代码表示

介于自然语言和计算机程序语言之间的表示

★ 比流程图简练，无图形，用文字表示

★ 比自然语言直观，无二义性

★ 不能直接在计算机上执行

例1：（顺序结构）输出一个数字的平方

```
开始                BEGIN
  输入s               input s
  t = s*s             t = s*s
  打印t的值           print t
结束                END
```

例2：（单分支结构）当一个成绩小于60分时，输出"不合格"

```
BEGIN
   input s
   if s<60 then
      print  "不合格"
END
```

例3：（双分支结构）当一个成绩小于60分时，输出"不合格"，否则输出"合格"

```
BEGIN
   input s
   if s<60 then
      print  "不合格"
   else
      print  "合格"
END
```

§3. 结构化程序设计基础

3.1. 面向过程的程序设计和算法

3.1.4. 算法的表示

3.1.4.3. 伪代码表示

例4：（循环结构）输出10个数字的平方

```
BEGIN
  i=0
  while i<10 {
    input s
    t = s*s
    print t
    i=i+1
    }
END
```

```
BEGIN
  i=0
  do {
    input s
    t=s*s
    print t
    i=i+1
  } while  i<10
END
```

例5：（综合应用）100个学生，对每个学生，当成绩小于60分时，输出"不合格"，否则输出"合格"

```
BEGIN
  i=0
  while i<100 {
    input s
    if s<60 then
       print "不合格"
    else
       print "合格"
     i=i+1
     }
END
```

```
BEGIN
  i=0
  do {
    input  s
    if s<60 then
       print "不合格"
    else
       print "合格"
    i=i+1
  } while i<100
END
```

§3.结构化程序设计基础

3.1.面向过程的程序设计和算法

3.1.4.算法的表示

3.1.4.1.自然语言表示

3.1.4.2.流程图表示

3.1.4.3.伪代码表示

3.1.4.4.计算机语言表示

★ 用某种具体的程序设计语言来表示，可直接在计算机上运行，即程序

§3. 结构化程序设计基础

3.2. C++的程序结构和C++语句

3.2.1. 程序的组成

★ 一个程序由若干源程序文件(*.cpp)及头文件(*.h)组成

★ 一个源程序文件由预处理指令、全局声明及若干函数组成

★ 一个函数由若干语句组成（定义语句、执行语句）

包含的头文件
命名空间
常量定义
函数的定义
全局变量的定义
函数1
. . .
. . .
函数n

§3. 结构化程序设计基础

3.2. C++的程序结构和C++语句

3.2.2. 语句的种类

共分为四种

★ 声明语句：定义变量，放在使用该变量的语句前面

```
int a;   //定义
a=10;    //使用
```

C++中变量必须先定义、后使用

★ 执行语句：

控制语句： 9种

```
if-else
switch
break
goto
for
while
do-while
continue
return
```

1、if、while、break等称为保留字
2、C++规定，标识符不能与保留字同名

函数和流对象调用语句：

```
max(a,b);            //调用函数max的语句
cout << x << endl;   //调用流对象cout的语句
```

表达式语句：表达式+；组成

```
a=3;
```

★ 空语句：只有一个；

★ 复合语句：用一对{...}组合而成的语句，里面可以若干声明、执行、空、复合语句

§3. 结构化程序设计基础

3.3. 赋值操作

赋值表达式+;

★ C++中赋值语句和赋值表达式有区别

★ C++中赋值表达式有值，可以参与表达式的运算

```
int a;
(a=3)*10      //正确，赋值表达式，可参与运算
(a=3;)*10     //错误，赋值语句，不能参与运算
```

cpp-demo.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a;
7      (a = 3) * 10;     //正确，赋值表达式，值可参与运算
8      (a = 3;) * 10;    //错误，赋值语句，语句不能参与运算
9      return 0;
10 }
```

```
(7,13): warning C4552: "*": 未使用表达式结果
(8,11): error C2059: 语法错误: ";"
(8,12): error C2059: 语法错误: ")"
(8,14): error C2100: 非法的间接寻址
```

错误的具体含义
暂时不需要了解

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.1. 流的基本概念

流的含义：流是来自设备或传给设备的一个数据流，由一系列字节组成，按顺序排列（也称为字节流）

★ C/C++的原生标准中没有定义输入/输出的基本语句

★ C语言用printf/scanf等函数来实现输入和输出，通过#include <stdio.h>来调用

★ C++通过cin和cout的流对象来实现，通过#include <iostream>来调用

cout：输出流对象　　<<：流插入运算符

cin：输入流对象　　>>：流提取运算符

下发的附录（D-运算符优先级）

1、>>和<<是优先级第7组，称为左移/右移运算符，本处所称的流插入/流提取运算符本质上是将左移/右移运算符经过重载而得到的（重载：后续荣誉课程）

2、优先级第15组中 <<=和>>= 称为左移/右移并赋值，也称为复合按位左移/右移运算符

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.2. 输出流的基本操作

格式：cout << 表达式1 << 表达式2 << ... << 表达式n;

```
cout << "hello" << endl;
cout << "hello\n";
cout << "hello" << "\n";
cout << "hello" << '\n';
```

换行符的多种形式

★ 插入的数据存储在缓冲区中，不是立即输出，要等到缓冲区满(不同系统大小不同)或者碰到换行符("\n"/endl)或者强制立即输出(flush)才一齐输出

```
//Windows下编译运行
#include <iostream>
#include <Windows.h>
//Sleep
using namespace std;

int main()
{
    cout << "12345";
    Sleep(1000*5); //毫秒
    cout << "abcde" << endl;

    return 0;
}
```

```
//Linux下编译运行
#include <iostream>
#include <unistd.h> //sleep
using namespace std;

int main()
{
    cout << "12345" << endl;
    sleep(5); //秒
    cout << "abcde" << endl;
    return 0;
}
```

```
//Linux下编译运行
#include <iostream>
#include <unistd.h> //sleep
using namespace std;
int main()
{
    cout << "12345";
    cout.flush();
    sleep(5); //秒
    cout << "abcde" << endl;
    return 0;
}
```

仔细回想课程 Windows 和 Linux 下的演示，哪个操作系统下更符合规范？

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.2. 输出流的基本操作

格式：cout << 表达式1 << 表达式2 << ... << 表达式n;

★ 插入的数据存储在缓冲区中，不是立即输出，要等到缓冲区满(不同系统大小不同)或者碰到换行符("\n"/endl)或者强制立即输出(flush)才一齐输出

★ 默认的输出设备是显示器(可更改，称输出重定向)

★ 一个cout语句可写为若干行，或者若干语句

```
cout << "This is a C++ program." << endl;
```

```
cout << "This is " << "a C++ " << "program." << endl;
```

```
cout << "This is "
     << "a C++ "
     << "program."
     << endl;
```

一个语句分4行
前三行无分号

```
cout << "This is ";
cout << "a C++ ";
cout << "program.";
cout << endl;
```

4个语句
每行有分号

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.2. 输出流的基本操作

格式：cout << 表达式1 << 表达式2 << ... << 表达式n;

★ 一个cout的输出可以是一行，也可以是多行，多个cout的输出也可以是一行

```
cout << "hello\nhello"<<endl;
```

```
hello
hello
```

★ 一个插入运算符只能输出一个值

```
#include <iostream>
using namespace std;

int main()
{
    int a=10, b=15, c=20;
    cout << a << b << c;
    return 0;
}
```

再次强调：只输出最原始数据，其它需要自行加入

输出：101520

```
cout << a << ' ' << b << ' '<< c;
```

```
#include <iostream>
using namespace std;
int main()
{
    int a=10, b=15, c=20;

    cout <<  a,b,c;
    cout << (a,b,c);
    cout << (a,b,c) << endl;
    cout <<  a,b,c << endl;

    return 0;
}
```

第1-3句cout输出什么？
第4句cout为什么编译错？

```
error C2563: 在形参表中不匹配
error C2568: "<<": 无法解析函数重载
message : 可能是"std::basic_ostream<_Elem,_Traits> &std::endl(std::basic_ostream<_Elem,_Traits> &)"
```

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.2. 输出流的基本操作

格式：cout << 表达式1 << 表达式2 << ... << 表达式n;

★ 系统会自动判断输出数据的格式

```
#include <iostream>
using namespace std;
int main()
{
    char ch = 65;
    cout << ch << endl;
    return 0;
}
```

A

保持char类型不变，
希望输出65，如何做？

保持char类型不变，
希望输出65，不准用强制
类型转换，又该如何做？

```
#include <iostream>
using namespace std;
int main()
{
    int ch = 65;
    cout << ch << endl;
    return 0;
}
```

65

保持int类型不变，
希望输出A，如何做？

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 键盘输入的数据存储在缓冲区中，不是立即被提取，要等到缓冲区满(不同系统大小不同)或碰到回车符才进行提取

★ 默认的输入设备是键盘(可更改，称输入重定向)

★ 一行输入内容可分为若干行，或者若干语句

cin >> a >> b >> c >> d;	
cin >> a >> b >> c >> d; 1个语句分4行 前3行无分号	cin >> a; cin >> b; cin >> c; cin >> d; 4个语句 每行有分号

★ 一个提取运算符只能输入一个值

例：int a, b, c;　希望键盘输入3个整数，则：

cin >> a >> b >> c; (正确)

cin >> a, b, c;　(VS编译出错,b,c未初始化；其他编译器可执行，观察bc的值)

★ 提取运算符后必须跟变量名,不能是常量/表达式等

例：int a=1, b=1, c=1;

cin >> a+10;　(编译时语法错)

cin >> (a, b, c);　(编译正确，运行后假设输入10 20 30，发现仅c得值)　为什么？

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 输入终止条件为回车、空格、非法输入

★ 系统会自动根据cin后变量的类型按最长原则来读取合理数据

★ 变量读取后，系统会判断输入数据是否超过变量的范围，若超过则置内部的错误标记并返回一个不可信的值（不同编译器处理不同）

★ cin输入完成后，通过cin.good()/cin.fail()可判断本次输入是否正确

输入	cin.good()返回	cin.fail()返回
正确范围+回车/空格/非法输入	1	0
错误范围+回车/空格/非法输入	0	1
非法输入	0	1

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 输入终止条件为回车、空格、非法输入

```
#include <iostream>
using namespace std;

int main()
{
    short k;
    cin >> k;
    cout << "k=" << k << endl;
    cout << "cin.good()=" << cin.good() << endl;
    cout << "cin.fail()=" << cin.fail() << endl;
    return 0;
}
```

运行7次

输入		值	good()	fail()
输入：123↙	(正确+回车)	123	1	0
输入：123␣456↙	(正确+空格)	123	1	0
输入：-123m↙	(正确+非法字符)	-123	1	0
输入：m↙	(直接非法字符)	0	0	1
输入：54321↙	(超上限)	上限	0	1
输入：-40000↙	(超下限)	下限	0	1
输入：↙	(只按回车)	输入：↙继续等待输入		

★ 变量读取后，系统会判断输入数据是否超过变量的范围，若超过则置内部的错误标记并返回一个不可信的值（不同编译器处理不同）

注意：当cin.good()为0 或 cin.fail()为1 时，值不可信，不讨论具体值!!!

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 输入终止条件为回车、空格、非法输入

```
#include <iostream>
using namespace std;

int main()
{
    unsigned short k;
    cin >> k;
    cout << "k=" << k << endl;
    cout << "cin.good()=" << cin.good() << endl;
    cout << "cin.fail()=" << cin.fail() << endl;
    return 0;
}
```

	值	good()	fail()
输入：-65536↙	65535	0	1
输入：-65535↙	1	1	0
输入：-1234↙	64302	1	0
输入：-1↙	65535	1	0
输入：70000↙	65535	0	1

注：

1、上面的三个65535中，有两个不可信，为什么？

2、无符号的错误表现与有符号不一样，规律自行总结（包括各种整型）

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 输入终止条件为回车、空格、非法输入

```
#include <iostream>
using namespace std;

int main()
{
    char c1, c2;
    int a;
    float b;
    cin >> c1 >> c2 >> a >> b;
    cout<< c1<<' '<< c2<<' '<< a<<' '<< b<< endl;
    return 0;
}
```

输入：1234␣56.78
输出：

输入：1␣2␣34␣56.78
输出：

```
#include <iostream>
using namespace std;

int main()
{
    short k;
    cin >> k;
    cout << "k=" << k << ' ' << cin.good() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
70000
k=32767 0
```

注意：输入时给出超范围值
赋值时给出超范围值
情况不相同!!!

```
#include <iostream>
using namespace std;

int main()
{
    short k;
    k = 54321;  // k = 70000;
    cout << "k=" << k << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
k=4464
```

```
(6,14): warning C4305: "=": 从"int"到"short"截断
(6,9): warning C4309: "=": 截断常量值
```

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.3. 输入流的基本操作

格式：cin >> 变量1 >> 变量2 >> ... >> 变量n;

★ 字符型变量只能输入图形字符(33-126)，不能以转义符方式输入
(单双引号、转义符全部当作单字符)

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cin >> ch;
    cout << cin.good() << ' ' << (int)ch <<endl;
    return 0;
}
```

输入：abc
输出：
输入：\n
输出：
输入：'a'
输出：
输入：Ctrl+Z
输出：
输入：Ctrl+C
输出：

★ 浮点数输入时，可以是十进制数或指数形式，只取有效位数(4舍5入)

```
#include <iostream>
using namespace std;
int main()
{
    float f;
    cin >> f;
    cout << cin.good() << ' ' << f <<endl;
    return 0;
}
```

输入：
123.456789
输出：

输入：12e3
输出：

输入：1.2e40
输出：

★ cin不能跟endl，否则编译错

error C2679: 二元“>>”: 没有找到接受“overloaded-function”类型的右操作数的运算符(或没有可接受的转换)

特别提示：有时候一大批错误指向系统文件，原因并不是系统没装好/感染病毒/被破坏…

§3.结构化程序设计基础

3.4.C++的输入与输出

3.4.4.在输入输出流中使用格式化控制符

3.4.4.1.C++方式输入输出的格式化控制

dec	设置整数为10进制
hex	设置整数为16进制
oct	设置整数为8进制
setbase(n)	设置整数为n进制(n = 8,10,16)
setfill(c)	设置填充字符，c可以是字符常量或字符变量
setprecision(n)	设置实数的精度为n位，在以一般十进制形式输出时，n代表有效数字， 在以fixed(固定小数位)形式和scientific(指数)形式输出时，n为小数位数
setw(n)	设置字段宽度为n
setiosflags(ios::fixed)	设置浮点数以固定的小数位数显示
setiosflags(ios::scientific)	设置浮点数以科学计数法(即指数形式)显示
setiosflags(ios::left)	输出数据左对齐
setiosflags(ios::right)	输出数据右对齐
setiosflags(ios::skipws)	忽略前导的空格
setiosflags(ios::uppercase)	在以科学计数法输出E和十六进制输出字母X时以大写表示
setiosflags(ios::showpos)	输出正数时，给出"+"号
resetiosflags()	终止已设置的输出格式状态，括号内为具体内容

§3.结构化程序设计基础

3.4.C++的输入与输出

3.4.4.在输入输出流中使用格式化控制符

3.4.4.2.C方式格式化输出函数printf的基本理解(补充)

形式：printf(格式控制表列，输出表列)；

格式控制表列的内容：

格式说明：以%开始+格式字符，表示按格式输出

普通字符(含转义符)：原样输出

输出表列：

要输出的数据（常量、变量、表达式、函数）

常用的格式符种类：

printf所用的格式字符的种类：

d,i	带符号的十进制形式整数(正数不带+)
o	八进制无符号形式输出整数(不带前导0)
x,X	十六进制无符号形式输出整数(不带前导0x)
u	十进制无符号形式输出整数
c	以字符形式输出(一个字符)
s	输出字符串
f	以小数形式输出浮点数
e,E	以指数形式输出浮点数
g.G	从f，e中选择宽度较短的形式输出浮点数

printf所用的附加格式字符的种类：

字母l	表示长整型整数，用于d,o,x,u前
字母h	表示短整型整数，用于d,o,x,u前
正整数m	表示输出数据的宽度
正整数.n	对浮点数，表示n位小数；对字符串，表示前n个字符
-	输出左对齐

§3.结构化程序设计基础

3.4.C++的输入与输出

3.4.4.在输入输出流中使用格式化控制符

3.4.4.3.C方式格式化输入函数scanf的基本理解(补充)

VS系列认为scanf函数是不安全的输入，因此缺省禁止使用，如果想继续使用，必须在源程序一开始加定义#define _CRT_SECURE_NO_WARNINGS

形式：scanf(格式控制表列，地址表列);

格式控制表列的内容：

格式说明：以%开始+格式字符，表示按格式输入

普通字符(含转义符)：原样输入

地址表列：

&表示取地址

&变量名：取该变量的内存地址

&不能跟表达式/常量

常用的格式符种类：

scanf所用的格式字符的种类：

d,i	输入带符号的十进制形式整数
o	输入八进制无符号形式整数(不带前导0)
x,X	输入十六进制无符号形式整数(不带前导0x)
u	输入十进制无符号形式整数
c	输入单个字符
s	输入字符串
f	输入小数/指数形式的浮点数
e,E,g,G	同f

scanf所用的附加格式字符的种类：

字母l	输入长整型数，用于d,o,x,u前；输入double型数，用于f,e,g前
h	输入短整型数，用于d,o,x,u前
正整数n	指定输入数据所占的宽度
*	本输入项不赋给相应的变量

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.5. 字符的输入和输出

3.4.5.1. 字符输出函数putchar

形式：putchar(字符变量/常量)

功能：输出一个字符

```
char a='A';
```

```
putchar(a);          putchar('A') ;
putchar('\x41');     putchar('\101');
```

这四个都在屏幕上输出A

★ 加#include <cstdio>或#include <stdio.h>

（目前两编译器均可不要）

★ 返回值是int型，是输出字符的ASCII码，可赋值给字符型/整型变量

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char ret1;
    cout << (ret1 = putchar('A')) << endl;
    int ret2;
    cout << (ret2 = putchar('B')) << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
AA
B66
```

为什么是这个输出？想明白！

写测试程序验证以下问题：

问题：如何证明putchar()的返回值是int而不是char？是否有多种方法？

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.5. 字符的输入和输出

3.4.5.2. 字符输入函数getchar

形式：getchar()

功能：输入一个字符(给指定的变量)

★ 加#include <cstdio>或#include <stdio.h>

(目前两编译器均可不要)

★ 返回值是int型，是输入字符的ASCII码，可赋值给字符型/整型变量

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char ch;
    ch = getchar();
    cout << ch << endl;
    return 0;
}
```

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char ch;
    cout << (ch = getchar()) << endl;
    return 0;
}
```

左右程序
1、假设键盘输入：a，则输出：？
2、左右蓝色框中语句是否等价？

写测试程序验证以下问题：
问题1：如何证明getchar()的返回值是int而不是char？是否有多种方法？
问题2：在不允许定义char型变量的前提下，如何使getchar()的输出为字符

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.5. 字符的输入和输出

3.4.5.2. 字符输入函数getchar

形式：getchar()

功能：输入一个字符(给指定的变量)

★ 输入时有回显，输入后需按回车结束输入(若直接按回车则得到回车的ASCII码)

★ 可以输入空格，回车等cin无法处理的非图形字符，但仍不能处理转义符

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    cout<< getchar() << endl;
    return 0;
}
```

输入：␣
输出：32

输入：↙
输出：10

输入：\n
输出：92

★ 调试程序时，可以用getchar()来延迟结束

(再次强调，本课程的作业严禁在程序最后用 getchar() / system("pause") / cin>>a 等来暂停)

★ cin/getchar 等每次仅从输入缓冲区中取需要的字节，多余的字节仍保留在输入缓冲区中供下次读取

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    cout << "--Step1--" << endl;
    cout << getchar()   << endl;
    cout << "--Step2--" << endl;
    cout << getchar()   << endl;
    cout << "--Step3--" << endl;
    cout << getchar()   << endl;
    cout << "--Step4--" << endl;
    cout << getchar()   << endl;
    return 0;
}
```

按下面方式执行3次，观察运行现象及结果

1. 每次输入一个回车
2. 每次输入一个字母并按回车
3. 第一次即输入4个以上字母并按回车

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.5. 字符的输入和输出

3.4.5.3. 字符输入函数_getch与_getche

```
#include<iostream>
#include<conio.h>  //_getch()/_getche()用到的头文件
using namespace std;
int main()
{
    char ch;
    ch = _getch();  //换成为_getche()/getchar()对比
    cout << (int)ch << endl;

    return 0;
} //注意：测试时不能是中文输入法
```

★ 几个字符输入函数的差别

getchar ：有回显，不立即生效，需要回车键

_getche ：有回显，不需要回车键

_getch ：无回显，不需要回车键

★ 在Dev C++中

getch() ⇔ _getch()

getche() ⇔ _getche()

§3. 结构化程序设计基础

3.5. 编写顺序结构的程序

例：求一元二次方程的根

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{   float a, b, c, x1, x2;
    cin >> a >> b >> c;
    x1 = (-b + sqrt(b*b-4*a*c) )/(2*a);
    x2 = (-b - sqrt(b*b-4*a*c) )/(2*a);
    cout << "x1=" << x1 << endl;
    cout << "x2=" << x2 << endl;
    return 0;
}
```

1、sqrt是系统提供的开方函数，需包含<math.h>或<cmath>
2、b^2的表示方法　b*b
3、4ac的表示方法 4*a*c
4、2*a必须加()
5、在b^2-4ac<0的情况下，出错
6、程序执行时无任何提示即等待输入

§3. 结构化程序设计基础

3.5. 编写顺序结构的程序

例：从键盘输入一个大写字母，要求改为小写字母输出

```
#include <iostream>
using namespace std;
int main()
{
    char c1, c2;

    cin >> c1;

    c2 = c1 + 32;

    cout << c1 << ' ' << c2 << endl;

    return 0;
}
```

```
c1=getchar();
scanf("%c", &c1);
```

```
c2 = c1 + 'a' - 'A';
```

```
printf("%c %c",c1,c2);
```

1、程序执行时无任何提示即等待输入
2、若输入的不是大写字母出错

§3.结构化程序设计基础

3.6.关系运算和逻辑运算

3.6.1.关系运算和关系表达式

3.6.1.1.关系运算

含义：将两个值进行比较，取值为"真"（用1表示）

"假"（用0表示）

3.6.1.2.关系运算符

种类：

< <= > >= 优先级相同（优先级第8组）

== != 优先级相同（优先级第9组）

优先级和结合性：附录D

3.6.1.3.关系表达式

含义：用关系运算符将两个表达式(算术、逻辑、赋值、关系)连接起来，称为关系表达式

```
10+20 > 30      //比较算术表达式和常量的值
(a=20)>(b=30)   //比较两个赋值表达式的值
```

关系表达式的值：

"真" - 1

"假" - 0

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.1. 关系运算和关系表达式

3.6.1.3. 关系表达式

关系表达式的值：

"真" - 1

"假" - 0

```
int a=1, b=2, c;
c=a>b    c=0
c=a<b    c=1
```

掌握用程序验证的方法

```
#include <iostream>
using namespace std;
int main()
{   int a=1, b=2, c;
    c = a>b;
    cout << c << endl;
    c = a<b;
    cout << c << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{   int a=1, b=2, c=3, d;
    d = a>b>c;
    cout << d << endl;
    d = a<b<c;
    cout << d << endl;
    d = b>a<c;
    cout << d << endl;
    return 0;
} //VS下有三个warning
```

Microsoft

```
0
1
1
```

```
#include <iostream>
using namespace std;
int main()
{   int a=3, b=2, c=1, d;
    d = a>b>c;
    cout << d << endl;
    d = a<b<c;
    cout << d << endl;
    d = b>a<c;
    cout << d << endl;
    return 0;
} //VS下有三个warning
```

Microsoft

```
0
1
1
```

```
d=1
d=0
d=0
```

1、左右两个程序编译时均有warning

```
">": 在操作中使用类型"bool"不安全
"<": 在操作中使用类型"bool"不安全
"<": 在操作中使用类型"bool"不安全
```

2、为什么左右结果一样?

3、a>b>c 正确的求解方式是什么?

4、C/C++的实际求值过程是怎样的?

注：一定要搞明白，为什么右侧程序的运行结果不是蓝色!!!

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.1. 关系运算和关系表达式

3.6.1.3. 关系表达式

关系表达式的值：

"真" - 1

"假" - 0

★ 关系表达式的值可以做为整型参与运算

```
int a, b, c;
cin >> a >> b;
c = a > b;
```

赋值表达式，值为0/1

```
10+(a<=b)*2
```

算术表达式，值为10/12

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.1. 关系运算和关系表达式

3.6.1.3. 关系表达式

关系表达式的值：

★ 关系表达式的值可以做为整型参与运算

★ 实数参与关系运算时要考虑误差

```
//第2章 IEEE754 作业中的例子

例1：float f=100.25;
32bit机内表示__0100 0010 1100 1000 1000 0000 0000 0000_
        尾数是_____________100 1000 1000 0000 0000 0000_
尾数转换为十进制小数是__0.56640625___
尾数表示的十进制小数是__1.56640625___(加整数部分的1后)
    1.56640625 x 2^6 = 100.25 （未体现出误差）

例2：float f = 1234567.7654321;
32bit机内表示__0100 1001 1001 0110 1011 0100 0011 1110_
        尾数是_____________001 0110 1011 0100 0011 1110__
尾数转换为十进制小数形式是__0.1773755503845214844___
尾数表示的十进制小数形式是__1.1773755503845214844___(加1后)
    1.17737555503845214844 * 2^20 = 1234567.75 （体现出误差）
```

```
#include <iostream>
using namespace std;
int main()
{
   float f1=100.25;
   cout << (f1-100.25)  << endl;
   cout << (f1==100.25) << endl;

   float f2=1234567.7654321;
   cout << (f2-1234567.7654321) << endl;
   cout << (f2==1234567.7654321) << endl;
   return 0;
}
```

第2章 IEEE754 作业中的例子

1、为什么第5行double赋值给float无warning，但第9行有？

2、如何理解==的结果是 1/0 ？

3、在不了解IEEE754的情况下，是否有通用的方法避免误差？

Microsoft Visual Studio 调试控制台

```
0
1
-0.0154321
0
```

```
demo.cpp(9,28): warning C4305: “初始化”: 从“double”到“float”截断
```

```
#include <iostream>
#include <cmath> //fabs需要，VS可不加
using namespace std;
int main()
{
   float f1=100.25;
   cout << (fabs(f1-100.25) < 1e-6) << endl;

   float f2=1234567.7654321;
   cout << (fabs(f2-1234567.7654321) < 1e-1) << endl;
   cout << (fabs(f2-1234567.7654321) < 1e-2) << endl;
   return 0;
}
```

fabs是求绝对值的系统函数

通用方法：

当两数相减的绝对值小于某个值则认为相等

Microsoft

```
1
1
0
```

问：为什么cout时要加红色的一对括号 ？

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.1. 关系运算和关系表达式

3.6.1.3. 关系表达式

关系表达式的值：

★ 关系表达式的值可以做为整型参与运算

★ 实数参与关系运算时要考虑误差

```
#include <iostream>
#include <cmath> //VS可不加
using namespace std;
int main()
{
    float b = 1.1; //有warning
    cout << (b - 1.1) << endl;
    cout << (b == 1.1) << endl;
    cout << (fabs(b - 1.1) < 1e-6) << endl;

    float c = 1.0; //无warning
    cout << (c - 1.0) << endl;
    cout << (c == 1.0) << endl;
    cout << (fabs(c - 1.0) < 1e-6) << endl;

    return 0;
}
```

2.38419e-08

0
1

0
1
1

结论：

- 用==判断实型数是否相等，某些情况下可能与预期结果不符合，因此禁用
- fabs()函数是通用的保证实型数误差方法，但是使用时不要超过有效位数限定

```
#include <iostream>
#include <cmath> //VS可不加
using namespace std;
int main()
{   double f1=123.456789012345678;
    double f2=123.456789123456789;
    cout << (f1==f2) << endl;
    cout << (fabs(f1-f2)<1e-6) << endl;
    cout << (fabs(f1-f2)<1e-7) << endl;

    float g1=123.456789012345678;
    float g2=123.456789123456789;
    cout << (g1==g2) << endl;
    cout << (fabs(g1-g2)<1e-6) << endl;
    cout << (fabs(g1-g2)<1e-7) << endl;

    return 0;
} //有warning
```

warning C4305: “初始化”: 从“double”到“float”截断
warning C4305: “初始化”: 从“double”到“float”截断

0
1
0

1
1
1

1、为什么不等的数判断==返回1？

2、为什么小数点后第7位不同但判断1e-7返回1？

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.1. 关系运算和关系表达式

3.6.1.3. 关系表达式

关系表达式的值：

★ 关系表达式的值可以做为整型参与运算

★ 实数参与关系运算时要考虑误差

结论：

● 实数输出时可以任意指定位数，但超过有效位数限定的值不可信

```
#include <iostream>
#include <iomanip>
#include <cmath> //VS可不加
using namespace std;

int main()
{
    cout << setiosflags(ios::fixed); //指定fixed输出

    double d1 = 123.456789012345678;
    cout << setprecision(15) << d1 << endl;
    cout << setprecision(20) << d1 << endl;

    float f1 = 123.456789012345678;
    cout << setprecision(15) << f1 << endl;
    cout << setprecision(20) << f1 << endl;

    cout << endl;

    double d2 = 123.456789123456789;
    cout << setprecision(15) << d2 << endl;
    cout << setprecision(20) << d2 << endl;

    float f2 = 123.456789123456789;
    cout << setprecision(15) << f2 << endl;
    cout << setprecision(20) << f2 << endl;

    return 0;
} //VS有两个warning
```

```
warning C4305: “初始化”: 从“double”到“float”截断
warning C4305: “初始化”: 从“double”到“float”截断
```

Microsoft Visual Studio 调试控制台

```
123.456789012345681
123.45678901234568058953
123.456787109375000
123.45678710937500000000

123.456789123456787
123.45678912345678668316
123.456787109375000
123.45678710937500000000
```

§3.结构化程序设计基础

3.6.关系运算和逻辑运算

3.6.1.关系运算和关系表达式

3.6.1.3.关系表达式

关系表达式的值：

★ 关系表达式的值可以做为整型参与运算

★ 实数参与关系运算时要考虑误差

★ 注意=和==的区别!!!

```
int a;
a==10;   关系表达式，值为0/1
a=10;    赋值表达式，值为10
```

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.2. 逻辑常量和逻辑变量(C++特有，C无)

逻辑常量：true / false

逻辑变量：bool 变量名

定义后赋值：	定义时赋初值：
bool f;	bool f=false;
f=true;	

★ 在内存中占1个字节，表示为整型值，取值只有0/1(true=1/false=0) sizeof(bool) => 1

★ cin时只能输入0/1，输出时按整型量进行处理

```
#include <iostream>
using namespace std;

int main()
{
    bool k;

    cin >> k;
    cout << cin.good() << ' ' << k << ' ' << (int)k << endl;

    return 0;
}
```

输入:	输出:		
0	1	0	0
1	1	1	1
123	0	1	1
true	0	0	0
false	0	0	0

这三种都是错误的输入方法
不同编译器下表现可能不同，
虽然值看似符合预期，但不可信

```
#include <iostream>
using namespace std;
int main()
{
    bool f;

    cout << typeid(bool).name()<<endl;
    cout << sizeof(bool) << endl;
    cout << sizeof(f) << endl;
    cout << sizeof(true) << endl;
    cout << sizeof(false) << endl;
    cout << sizeof("true") << endl;
    cout << sizeof("false") << endl;
    return 0;
}
```

最后两项是什么？

Microsoft

```
bool
1
1
1
1
5
6
```

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.2. 逻辑常量和逻辑变量(C++特有，C无)

逻辑常量：true / false

逻辑变量：bool 变量名

★ 在内存中占1个字节，表示为整型值，取值只有0/1(true=1/false=0) sizeof(bool) => 1

★ cin时只能输入0/1，输出时按整型量进行处理

★ 赋值及运算时，按"非0为真零为假"的原则进行

```
#include <iostream>
using namespace std;

int main()
{
    bool k;
    k=123;
    cout << k << ' ' << (int)k << endl;
    k=0;
    cout << k << ' ' << (int)k << endl;
    k=256;
    cout << k << ' ' << (int)k << endl;
    return 0;
}
```

Microsoft

```
1 1
0 0
1 1
```

```
warning C4305: "=" : 从"int"到"bool"截断
warning C4305: "=" : 从"int"到"bool"截断
```

256 = 0..01 00000000 （4字节）
即低8bit为0
因此，本例证明bool赋值时
不是多字节赋少字节的规则，
而是"非0为真0为假"

★ 可按整型值(0/1)参与表达式的运算

```
bool f=true;
int  a=10;
a=a+f;
cout << a << endl;    11
```

★ C方式（源程序后缀为.c）加 #include <stdbool.h>后，
也可使用bool/true/false

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.3. 逻辑运算和逻辑表达式

3.6.3.1. 逻辑运算

含义：将多个关系表达式或逻辑量共同进行逻辑运算，取值为"真"/"假"　　1/0

3.6.3.2. 逻辑运算符

&&	A&&B	均为真，取值为真　(优先级第13组)
||	A||B	有一个为真，值为真(优先级第14组)
!	!A	取反(单目运算符)　(优先级第3组)

优先级与结合性：附录D

3.6.3.3. 逻辑表达式

含义：将多个表达式或逻辑量用逻辑运算符连接起来

逻辑表达式的值：

★ 取值

真　1

假　0

★ 表达式参与运算时

非0　真

0　假

逻辑运算的真值表（取值时）：

a	b	!a	!b	a&&b	a||b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

逻辑运算的真值表（表达式参与运算时）：

a	b	!a	!b	a&&b	a||b
非0	非0	0	0	1	1
非0	0	0	1	0	1
0	非0	1	0	0	1
0	0	1	1	0	0

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.3. 逻辑运算和逻辑表达式

3.6.3.3. 逻辑表达式

含义：将多个表达式或逻辑量用逻辑运算符连接起来

例：a=4 b=5

```
a  && b  = 1
a  || b  = 1
!a || b  = 1
!a && b  = 0
```

例：4 && 0 || 2 = 1

例：5>3 && 2 || 8<4 - !0 = 1

自行给出求值顺序

1、按优先级结合性得到的求解顺序

2、再结合本文档下页的短路运算得到的求解顺序

问：应该是哪个？如何验证？

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.3. 逻辑运算和逻辑表达式

3.6.3.3. 逻辑表达式

含义：将多个表达式或逻辑量用逻辑运算符连接起来

逻辑表达式的值：

★ 取值

★ 表达式参与运算时

★ 仅当必须执行下一个逻辑运算符才能求出解时，才执行该运算符，否则不执行(短路运算)

a&&b&&c　若a=0，值必为0，b,c不求解

a||b||c　若a=1，值必为1，b,c不求解

```
#include <iostream>
using namespace std;
int main()
{
    int a=1, b=2, c=3, d=4, m=1, n=1;
    cout << "m=" << m << " n=" << n << endl;
    (m=a>b)&&(n=c>d); //m=0,n不再求解,保持原值
    cout << "m=" << m << " n=" << n << endl;
    return 0;
}
```

```
m=1   n=1
m=0   n=1
```

★ 容易犯的错误：表示 t在(70,80]之间

错误：70<t<=80　　!!!!!

正确：t>70 && t<=80

★ 常见的等价表示

a==0　<=>　!a

a!=0　<=>　a

例：若某年是闰年，则符合下列两个条件之一

（1）被4整除，不被100整除

（2）被4整除，又被400整除

各种形式的表示：

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

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

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

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

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

真：闰年

假：非闰年

例：若某年不是闰年，则符合下列两个条件之一

（1）不被4整除

（2）被100整除，不被400整除

条件1：(year%4!=0)

条件2：(year%100==0)&&(year%400!=0)

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

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

为什么条件2的整体括号(蓝色)可以省略？

真：非闰年

假：闰年

§3. 结构化程序设计基础

3.6. 关系运算和逻辑运算

3.6.3. 逻辑运算和逻辑表达式

3.6.3.4. 按位与/或运算

附录D：

按位与：& 两数按对应bit位与，对应bit位均为1则取值为1(真)　(优先级第10组)

按位或：| 两数按对应bit位或，对应bit位有1个为1则值为1(真)　(优先级第12组)

```
#include <iostream>
using namespace std;
int main()
{
    short a=3, b=4;
    cout << (a && b) << endl;
    cout << (a &  b) << endl; //按位&
    cout << (a || b) << endl;
    cout << (a |  b) << endl; //按位|

    return 0;
}
```

```
Micro
1
0
1
7
```

常见错误：

&&/||写成&/|，某些情况下也表现正确

本例：a||b写成a|b，结果均为逻辑真

```
  0000 0000 0000 0011 (a)
& 0000 0000 0000 0100 (b)
-----------------------------
  0000 0000 0000 0000 (0)
```

```
  0000 0000 0000 0011 (a)
| 0000 0000 0000 0100 (b)
-----------------------------
  0000 0000 0000 0111 (7)
```

问题：

1、除&和|外，是否还有其它位运算符？

还有 ^(异或) ~(取反) >>(右移) <<(左移)

2、两数长度不等时(例：short | int)，该如何运算？

低位对齐，高位补符号位/0

★ 不再展开，后续面向对象等课程再学习，有兴趣可以自行了解

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.1. if语句的三种形式

3.7.1.1. 单支语句

```
if (表达式) {
    语句序列;
    }
```

★ 当表达式为真时执行语句序列，为假则不执行

★ 表达式可以是任意类型，但按逻辑值求解(非0为真0为假)

★ 表达式后无；(表达式和表达式语句的区别)

编译错

```
if (表达式;) {
    语句序列;
    }
```

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    cout << "请输入成绩(0-100)" << endl;
    cin >> i;
    if (i < 60;) {
        cout << "不及格" << endl;
    }
    cout << "程序结束" << endl;
    return 0;
}
```

```
warning C4552: ”<“: 未使用表达式结果
error C2429: 语言功能 "if/switch 中的 init-statement" 需要编译器标志 "/std:c++17"
error C2059: 语法错误: “)”
error C2143: 语法错误: 缺少“;”(在“{”的前面)
```

★ 若语句序列中只有一个语句，{}可省

```
if (i<60) {
    cout << "不及格" << endl;
    }
```

等价于

```
if (i<60)
    cout << "不及格" << endl;
```

★ 整个单支语句可以作为一个语句来看待(复合语句)

例：输入一个成绩，若不及格，则打印提示信息

```
int main()
{
   int i;
   cout<<"请输入成绩(0-100)"<<endl;
   cin >> i;
   if (i<60) {
      cout << "不及格" << endl;
      }
   cout << "程序结束" << endl;
   return 0;
}
```

输入:37 输出:不及格
程序结束

输入:74 输出:程序结束

```
int main()
{
   int i;
   cout<<"请输入成绩(0-100)"<<endl;
   cin >> i;
   if (i<60) {
      cout << "不及格" << endl;
   cout << "程序结束" << endl;
   } //注意：括号位置已变!!!
   return 0;
}
```

注意：括号位置对程序正确性影响很大

输入:37 输出:不及格
程序结束

输入:74 输出:

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.1. if语句的三种形式

3.7.1.1. 单支语句

3.7.1.2. 双支语句

```
if (表达式) {
    语句序列1;
    }
else {
    语句序列2;
    }
```

★ 当表达式为真时执行语句序列1，为假则执行语句序列2

★ 表达式可以是任意类型，但按逻辑值求解(非0为真0为假)

★ 表达式后无；

★ 若语句序列1、2中只有一个语句，{}可省

★ 整个双支语句可以作为一个语句来看待，中间不允许插入任何的其它语句

```
if (i<60) {
    ...;
    }
cout << "..." << endl;
else {
    ...;
    }
```

编译错，提示else没有对应的if

cpp-demo.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int i;
7      cout << "请输入成绩(0-100)" << endl;
8      cin >> i;
9      if (i < 60) {
10         cout << "不及格" << endl;
11     }
12     cout << "Hello" << endl;
13     else {
14         cout << "及格" << endl;
15     }
16     cout << "程序结束" << endl;
17     return 0;
18 }
19
```

人的思维：整体看，cout插在if-else中间，cout多余

编译器思维：从上到下看，单if、并列的cout，因此else非法

例：输入一个成绩，根据分数是否及格打印相应的提示信息

```
int main()
{
    int i;
    cout << "请输入成绩(0-100)" << endl;
    cin >> i;
    if (i<60)
        cout << "不及格" << endl;
    else
        cout << "及格" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

注意：若没有括号，则只有第1句语句属于if-else

```
int main()
{
    int i;
    cout << "请输入成绩(0-100)" << endl;
    cin >> i;
    if (i<=60)
        cout << "不及格" << endl;
    else
        cout << "及格" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

即使能保证输入正确[0..100]，仍有一个数据的运行结果是错误的

程序设计中的几个很重要的概念：

1、部分测试数据的正确性不代表程序一定是正确的，只是错误没有暴露出来而已

2、错误的发现可能需要相当长时间，时间不是证明没有错误的借口

3、程序的测试很重要，测试的目的是为了证明程序有错误，而不是为了证明程序是正确的

4、复杂程序无法保证完全正确，因此如何快捷方便地更正错误很重要

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.1. if语句的三种形式

3.7.1.1. 单支语句

3.7.1.2. 双支语句

3.7.1.3. 多支语句

★当表达式1为真时，执行语句序列1，为假时，则判断表达式2

当表达式2为真时，执行语句序列2，为假时，则判断表达式3

...

当表达式n为真时，执行语句序列n，为假时，则执行语句序列n+1

★ 表达式可以是任意类型，按逻辑值求解（非0为真0为假）

★ 表达式后无；

★ 若语句序列中只有一个语句，{}可省

★ 整个多支语句可以作为一个语句来看待，中间不允许插入任何的其它语句

```
if (表达式1) {
    语句序列1;
    }
else if (表达式2) {
    语句序列2;
    }
...
else if (表达式n) {
    语句序列n;
    }
else {
    语句序列n+1;
    }
```

例：输入一个分数，根据所处的分数段0-59,60-69,70-79,80-89,90-100分别打印"优"、"良"、"中"、"及格"、"不及格"，其它则打印"输入错误"

```
int main()
{    int i;
     cout << "请输入成绩(0-100)" << endl;
     cin >> i;
     if (i>=90 && i<=100)
         cout << "优" << endl;
     else if (i>=80 && i<90)
         cout << "良" << endl;
     else if (i>=70 && i<80)
         cout << "中" << endl;
     else if (i>=60 && i<70)
         cout << "及格" << endl;
     else if (i>=0 && i<60)
         cout << "不及格" << endl;
     else
         cout << "输入错误" << endl;
     cout << "程序结束" << endl;
     return 0;
}
```

问题1：
能否改为i<=89？哪个更好？

问题2：
能否改为i<=90？运行是否正确？

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.1. if语句的三种形式

3.7.2. if语句的嵌套

各红色？的语句在什么情况下执行
各蓝色？的语句在什么情况下执行
各粉色？处若有语句，是否正确？

★ {}的匹配原则：自上而下，忽略 { ，以 } 为准向上匹配未配对的 {

★ {}的匹配可用栈理解，遇 { 进栈，遇 } 则栈顶 { 出栈并匹配为一对,若到最后仍有 { 或 } 未配对则语法错

★ if/else匹配原则：以{}成对为基准，将else与它上面最近的if配对

(if不一定有else，else一定要有if，因此和{}的匹配方式不完全相同)

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

初始：空

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

1进栈

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

2进栈

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

遇3，2出，匹配

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

4进栈

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

遇5, 4出, 匹配

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

遇6，1出，匹配

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

7进栈

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

8进栈

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

遇9, 8出, 匹配

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

遇10，7出，匹配

例：

```
if (表达式) { 1
    if (表达式) { 2
        } 3
    else { 4
        } 5
    } 6
else { 7
    if (表达式) { 8
        } 9
    } 10
```

栈空，外面也无未匹配的}
正确结束

例：输入一个分数，根据所处的分数段0-59,60-69,70-79,80-89,90-100分别打印"优"、"良"、"中"、"及格"、"不及格"

```
if (i>=80)
    if (i>=90)
        printf("优\n");
    else
        printf("良\n");
else
    if (i>=60)
        if (i>=70)
            printf("中\n");
        else
            printf("及格\n");
    else
        printf("不及格\n");
```

单个语句都加{}

```
if (i>=80) {
    if (i>=90) {
        printf("优\n");
        }
    else {
        printf("良\n");
        }
    }
else {
    if (i>=60) {
        if (i>=70) {
            printf("中\n");
            }
        else {
            printf("及格\n");
            }
        }
    else {
        printf("不及格\n");
        }
    }
```

★ 假设输入正确

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.3. 条件运算符和条件表达式

3.7.3.1. 引入

if - else语句中语句序列均为一个赋值语句且给同一变量赋值的，可以使用条件运算符

3.7.3.2. 形式

表达式1 ？表达式2 ：表达式3

★ C/C++中唯一的一个三目运算符(附录D：优先级第15组)

★ 表达式1按逻辑值求解，若为真，求解表达式2并使整个条件表达式的值为表达式2的值，
否则，求解表达式3并使整个条件表达式的值为表达式3的值

```
例：int a,b,max;
    cin >> a >> b;

    if (a>b)
        max = a;
    else
        max = b;

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

```
例：int a,b;
    cin >> a >> b;

    if (a>b)
        cout << "max=" << a << endl;
    else
        cout << "max=" << b << endl;

    a > b ? cout << "max=" << a << endl :
            cout << "max=" << b << endl;

    cout << "max=" << (a>b?a:b) << endl;

    printf("max=%d", a>b?a:b);
```

```
#include <iostream>
using namespace std;
int main()
{
    int a = 1, b = 2;

    //编译报错
    a == 1 ? "Hello" : 123;

    //编译报错
    a > b ? cout << a : printf("%d", b);

    //编译正确
    a == 1 ? 'A' : 123;
    return 0;
}
```

```
error C2446: ":": 没有从"int"到"const char [6]"的转换
message : 从整型强制转换为指针类型要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
 error C2678: 二进制"?": 没有找到接受"std::basic_ostream<char,std::char_traits<char>>"类型的左操作数的运算符(或没有可接受的转换)
 message : 可能是"内置 C++ operator?(int, int)"
 message : 尝试匹配参数列表"(std::basic_ostream<char,std::char_traits<char>>, int)"时
```

★ 表达式1、2、3的类型可以不同，但2、3的类型必须相容
(否则无法确定条件表达式的值类型)

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.4. 多分支选择结构和switch语句

3.7.4.1. 作用

替代多重if语句的嵌套，增强可读性

3.7.4.2. 形式

见右侧

```
switch(整型表达式) {
    case 整型常量表达式1:
        语句序列1;
    case 整型常量表达式2:
        语句序列2;
    ...
    case 整型常量表达式n:
        语句序列n;
    default:
        语句序列n+1;
}
```

3.7.4.3. 使用

★ 表达式可以是任何类型，最终取值为整型即可

★ 当整型表达式的取值与整型常量表达式1-n中的任意一个相等时,执行对应的语句序列;否则执行default后的语句序列

（不能是实数，因为无法直接判断相等）

★ 各整型常量表达式的值应各不相同，但顺序无要求

★ 各语句序列的最后一句应是break；否则连续执行下一case语句，最后一个可省

★ 语句序列不必加{}

★ 多个case可以共用一组语句

★ 不能完全替代多重if语句的嵌套

例：输入一个分数，根据所处的分数段0-59, 60-69, 70-79, 80-89, 90-100分别打印"优"、"良"、"中"、"及格"、"不及格"，其它则打印"输入错误"。

```
#include <iostream>
using namespace std;

int main()
{   int i;
    cout << "请输入成绩（0-100）" << endl;
    cin >> i;
    if (i>=90 && i<=100)
        cout << "优" << endl;
    else if (i>=80 && i<90)
        cout << "良" << endl;
    else if (i>=70 && i<80)
        cout << "中" << endl;
    else if (i>=60 && i<70)
        cout << "及格" << endl;
    else if (i>=0 && i<60)
        cout << "不及格" << endl;
    else
        cout << "输入错误" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

本程序在哪两个数据区间会得到错误结果？

```
#include <iostream>
using namespace std;
int main()
{   int i;
    cout<<"请输入成绩(0-100)"<<endl;
    cin >> i;
    switch(i/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }
    cout<<"程序结束"<<endl;
    return 0;
}
```

例：输入一个分数，根据所处的分数段0-59, 60-69, 70-79, 80-89, 90-100分别打印"优"、"良"、"中"、"及格"、"不及格"，其它则打印"输入错误"。

```
#include <iostream>
using namespace std;

int main()
{   int i;
    cout << "请输入成绩（0-100）" << endl;
    cin >> i;
    if (i>=90 && i<=100)
        cout << "优" << endl;
    else if (i>=80 && i<90)
        cout << "良" << endl;
    else if (i>=70 && i<80)
        cout << "中" << endl;
    else if (i>=60 && i<70)
        cout << "及格" << endl;
    else if (i>=0 && i<60)
        cout << "不及格" << endl;
    else
        cout << "输入错误" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

保证输入区间的正确性

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    cout<<"请输入成绩(0-100)"<<endl;
    cin >> i;
    if (i>=0 && i<=100) {
        switch(i/10) {
            case 10:
            case 9:
                cout<<"优"<<endl;
                break;
            case 8:
                cout<<"良"<<endl;
                break;
            case 7:
                cout<<"中"<<endl;
                break;
            case 6:
                cout<<"及格"<<endl;
                break;
            default:
                cout<<"不及格"<<endl;
                break;
            }
        }
    else
        cout << "输入错误" << endl;

    cout<<"程序结束"<<endl;
    return 0;
}
```

```
case 5:
case 4:
case 3:
case 2:
case 1:
case 0:
```

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.4. 多分支选择结构和switch语句

3.7.4.3. 使用

```
double x = **;
switch(int(x)) {
    ...
    }
```

★ 表达式可以是任何类型，最终取值为整型即可

★ 当整型表达式的取值与整型常量表达式1-n中的任意一个相等时，执行对应的语句序列；否则执行default后的语句序列

（不能是实数，因为无法直接判断相等）

★ 各整型常量表达式的值应各不相同，但顺序无要求

正确

```
switch(i/10) {
   case 10:
   case 9:
      cout<<"优"<<endl;
      break;
   case 8:
      cout<<"良"<<endl;
      break;
   ...
   }
```

正确

```
switch(i/10) {
   case 8:
      cout<<"良"<<endl;
      break;
   case 10:
   case 9:
      cout<<"优"<<endl;
      break;
   ...
   }
```

正确

```
const int k = 1;

switch(i/10) {
   case 3+7:
   case 8+k:
      cout<<"优"<<endl;
      break;
   ...
   }
```

★ 各语句序列的最后一句应是break；否则连续执行下一case语句，最后一个可省（但不建议）

```
switch(i/10) {
    ...
    case 8:
        cout << "良" << endl;
    case 7:
        cout << "中" <<endl;
        break;
    ...
    }
```

Microsoft Visual Studio 调试控制台

```
请输入成绩(0-100)
83
良
中
程序结束
```

Microsoft Visual Studio 调试控制台

```
请输入成绩(0-100)
78
中
程序结束
```

错误

```
const int k = 1;

switch(i/10) {
    case 8:
    case 7+k:
        cout<<"优"<<endl;
        break;
    ...
    }
```

error C2196: case 值“8”已使用

错误

```
int k = 1;
switch(i/10) {
    case 3+7:
    case 8+k:
        cout<<"优"<<endl;
        break;
    ...
    }
```

error C2131: 表达式的计算结果不是常数
message : 因读取超过生命周期的变量而失败
message : 请参见“k”的用法
error C2051: case 表达式不是常量

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.4. 多分支选择结构和switch语句

3.7.4.3. 使用

★ 不能完全替代多重if语句的嵌套

例：输入一个分数，根据所处的分数段0-59, 60-69,70-84,85-100分别打印"优"、"良"、"及格"、"不及格"，其它则打印"输入错误"。

★ 若用 switch(score/10)，则部分分数无法区分

★ 若用 switch(score)，要写 101 个case

★ 若分数精确到小数点后，则无法用switch

```
//if的实现方式
if (score>=0 && score<60)
    cout << "不及格" << endl;
else if (score>=60 && score<70)
    cout << "及格" << endl;
else if (score>=70 && score <85)
    cout << "良" << endl;
else if (score>=85 && score<=100)
    cout << "优" << endl;
else
    cout << "输入错误" << endl;
```

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.5. 编写选择结构的程序

例：键盘输入一个整数当年份，判断该年是否为闰年

```
#include <iostream>
using namespace std;

int main()
{
    int year;
    bool leap;
    cin >> year;
    if (year%4==0) {
        if (year%100==0) {
            if (year%400==0)
                leap=true;
            else
                leap=false;
            }
        else
            leap=true;
        }
    else
        leap=false;

    if (leap)
        cout << year << " is a leap year" << endl;
    else
        cout << year << " is not a leap year" << endl;
    return 0;
}
```

被4、100、400整除

被4、100整除，不被400整除

被4整除，不被100整除

不被4整除

```
#include <iostream>
using namespace std;

int main()
{
    int year;
    bool leap;
    cin >> year;

    if (year%4==0)
        if (year%100==0)
            if (year%400==0)
                leap=true;
            else
                leap=false;
        else
            leap=true;
    else
        leap=false;


    if (leap)
        cout << year << " is a leap year" << endl;
    else
        cout << year << " is not a leap year" << endl;
    return 0;
}
```

可以省略两组括号

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.5. 编写选择结构的程序

例：键盘输入一个整数当年份，判断该年是否为闰年

```
#include <iostream>
using namespace std;

int main()
{
    int year;
    bool leap;
    cin >> year;

    if (year%4!=0)
       leap=false;
    else if(year%100!=0)
       leap=true;
    else if (year%400!=0)
       leap=false;
    else
       leap=true;

    if (leap)
        cout << year << " is a leap year" << endl;
    else
        cout << year << " is not a leap year" << endl;
    return 0;
}
```

改写

不被4整除

被4整除，不被100整除

被4、100整除，不被400整除

被4、100、400整除

```
#include <iostream>
using namespace std;

int main()
{
    int year;
    bool leap;
    cin >> year;

    if ((year%4==0 && year%100!=0)||(year%400==0))
        leap=true;
    else
        leap=false;

    leap =  year%4==0 && year%100!=0 || year%400==0;

    if (leap)
        cout << year << " is a leap year" << endl;
    else
        cout << year << " is not a leap year" << endl;
    return 0;
}
```

改写1

改写2

可简化为以下3种形式：

```
cout << year << (leap ? "is" : "is not") << " a leap year."<<endl;

printf("%d %s a leap year.\n", year, leap ? "is" : "is not");

printf("%d is%s a leap year.\n", year, leap ? "" : " not");
```

§3.结构化程序设计基础

3.7.选择结构和if语句

3.7.5.编写选择结构的程序

例：键盘输入一个整数当年份，判断该年是否为闰年

```
//最简版本
#include <iostream>
using namespace std;

int main()
{
    int year, leap;
    cin >> year;

    leap =  year%4==0 && year%100!=0 || year%400==0;

    cout << year << (leap ? "is" : "is not") << " a leap year."<<endl;

    return 0;
}
```

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.5. 编写选择结构的程序

例：四个椭圆塔，圆心分别为(5,5),(-5,5),(-5,-5)，(5,-5)，长半径为3，短半径为2，这4个塔的高度为10m，塔以外无建筑物(高度为0)，编写程序，输入任一点的坐标，求该点的建筑高度

$$\frac{(x-5)^2}{3^2}+\frac{(y-5)^2}{2^2}=1$$

(x-5)*(x-5)/9+(y-5)*(y-5)/4 <=1

```
#include <iostream>
using namespace std;
int main()
{
    double x, y;            //尽量别用int
    cout << "请输入坐标："; //不加endl，则光标停在本行等待输入
    cin >> x >> y;

    if ( ((x-5)*(x-5)/9+(y-5)*(y-5)/4<=1) ||
         ((x-5)*(x-5)/9+(y+5)*(y+5)/4<=1) ||
         ((x+5)*(x+5)/9+(y+5)*(y+5)/4<=1) ||
         ((x+5)*(x+5)/9+(y-5)*(y-5)/4<=1) )
        cout << "高度为10" << endl;
    else
        cout << "高度为0" << endl;

    return 0;
}
```

注意:
复制粘贴时要改对

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.5. 编写选择结构的程序

例：对n个人分班，每班k(k>0)人，最后不足k人也编为一个班，问要分几个班？键盘输入n，k的值，输出分班数(尝试使用if语句)

```
#include<iostream>
using namespace std;
int main()
{
double o,p,m,b;
int c;
cin>>o>>p;
b=o/p;
c=o/p;
if(b-c>0)m=c+1;
else m=c;
cout<<m;
return 0;
}
```

作者原意：

虽然输入类型为double，但希望键盘输入为整数(o=n p=k)

b是浮点除法
c是整数除法
b-c>0 表示除不尽 => o不是p的整数倍

程序存在的问题：
1、垃圾格式
2、o、p变量用浮点数，不能保证输入的正确性
3、浮点数有误差，导致b-c>0不可信
4、m应该是int型

§3. 结构化程序设计基础

3.7. 选择结构和if语句

3.7.5. 编写选择结构的程序

例：对n个人分班，每班k(k>0)人，最后不足k人也编为一个班，问要分几个班？键盘输入n，k的值，输出分班数(尝试使用if语句)

```
#include <iostream>
using namespace std;
int main()
{
    int n, k;
    cin >> n >> k;

    if (n%k==0)
        cout << n/k << endl;
    else
        cout << n/k+1 << endl;

    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int n, k;
    cin >> n >> k;

    cout << (n%k == 0 ? n/k : n/k+1)  << endl;
    cout << (n/k + (n%k == 0 ? 0 : 1))  << endl;两种方法

    return 0;
}
```

限制：
1、不允许用if-else，如何实现？

```
#include <iostream>
using namespace std;
int main()
{
    int n, k;
    cin >> n >> k;

    cout << n/k + (n%k > 0)  << endl;
    cout << n/k + bool(n%k)  << endl;
    cout << n/k + !!(n%k)    << endl;

    return 0;
}
```

三种方法

限制：
1、不允许用if-else
2、不允许用条件表达式，如何实现？

```
#include <iostream>
using namespace std;
int main()
{
    int n, k;
    cin >> n >> k;

    cout << (n+k-1)/k << endl;

    return 0;
}
```

限制：
1、不允许用if-else
2、不允许用条件表达式
3、不允许用bool，不允许用关系、逻辑运算符如何实现？

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

★ 当型：先判断，后执行（可能一次都不执行）

★ 直到型：先执行，后判断（至少执行一次）

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.1. GOTO语句

形式：

goto 语句标号；

语句标号的组成：

语句标号：

（命名规则同变量：以字母或下划线开始，由字母、数字、下划线组成）

用途：

无条件跳转到语句标号处

```
#include <iostream>
using namespace std;
int main()
{
LOOP:
    cout << "Hello";
    goto LOOP;

    return 0;
}
//程序执行会陷入死循环
```

缺点：

使程序的流程无规律，可读性差(建议少用或不用)

强调：

★ 学习goto的目的，是了解，读别人程序时能看懂

★ 本课程禁止在任何时候、任何方式下使用goto语句

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.1. GOTO语句

与if语句一起构成循环：

例：求1+2+...+100的和

1、当型

```
int main()
{   int i, sum;
    i=1;
    sum=0;
LABEL1:
    if (i<=100) {
        sum=sum+i;
        i++;
        goto LABEL1;
        }
    cout<<"sum="<<sum<<endl;
    return 0;
}
```

2、直到型

```
int main()
{
    int i=1; sum=0;
LABEL1:
    sum+=i;
    i++;
    if (i<=100)
        goto LABEL1;
    cout<<"sum="<<sum<<endl;

    return 0;
}
```

问题：若调整为
i=i+1;
sum=sum+i;
程序如何变动才正确？

§3.结构化程序设计基础

3.8.循环结构和循环语句

3.8.2.用while语句构成循环

3.8.2.1.形式

```
while（表达式）{
        语句序列;
        }
```

3.8.2.2.使用

★ 先判断，后执行

★ 表达式可以是任意类型，按逻辑值求解（非0为真0为假），为真时反复执行

★ 语句序列中只有一个语句时，{}可省

★ 语句序列中应有改变表达式取值的语句,否则死循环

例：求1+2+...+100的和

```
#include <iostream>
using namespace std;

int main()
{
    int i=1, sum=0;

    while(i<=100) {
        sum=sum+i;
        i++;
        }

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

    return 0;
}
```

```
while(i<=100)
    sum+=i++;
```

例：打印1-1000内7的倍数

```
#include <iostream>
using namespace std;

int main()
{
    int i=1;

    while(i<=1000) {
        if (i%7==0)
            cout << i << ' ';
        i++;
        }

    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    int i=7;

    while(i<=1000) {
        cout << i << ' ';
        i+=7;
        }

    return 0;
}
```

§3.结构化程序设计基础

3.8.循环结构和循环语句

3.8.3.用do-while语句构成循环

3.8.3.1.形式

```
do {
        语句序列;
} while(表达式);
```

3.8.3.2.使用

★ 先执行，后判断

★ 表达式可以是任意类型，按逻辑值求解非0为真0为假），为真时反复执行

★ 语句序列中只有一个语句时，{}可省

★ 语句序列中应有改变表达式取值的语句,否则死循环

例：求1+2+...+100的和

```
#include <iostream>
using namespace std;

int main()
{
    int i=1, sum=0;

    do {
        sum=sum+i;
        i++;
    } while(i<=100);

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

    return 0;
}
```

例：打印1-1000内7的倍数

```
#include <iostream>
using namespace std;

int main()
{
    int i=1;

    do {
       if (i%7==0)
           cout << i << ' ';
       i++;
    } while(i<=1000);

    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    int i=7;

    while(i<=1000) {
        cout << i << ' ';
        i+=7;
        }

    return 0;
}
```

更简洁高效的做法，
只要数学上等价即可

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.1. 形式

```
for(表达式1; 表达式2; 表达式3) {
    语句序列;
    }
```

3.8.4.2. for语句的执行过程

① 求解表达式1(初值)

② 以逻辑值求解表达式2，为真则执行循环体，
(当型)　　为假则结束循环体的执行

③ 执行完循环体后，求解表达式3，重复②
(语句序列或表达式3中应有改变表达式2的求解条件)

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.1. 形式

3.8.4.2. for语句的执行过程

3.8.4.3. 与while语句的互换性

```
for(表达式1;表达式2;表达式3){
    语句序列;
    }
```

```
表达式1;
while(表达式2){
    语句序列;
    表达式3;
}
```

例：求1+2+...+100的和

```
#include <iostream>
using namespace std;
int main()
{
   int i=1, sum=0;

   while(i<=100) {
      sum=sum+i;
      i++;
      }

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

   return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
   int i, sum=0;

   for(i=1;i<=100;i++)
      sum=sum+i;



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

   return 0;
}
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.4. for语句的基本使用形式

```
for(循环变量赋初值；循环条件；循环变量增值) {
        语句序列；
        }
```

★ 对已知循环结束条件(或循环次数)的循环表达方式较直观

例：i从1开始累加,加到10000为止,打印出i及累加的和

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int i,sum;
    for(i=1,sum=0;sum<=10000;i++)
        sum=sum+i;
    --i;
    cout << "i=" << i << " sum=" << sum << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
i=141 sum=10011
```

为什么循环结束后要--i ？

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int i=1,sum=0;
    while(sum<=10000)
        sum+=i++;
    --i;
    cout << "i=" << i << " sum=" << sum << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
i=141 sum=10011
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.5. for语句的扩展使用

★ 表达式1可省，在for语句前给变量赋值

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

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

★ 若表达式2省略，则永真（死循环）

可以在语句序列中设置相应条件以退出

```
for(i=1;;i++) {
    …
    if (i>100)

    …
    }
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.5. for语句的扩展使用

★ 表达式3可省，另外设法改变表达式2的取值

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

```
for(i=1;i<=100;) {
    sum=sum+i;
    i++;
    }
```

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

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

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

★ 省略表达式1、3，完全等同于while语句的形式

```
i=1;
for(;i<=100;) {
    sum=sum+i;
    i++;
    }
```

```
i=1;
while(i<=100) {
    sum=sum+i;
    i++;
    }
```

★ 三个表达式全省，相当于永真

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

```
while(1) {
    ...
    }
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.5. for语句的扩展使用

★ 表达式1、3可以是简单表达式，也可以是多个简单表达式组合形式的逗号表达式，
如果因此而导致循环不需要包含任何语句，则要加一个;表示不需要语句序列(;也可以理解为空语句)

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

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

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

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

```
//分号单独一行，表示为空语句形式，更直观

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

★ 最后加一个分号，表示for循环不需要语句序列

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.5. for语句的扩展使用

★ 表达式1、3可以是简单表达式，也可以是多个简单表达式组合形式的逗号表达式，
如果因此而导致循环不需要包含任何语句，则要加一个;表示不需要语句序列(;也可以理解为空语句)

```
#include <iostream>
using namespace std;

int main()
{
    int i, sum;
    for(i=1,sum=0; i<=100; sum+=i++);
    cout << "sum=" << sum << endl;
    return 0;
}
```

有分号

```
#include <iostream>
using namespace std;

int main()
{
    int i, sum;
    for(i=1,sum=0; i<=100; sum+=i++)
    cout << "sum=" << sum << endl;
    return 0;
} //虽然cout形式上无缩进，仍然是for的子句
```

无分号

```
#include <iostream>
using namespace std;

int main()
{
    int i, sum;
    for(i=1,sum=0; i<=100; sum+=i++);
        cout << "sum=" << sum << endl;
    return 0;
} //虽然cout形式上缩进了，但仍然和for是平级的
```

有分号

```
#include <iostream>
using namespace std;
int main()
{
    int i, sum;
    for(i=1,sum=0; i<=100; sum+=i++)
        ;  //空语句可单独一行，表达更清晰一些
    cout << "sum=" << sum << endl;
    return 0;
}
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.4. 用for语句构成循环

3.8.4.5. for语句的扩展使用

★ 表达式2可以是任何类型，但按逻辑值求解

```
char c;
c=getchar();
while(c!='\n') {
    cout<<c;
    c=getchar();
    }
```

```
char c;
for(;(c=getchar())!='\n';cout<<c)
    ;
```

for的表达式2要分三步理解

虽然简单，但可读性差，建议初学者不把与循环变量无关的内容放入for语句

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.5. 循环的嵌套

多种形式

★ {}的匹配原则

与if/else{}的匹配类似，"{"进栈，遇到"}"出栈匹配

三重循环的例子：

```
for(...) {
    ...
    while(...) {
        ...
        for(...) {
            ...
        }
        ...
    }
    ..
}
```

外for循环的其它语句，与while并列

while循环的其它语句，与内for并列

while(...) {
while(...) {
...
}
}
	do {
	do{
	...
	} while(...);
	} while(...);

for(...) {
for(...) {
...
}
}
	while(...) {
	do(...) {
	...
	} while(...);
	}

for(...) {
while(...) {
...
}
}
	do (...) {
	for(...) {
	...
	}
	} while(...);

§3. 结构化程序设计基础

3.8.5. 循环的嵌套

多种形式

★ {}的匹配原则

与if/else{}的匹配类似，"{"进栈，遇到"}"出栈匹配

★ 外层循环每执行一次，内层循环都要执行一遍

★ 各种分支语句、循环语句之间可相互任意嵌套，只要 { } 的匹配理解没有问题，就是正确的

for(i=1;i<=100;i++)
  for(j=1;j<=100;j++)
  cout << i*j << ' ';
 	
cout 语句执行了<u>10000</u>遍
	for(i=1;i<=100;i++) {
	  cout << i << endl;
	  for(j=1;j<=100;j++) {
	  cout << i*j << endl;
	  for(k=1;k<=100;k++)
	  cout << i*j*k << ' ';
	  cout << j*i << endl;
	  }
	  cout << i << endl;
	  }
	红语句及for j执行了_____遍
	蓝语句及for k执行了_____遍
	粉语句执行了_____遍

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.6. 改变循环控制的语句

3.8.6.1. break语句（前面switch/case中用过）

作用：提前结束循环体的循环

例：i从1开始累加，加到10000为止，打印出i及累加的和

```
int main()
{
  int i,sum;
  for(i=1,sum=0;sum<=10000;i++)
     sum=sum+i;
  i--;
  printf("i=%d sum=%d",i,sum);
  return 0;
}
```

```
int main()
{
   int i=1,sum=0;
   while(sum<=10000)
      sum+=i++;
   i--;
   printf("i=%d sum=%d",i,sum);
   return 0;
}
```

```
for(i=1,sum=0;;i++) {
     sum=sum+i;
     if (sum>10000)
        break;
     }
//不需要i--，为什么？
```

```
while(1) {
    sum=sum+i++;
    if (sum>10000)
       break;
    }
 i--;
```

按程序执行逻辑逐步分析，注意细节!!!

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.6. 改变循环控制的语句

3.8.6.1. break语句（前面switch/case中用过）

作用：提前结束循环体的循环

★ 无条件结束循环体，因此必须和if/else语句一起使用才能体现实际的意义

★ 当多重循环嵌套时，break仅跳出本循环

```
for(……) {
    while(……) {
        ……
        break;
        ……
        }
    ……
    }
```

★ 若出现循环和switch语句的嵌套，则break的位置决定了跳转的位置

```
for(...) {
    switch(...) {
        case ...:
            ...
            break;
        ...
    }
    ...
    }
```

```
switch(...) {
    case ...:
        for(...) {
            ...
            break;
            }
        ...
        break;
    case ...
    }
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.6. 改变循环控制的语句

3.8.6.1. break语句

3.8.6.2. continue语句

作用：结束本次循环，进行下一次是否执行循环的判断

★ 无条件结束本次循环，因此必须和if/else语句一起使用才能体现实际的意义

★ 若出现循环和switch语句的嵌套，则continue只对循环体有效

★ for语句中若出现continue，则先执行表达式3，再去判断表达式2是否应该继续执行

```
switch(...) {
    case ...:
        for(...) {
            ...
            continue;
            }
        ...
        break;
    case ...
    }
```

```
for(...) {
    switch(...) {
        case ...:
            ...
            continue;
        ...
        }
    ...
    }
```

例：打印1-1000内7的倍数

```
#include <iostream>
using namespace std;

int main()
{
    int i=1;

    while(i<=1000) {
        if (i%7==0)
            cout << i << ' ';
        i++;
        }

    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    int i=1;
    while(i<=1000) {
        if (i%7!=0) {
            i++;
            continue;
            }
        cout << i << ' ';
        i++;
        }
    return 0;
}
```

仅为了举例continue的使用，逻辑复杂，不建议

例：打印1-1000内7的倍数

```
#include <iostream>
using namespace std;

int main()
{
    int i=0;
    while(++i<=1000) {
        if (i%7!=0)
            continue;
        cout <<i << ' ';
        }
    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    for(i=1;i<=1000;i++) {
        if (i%7!=0)
            continue;
        cout << i << ' ';
        }
    return 0;
}
```

§3. 结构化程序设计基础

3.8.6. 改变循环控制的语句

3.8.6.3. break和continue的比较

```
while(表达式1) {
    语句序列1;
    break/continue;
    语句序列2;
    }
```

例：给出下列程序的运行结果

```
#include <iostream>
using namespace std;      ?

int main()
{
    int i=0,sum=0;
    while(i<1000) {
        i++;
        break;
        sum=sum+i;
        }
    cout << "i=" << i
         << " sum=" << sum;
         << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;      ?

int main()
{
    int i=0,sum=0;
    while(i<1000) {
        i++;
        continue;
        sum=sum+i;
        }
    cout << "i=" << i
         << " sum=" << sum;
         << endl;
    return 0;
}
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.7. 编写循环结构的程序

例：用公式 $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$ 求 π 值(到最后一项绝对值<10^{-7}为止)

$$\boxed{\frac{1}{1} + \frac{-1}{3} + \frac{1}{5} + \frac{-1}{7}}$$

```
#include <cmath> //所有数学类函数对应的头文件

//核心算法
while(fabs(t) > 1e-7) {
    pi = pi + t;    //pi为累加和
    n  = n + 2;     //n为分母
    s  = -s;        //分子在正负1间变化
    t  = s/n;       //每项的值
    }
```

```
#include <iostream>
#include <iomanip>    //格式输出
#include <cmath>      //fabs
#include <windows.h>  //取系统时间
using namespace std;

int main()
{   int s=1;
    double n=1, t=1, pi=0;
    LARGE_INTEGER tick, begin, end;
    QueryPerformanceFrequency(&tick);     //取计数器频率
    QueryPerformanceCounter(&begin);      //取初始硬件定时器计数
    while(fabs(t) > 1e-6) {
       pi=pi+t;
       n=n+2;
       s=-s;
       t=s/n;
       }
    QueryPerformanceCounter(&end);//获得终止硬件定时器计数
    /* 执行到此，打印pi的值 */
    pi=pi*4;
    cout << "n="  << setprecision(10) << n << endl;
    cout<<"pi="<<setiosflags(ios::fixed)<<setprecision(9)<<pi<< endl;

    cout << "计数器频率：" << tick.QuadPart << "Hz" << endl;
    cout << "时钟计数  ：" << end.QuadPart - begin.QuadPart << endl;
    cout << setprecision(6) << (end.QuadPart - begin.QuadPart)/double(tick.QuadPart) << "秒" <<endl;
    return 0;
}
```

调整两处蓝色箭头处的值，
对比不同精度的执行时间

(1) n，t，pi为double型

精度为1e-6：pi=　　n=　　时间=

1e-7：pi=　　n=　　时间=

1e-8：pi=　　n=　　时间=

1e-9：pi=　　n=　　时间=

(2) n，t，pi为float型

精度为1e-6：pi=　　n=　　时间=

1e-7：pi=　　n=　　时间=

1e-8：为什么无结果？

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.7. 编写循环结构的程序

例：求 Fibonacci 数列的前40项

本程序的不足之处：无法回溯
(当f1表示第7个月后，第5个月的值无法再现)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int i, f1=1, f2=1;    //初值

    for(i=1; i<=20; i++) { //每次2数，20次=40个
        cout << setw(12) << f1 << setw(12) << f2;
        if (i%2==0)
            cout << endl; //每2次(4个)加换行

        f1 = f1 + f2;     //f1为第3/5/7/...个月
        f2 = f1 + f2;     //f2为第4/6/8/...个月
        }

    return 0;
}
```

§3. 结构化程序设计基础

3.8. 循环结构和循环语句

3.8.7. 编写循环结构的程序

例：找出100-200间的全部素数

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int m, k, i, line=0;
    bool prime;

    for(m=101; m<=200; m+=2) {//偶数没必要判断
        prime=true;  //对每个数，先认为是素数
        k=int(sqrt(m)); //k=sqrt(m)则VS有警告
        for(i=2; i<=k; i++)
            if (m%i==0) {
                prime=false;
                break;
                }

        if (prime) {
            cout << setw(5) << m;
            line++;  //计数器，只为了加输出换行
            }
         if (line%10==0)     //每10个数输出一行
             cout<<endl;
    } //end of for
    return 0;
}
```

任意m%i==0就不是素数，
循环不必再继续执行
整个循环完成，m%i==0
都未满足，才认为是素数

```
for(i=2; prime && i<=k; i++)
    if (m%i==0)
        prime=false;

//是否可以改成这种形式？
```

```
if (prime) {
    cout << setw(5) << m;
    n=n+1;
    if (n%10==0)
        cout<<endl;
    }
```

m=103-200，看一下两者区别，
为什么？哪个是正确的？

改写：
不用逻辑变量prime，
直接用i和k的关系来判断
想清楚，为什么!!!

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int m, k, i, line=0;

    for(m=101; m<=200; m+=2) {
        k=int(sqrt(m));

        for(i=2; i<=k; i++)
            if (m%i==0)
                break;

        if (i>k) {
            cout << setw(5) << m;
            line++;
            if (line%10==0)
                cout << endl;
            }
        } //end of for

    return 0;
}
```

i循环的退出有两个可能
1、不满足i<=k(是,且i>k)
2、满足m%i==0(否,且i<=k)

§3.结构化程序设计基础

3.9.输入输出重定向

3.9.1.基本概念

输入重定向：程序执行时，系统默认的输出设备是显示器，如果改为其他设备/文件，则成为输出重定向

输入重定向：程序执行时，系统默认的输入设备是键盘，如果改为其他设备/文件，则成为输入重定向

3.9.2.输出的分类

cout: 标准输出

cerr: 错误输出

clog: 错误输出

3.9.3.标准和错误输出重定向到文件中

```
demo  > a.txt
demo 1> a.txt
demo 2> a.txt
demo 1> a.txt 2> b.txt
demo 1> a.txt 2> &1
```

（将 > 换为 >> 即可实现重定向文件的追加而不清空原有内容）

3.9.4.将输入重定向为来自文件中

```
demo < z.dat
```

3.9.5.同时进行输入/输出重定向

```
demo  < z.dat 1> a.txt
demo 1> a.txt  < z.dat
demo  > a.txt  < z.dat
demo 1> a.txt 2> b.txt < z.dat
demo 1> a.txt 2> &1    < z.dat
```

§3.结构化程序设计基础

3.10.管道运算符

3.10.1.管道运算符的使用

将前一个程序的输出当做后一个程序的输入

可级联使用

3.10.2.用工具get_input_data.exe并配合管道运算符测试程序

测试文件的基本组织结构为：[组名（不建议含特殊字符）]+数据

从测试文件中读取某组的输入数据：get_input_data 测试文件名 组名

从测试文件中读取某组的输入数据给测试程序：get_input_data 测试文件名 组名 | 测试程序名

（可以在后面通过 > 和 >> 方式将执行结果输出重定向到文件中）

§3. 结构化程序设计基础

3.11. 条件编译

3.11.1. 问题的提出

例：

输入成绩，根据分数打印及格或不及格
 	
if (...) {
...
}
else {
...
}
	某个游戏软件，适应不同大小的屏幕
	if (屏幕是1920*1080) {
	...
	}
	else {
	...
	}

输入成绩，根据分数打印分数等级
 	
if (...) {
...
}
else if (...) {
...
}
else {
...
}
 	
可能else if会重复多次
	某个游戏软件，适应不同大小的屏幕
	
	if (屏幕是1920*1080) {
	...
	}
	else if (屏幕是1280*720){
	...
	}
	else {
	...
	}
	
	可能else if会重复多次

左右的区别：

输入成绩，根据分数打印分数等级 对于if-else形式的双分支(包括if-elseif-else形式的多分支)， 每次执行时只能选择其中的某一个分支执行但多次执行时可能每个分支都可能会被执行到， 因此每个分支都是有意义的	某个游戏软件，适应不同大小的屏幕 对于if-else形式的双分支(包括if-elseif-else形式的多分支)， 对于某种型号的设备，多次执行的都是其中的一个固定分支， 因此分支中的其他部分对某种型号设备的可执行文件而言是无意义的

因为多种型号设备的公用代码很多(非显示部分)，因此希望维护一套源程序代码

既希望维护一套源代码，
又希望可执行文件中
仅包含有效部分(节约空间)

§3. 结构化程序设计基础

3.11. 条件编译

3.11.2. 问题的引入

引入 ：某些程序在调试、兼容性、平台移植等情况下希望通过简单地设置参数就能生成不同的软件

方法1：把所有可能用到的代码都写进源程序文件中，再全部编译到可执行文件中，执行时根据相应的条件选择不同的代码执行
（分支语句方式，源程序统一，可执行代码大）

方法2：把所有可能用到的代码都写进源程序文件中，在编译之前根据需要选择待编译的代码段，再进行编译，可执行文件中只包含需要的代码段
（条件编译方式，源程序统一，可执行代码小）

§3. 结构化程序设计基础

3.11. 条件编译

3.11.3. 条件编译的三种形式

#ifdef 标识符 　程序段1 #else 　程序段2 #endif 若定义了标识符 则编译程序段1 否则编译程序段2	#ifndef 标识符 　程序段1 #else 　程序段2 #endif 若未定义标识符 则编译程序段1 否则编译程序段2	#if 表达式 　程序段1 #else 　程序段2 #endif 若表达式为真 则编译程序段1 否则编译程序段2

★ 该过程在编译阶段完成，最终形成的可执行文件中只包含两个程序段中的某一个

- #xxx 是C/C++约定的编译预处理指令(例：#define/#include/...)

★ 预编译指令中的表达式与C语言本身的表达式基本一致，逻辑运算、算术运算等均可用于预编译指令

★ 修改预编译条件后，每次都要重新编译链接才能生成新的可执行文件

- 修改条件编译的方法一般是手动修改，再次编译
- 更进一步的方法：1、将条件编译开关放在设置或makefile文件中，通过修改makefile来编译
  2、利用标识不同编译器的预置宏定义来区分不同编译器

★ 形式如if-elseif的多分支条件编译请自行学习

★ 不可能通过执行程序时输入某值或根据程序执行时是否满足某条件的方式来选择，因为是"编译预处理"

§3.结构化程序设计基础

3.11.条件编译

3.11.4.预处理器编译指令

#define　定义一个预处理宏

#undef　取消宏的定义

#if　编译预处理中的条件命令

#ifdef　判断某个宏是否被定义，若已定义，执行随后的语句

#ifndef　与#ifdef相反，判断某个宏是否未被定义

#elif　若#if，#ifdef，#ifndef或前面的#elif条件不满足，则执行#elif之后的语句

#else　与#if，#ifdef，#ifndef对应，若这些条件不满足，则执行#else之后的语句

#endif　#if，#ifdef，#ifndef这些条件命令的结束标志

defined　与#if，#elif配合使用，判断某个宏是否被定义

程序如下，假设两次的输入为7，-3，写出程序的运行结果

```
#define PROGRAM
main()
{  int t;
   cin >> t;
#ifdef PROGRAM
   if (t>=0)
      cout << "t是非负整数" << endl;
#else
   if (t<0)
      cout << "t是负整数" << endl;
#endif
   cout << "End." << endl;
   return 0;
}
```

注：#ifdef 只判断是否定义，不判断定义值的T/F

未被编译进可执行文件中

输入为7时：
t是非负整数
End

输入为-3时：
End

程序如下，假设两次的输入为7，-3，写出程序的运行结果

```
#define PROGRAM
main()
{  int t;
   cin >> t;
#ifndef PROGRAM
   if (t>=0)
      cout << "t是非负整数" << endl;
#else
   if (t<0)
      cout << "t是负整数" << endl;
#endif
   cout << "End." << endl;
   return 0;
}
```

注：#ifndef 只判断是否定义，不判断定义值的T/F

未被编译进可执行文件中

输入为7时：
End

输入为-3时：
t是负整数
End

程序如下，假设两次的输入为7，-3，写出程序的运行结果

```
#define PROGRAM 0
main()
{  int t;
   cin >> t;
#if PROGRAM
   if (t>=0)
      cout << "t是非负整数" << endl;
#else
   if (t<0)
      cout << "t是负整数" << endl;
#endif
   cout << "End." << endl;
   return 0;
}
```

注：#if 要判断定义值的T/F，未定义则按F处理

未被编译进可执行文件中

输入为7时：
End

输入为-3时：
t是负整数
End

程序如下，假设两次的输入为7，-3，写出程序的运行结果

```
#define PROGRAM 1
main()
{  int t;
   cin >> t;
#if PROGRAM
   if (t>=0)
      cout << "t是非负整数" << endl;
#else
   if (t<0)
      cout << "t是负整数" << endl;
#endif
   cout << "End." << endl;
   return 0;
}
```

注：#if 要判断定义值的T/F，未定义则按F处理

未被编译进可执行文件中

输入为7时：
t是非负整数
End

输入为-3时：
End

§3. 结构化程序设计基础

3.11. 条件编译

最初的问题：如何让下面程序在多编译器中均能通过

提示：去找各编译器的预置标识

```
int main()
{
    char a[10];
    #if ***   //如果是Dev
        gets(a);
    #elif *** //如果是VS
        gets_s(a);
    #elif *** //如果是Linux
        fgets(a, 10, stdin);
    #endif
    cout << a << endl;
    return 0;
}
```

```
#if ***   //如果是VS
    gets_s(a);
#elif *** //如果是Linux
    fgets(a, 10, stdin);
#elif *** //如果是Dev
    gets_s(a);
#endif
```

```
#if (__GNUC__)     //Dev(GNU GCC)
    gets(...);
#elif (_MSC_VER)   //Microsoft Visual C/C++
    gets_s(...);
#endif
```

§4. 函数

4.1. 概述

★ C/C++程序的基本组成单位

★ 一个函数实现一个特定的功能

★ 有且仅有一个main函数，程序执行从main开始

★ 函数平行定义，嵌套调用

★ 一个源程序文件由多个函数组成，一个程序可由多个源程序文件组成

```
//a1.cpp          //a2.cpp          //a3.cpp
int fun1( )       float fun5( )     double fun4( )
{                 {                 {
}                 }                 }
short fun2( )                       int main()
{                                   {
}                                   }
long fun6( )
{
}
```

一个程序由3个源程序文件组成
共6个函数，有且仅有一个main

★ 函数的分类

用户使用角度	标准函数（库函数）由系统提供（fabs/sqrt/strlen）使用时需要包含相应头文件 自定义函数 用户自己编写 ● 在使用上无任何的区别
函数形式	无参 调用函数无数据传递给被调用函数（getchar()） 有参 调用函数有数据传递给被调用函数（putchar('A')）

§4. 函数

4.2. 函数的定义

4.2.1. 无参函数的定义

函数返回类型 函数名（[void]）{ () / (void) }

{

函数体 { 声明语句 / 执行语句 }

}

```
int fun()
{
    cout << "***" << endl;
    return 0;
}
int fun(void)
{
    cout << "***" << endl;
    return 0;
}
```

★ 函数名的命名规则同变量

★ 返回类型与数据类型相同

```
int fun(void)
{
    ...
}
```

```
long fun2()
{
    ...
}
```

★ 返回类型可以是void，表示不需要返回类型

```
void fun3()
{
}
```

★ C缺省返回类型为int(不建议缺省，int也写)，C++不支持默认int，必须写

```
fun3(…)
{
}//C++编译报错
```

```
已启动生成…
1>------ 已启动生成: 项目: c-demo, 配置: Debug Win32 ------
1>c-demo.c
1>c-demo.vcxproj -> D:\WorkSpace\VS2019-Demo\Debug\c-demo.exe
========== 生成: 成功 1 个，失败 0 个，最新 0 个，跳过 0 个 ==========
```

c-demo.c

```
#include <stdio.h>

fun()
{
    return 0;
}

int main()
{
    fun();
    return 0;
}
```

cpp-demo.cpp

```
#include <iostream>
using namespace std;

fun()
{
    return 0;
}

int main()
{
    fun();
    return 0;
}
```

error C4430: 缺少类型说明符 - 假定为 int。注意: C++ 不支持默认 int

§4. 函数

4.2. 函数的定义

4.2.1. 无参函数的定义

★ ANSI C++要求main函数的返回值只能是int并且不能缺省不写，否则编译会报错；但部分编译器可缺省不写；VS系列还允许void等其它类型(建议唯一int)

```
#include <iostream>
using namespace std;

main()
{
    return 0;
}
//VS报error
//Dev正确
```

```
error C4430: 缺少类型说明符 - 假定为 int。注意: C++ 不支持默认 int
```

```
编译日志  调试
编译结果...
--------
- 错误: 0
- 警告: 0
```

```
#include <iostream>
using namespace std;

void main()
{
    return;
}
//VS报warning
//Dev报error
```

```
warning C4326: "main" 的返回类型应为 "int" 而非 "void"
```

```
信息
[Error] '::main' must return 'int'
In function 'int main()':
[Error] return-statement with no value, in function returning 'int' [-fpermissive]
```

```
#include <iostream>
using namespace std;

long main()
{
    return 0L;
}
//VS报warning
//Dev报error
```

```
warning C4326: "main" 的返回类型应为 "int" 而非 "long"
```

```
信息
[Error] '::main' must return 'int'
```

§4. 函数

4.2. 函数的定义

4.2.2. 有参函数的定义

函数返回类型 函数名（形式参数表）

{

函数体 { 声明语句 / 执行语句 }

}

```
int max(int x, int y)
{
    int z;          /* 声明语句 */
    if (x>y)
        z=x;
    else
        z=y;
    return z;
}
```

```
return x>y?x:y;
```

★ 函数名的命名规则同变量

★ 返回类型与数据类型相同

★ 返回类型可以是void，表示不需要返回类型

★ C缺省返回类型为int(不建议缺省，int也写)，C++不支持默认int，必须写

§4. 函数

4.3. 函数的嵌套调用

4.3.1. C++程序的执行过程(一个具体的例子)

例：程序如下

```
void b()
{
    ...
}
void a()
{   ...
    b();
    ...
}
int main()
{   ...
    a();
    ...
    return 0;
}
```

//左例，9步

(1) 执行main函数的开头部分

(2) 遇到调用a函数的语句，流程转去a函数

(3) 执行a函数的开头部分

(4) 遇到调用b函数的语句，流程转去b函数

(5) 执行b函数，如果再无其他嵌套的调用，则完成b函数的全部操作

(6) 返回原来调用b函数的位置，即返回a函数

(7) 继续执行a函数中尚未执行的部分，直到a函数结束

(8) 返回main中调用a函数的位置

(9) 继续执行main函数的剩余部分直到结束

如何返回？

§4. 函数

4.3. 函数的嵌套调用

4.3.2. C++程序的执行过程(通用描述)

(1) 从main函数的第一个执行语句开始依次执行

(2) 若执行到函数调用语句，则保存调用函数当前的一些系统信息(保存现场 – 进栈)

(3) 转到被调用函数的第一个执行语句开始依次执行

(4) 被调用函数执行完成后，返回到调用函数的调用处，恢复调用前保存的系统信息(恢复现场 – 出栈)

(5) 若被调用函数中仍有调用其它函数的语句，则嵌套执行步骤(2)-(4)(保存和恢复现场的操作遵循栈的操作规则)

(6) 所有被调用函数执行完后，顺序执行main函数的后续部分直到结束

4.3.3. C++程序的执行过程(栈方式理解)

自行画出调用过程中栈的变化形式

图示：main-f1表示保存main的现场，转去f1函数执行

4.3.4. 特点

★ 嵌套的层次、位置不限

★ 遵循后进先出的原则(栈)

★ 调用函数时，被调用函数与其所调用的函数的关系是透明的，适用于大程序的分工组织

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

形式参数：在被调用函数中出现的参数

实际参数：在调用函数中出现的参数

★ 实参与形参分别占用不同的内存空间，实形参名称既可以相同，也可以不同

★ 参数的传递方式是"单向传值"，即将实参的值复制一份到形参中(理解为 形参=实参 的形式)

```
int main()
{
    int i=15, j=10, m;
    m = max(i, j);
    cout<< "max=" <<m;
    return 0;
}
```

i, j为实参

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

x, y为形参

```
int main()
{
    int x=15, y=10, m;
    m = max(x, y);
    cout<< "max=" <<m;
    return 0;
}
```

x, y为实参

允许同名

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

x, y为形参

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

★ 实参与形参分别占用不同的内存空间

★ 参数的传递方式是"单向传值"，即将实参的值复制一份到形参中(理解为 形参=实参 的形式)

=> 推论：执行后，形参的变化不影响实参值

```
#include <iostream>
using namespace std;
void fun(int x)
{   cout << "x1=" << x << endl;
    x=5;
    cout << "x2=" << x << endl;
}
int main( )
{   int k=15;
    cout << "k1=" << k << endl;
    fun(k);
    cout << "k2=" << k << endl;
    return 0;
}
```

k	15
x	~~15~~ 5

```
k1=15
x1=15
x2=5
k2=15
```

★ 实参可以是常量、变量、表达式，形参只能是变量

```
int main()
{   int k=10;
    fun(2+k*3);
    return 0;
}
```

```
void fun(int x)
{
    ...
}
```

x = 2+k*3

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

★ 形参在使用时分配空间，函数运行结束后释放空间

```
int main()      void f1(int x)     void f2(int y)
{               {                  {
    f1(10);         ...                ...
    f2(15);     }   常量  10       }
    ...                 |
}                       v
                    x   10
```

此时f2还未执行到，尚未为y分配空间

```
int main()      void f1(int x)     void f2(int y)
{               {                  {
    f1(10);         ...                ...
    f2(15);     }   常量  15       }
    ...                 |
}                       v
                    x y 15
```

此时f1已执行完成，分配给x的空间已回收为y分配的空间可能占原x的位置

x和y可能共用4个字节的空间

```
int main()      void f1(int x)
{  ...          { ...
   f1(..);      }
   ...
   f1(..);
   ...
   f1(..);
   ...
}
```

1、假设main中调用10000次f1()，则x的分配释放会重复10000次

2、每次x分配的4字节不保证是同一个空间

具体理解等到第6章指针后

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

★ 实参、形参类型必须一致，否则结果可能不正确

```
#include <iostream>
using namespace std;
int fun(short x)
{
    cout << "x=" << x << endl;   x=4464
    return 0;
}
int main()
{
    long k=70000;
    fun(k); //编译有警告
    cout << "k=" << k << endl;   k=70000
    return 0;
}
```

实形参类型不一致时，
转换规则同赋值（形参 = 实参）

warning C4244: “参数”: 从“long”转换到“short”，可能丢失数据

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 通过return语句获得，若return类型与返回类型定义不一致，以返回类型为准进行数据转换

```
... f(...)
{
    int k;
    ...
    k = fun(...);
    ...
}
```

```
int fun(...)
{
    int s;
    ...
    ...
    return s;
}
```

若s=10
则k=10

理解为
调用函数中值=return后值的形式

```
long fun2()
{
  long a;
  …
  return a;
}
```
正确

```
long fun2()
{
    short a;
    …
    return a;
}
```
正确

```
short fun3()
{
    long a;
    …
    return a;
}
```
可能不正确

```
//问1：运行结果（d的值是多少？）
//问2：哪句会有warning错？
#include <iostream>
using namespace std;

short fun3()
{
    long a = 70000;
    return a;
}
int main()
{
    long d;
    d = fun3();
    cout << d << endl;

    return 0;
}
```

warning C4244: “参数”: 从“long”转换到“short”，可能丢失数据

0000000000000001 0001000101110000

0001000101110000

0000000000000000 0001000101110000

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ return后可以是变量、常量、表达式，有两种形式(带括号、不带括号)

```
return a;          return k*2;
return (a);        return (k*2);
```

★ 若函数不要求有返回值，则指定返回类型为void

```
void fun1()      int main()     int fun()
{                {              {
   ...              ...            ...
   return;          return 0;      return 0;
}                }              }
```

无return语句
空return语句

return int 型

return int型

=> 推论：① 返回类型为void的函数不能出现在除逗号表达式外的任何表达式中

② 若逗号表达式要参与其它运算，则不能做为最后一个表达式出现

返回类型非void的函数，如果不带return语句，不同编译器表现不同(error/warning/不报错)
VS：main无return不报错，其余函数报error

```
error C2186: "+": "void"类型的操作数非法
error C2679: 二元"<<": 没有找到接受"void"类型的右操作数的运算符(或没有可接受的转换)
```

```
#include <iostream>
using namespace std;
void f()
{   int x=10;
}
int main()
{
    int k=10;
    k=k+f();   //编译错
    k,f();     //可编译通过，无意义
    cout << (k, f()) << endl;       //编译错
    cout << (k, f(), k+2) << endl; //可编译通过
    return 0;
}
```

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 一个return只能带回一个返回值

★ 函数中可以有多个return语句，但只能根据条件执行其中的一个，执行return后，函数调用结束(return后的语句不会被执行到)

```
int fun(void)
{
    if (...)
        return ...;
    else
        return ...;
    ....;  //无法被执行到
}
```

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 如果函数中有分支/循环语句，但return未覆盖全部分支/出口，则VS会报warning错(无论判断条件是否覆盖!)

```
#include <iostream>
using namespace std;
int fun(int x)
{
    if (x > 1) {
        if (x > 10)
            return 1;
    }
    else
        return 2;
}
int main()
{
    int x, r;
    cin >> x;
    r = fun(x);
    return 0;
}
```

warning C4715: “fun”: 不是所有的控件路径都返回值

人的思维 ：已全覆盖
编译器思维：缺else

```
#include <iostream>
using namespace std;
int fun(int x)
{
    if (x > 1)
        return 1;
    else if (x <= 1)
        return 2;
} //已覆盖int型的全部表示范围
int main()
{
    int x, r;
    cin >> x;
    r = fun(x);
    return 0;
}
```

§4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 如果函数中有分支/循环语句，但return未覆盖全部分支/出口，则VS会报warning错(无论判断条件是否覆盖!)

```
1  #include <iostream>
2  using namespace std;
3  int fun()
4  {
5      int i;
6      for (i = 0; i <= 100; i++) {
7          if (i >= 10)
8              return 0;
9      }
10 }
11 int main()
12 {
13     fun();
14     return 0;
15 }
```

warning C4715: “fun”: 不是所有的控件路径都返回值

同理：
人的思维：
i<=100不可能被执行到，因此只有i>=10这一个出口

编译器思维：
循环退出有两个出口
(1) i>=10 满足后 return 0
(2) 表达式2（i<=100）不满足，结束循环(但无return)

```
1  #include <iostream>
2  using namespace std;
3  int fun()
4  {
5      int i;
6      for (i = 0; ; i++) {
7          if (i >= 10)
8              return 0;
9      }
10 }
11 int main()
12 {
13     fun();
14     return 0;
15 }
```

200 % 未找到相关问题

输出

显示输出来源(S): 生成

已启动生成…
1>------ 已启动生成: 项目: cpp-demo, 配置: Debug Win32 ------
1>cpp-demo.cpp
1>cpp-demo.vcxproj -> D:\WorkSpace\VS2019-Demo\Debug\cpp-demo.exe
========== 生成: 成功 1 个，失败 0 个，最新 0 个，跳过 0 个 ==========

§4. 函数

4.5. 函数的调用

函数的编写方法：

通过第2-3章的基本知识，定义不同数据类型的变量，采用顺序、分支、循环等基本结构，按照函数的预期功能来编写每个函数

§4. 函数

4.5. 函数的调用

4.5.1. 基本形式

函数名()　　：适用于无参函数

函数名(实参表列)：适用于有参函数，用，分开

与形参表的个数、顺序、类型一致

★ 若同一变量同时出现在一个函数的多个参数中，且有自增、赋值、复合赋值等改变变量值的操作，则不同编译器处理的方式可能不同（不在讨论，也不建议深入）

```
int i=3;
fun(i++, i)
```

从左至右：fun(3,4)

从右至左：fun(3,3)

不再讨论
也不建议深入

注意：fun(i++, --j) 这种不同变量是必须讨论的

```
printf/scanf等函数有参数个数、类型不等的
情况出现，称为可变参数方式，本课程暂不讨论

printf("%d\n", a);          //2个参数
printf("%d %d\n", a, b);    //3个参数
scanf("%d", &a);            //2个参数
scanf("%d %d", &a, &b);     //3个参数
```

```cpp
#include <iostream>
using namespace std;
void fun(int x, int y)
{
    cout << x << ' ' << y << endl;
}
int main()
{
    int i = 3;
    fun(i++, i);
    return 0;
}
```

D:\WorkSpace\VS2019-Demo\cpp-

```
3 4
```

Dev
VS

Microsoft Visual Studio 调试控制台

```
3 4
```

```
[root@RF5-X64 ~]# cat t2.cpp
#include <iostream>
using namespace std;
void fun(int x, int y)
{
        cout << x << ' ' << y << endl;
}
int main()
{
        int i = 3;
        fun(i++, i);
        return 0;
}
[root@RF5-X64 ~]#
[root@RF5-X64 ~]# c++ -Wall -o t2 t2.cpp
[root@RF5-X64 ~]#
[root@RF5-X64 ~]# ./t2
3 3
```

某Linux版本
的编译器

§4. 函数

4.5. 函数的调用

4.5.2. 调用方式

函数语句：函数调用+；

```
printf("Hello.\n");
putchar('A');
```

函数表达式：出现在某个表达式中

```
c=my_max(a,b)+4;
k=sqrt(m);
```

函数参数：作为另一个函数的参数

```
printf("max=%d", my_max(a,b));
putchar( getchar() );
sqrt( fabs(x) );
```

函数返回类型不能是void

§4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

定义及实现时：	调用时：
long f1() { ... } int max(int x, int y) { ... }	k = f1(); ✓ k = long f1(); ✗ k = max(i, j); ✓ k = int max(i,j); ✗

```
1   #include <iostream>
2   using namespace std;
3   long f1()
4   {
5       cout << "f1" << endl;
6       return 0L;
7   }
8   int main()
9   {
10      int k;
11      k = long f1(); //调用f1函数
12      return 0;
13  }
```

error C2062: 意外的类型“long”

§4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

★ 无参函数调用时，参数位置不能写void

定义及实现时：	调用时：

int fun()　　//空
{　...
}
int fun(void) //写void
{　...
}
	k = fun();　✓
	k = fun(void);　✗

```
1   #include <iostream>
2   using namespace std;
3   int fun(void)
4   {
5       cout << "fun" << endl;
6       return 0;
7   }
8   int main()
9   {
10      int k;
11      k = fun(void); //调用f1函数
12      return 0;
13  }
```

```
error C2144: 语法错误: "void" 的前面应有 ")"
error C2144: 语法错误: "void" 的前面应有 ":"
error C2059: 语法错误: ")"
warning C4091: "": 没有声明变量时忽略 "void" 的左侧
```

§4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

★ 无参函数调用时，参数位置不能写void

★ 有参函数调用时，实参不能写类型

定义及实现时：	调用时：
int max(int x, int y) { ... }	int i=10, j=15; k=max(i, j); ✓ k=max(int i, int j); ✗

```
1   #include <iostream>
2   using namespace std;
3   int my_max(int x, int y)
4   {
5       return x > y ? x : y;
6   }
7   int main()
8   {
9       int i = 10, j = 15, k;
10      k = my_max(int i, int j);
11      cout << "max=" << k << endl;
12      return 0;
13  }
```

```
(10,13): error C2144: 语法错误:“int”的前面应有“)”
(10,17): error C2660: “my_max”: 函数不接受 0 个参数
(3,5): message : 参见“my_max”的声明
(10,13): error C2144: 语法错误:“int”的前面应有“;”
(10,17): error C2086: “int i”: 重定义
(9): message : 参见“i”的声明
(10,20): error C2062: 意外的类型“int”
(10,25): error C2059: 语法错误:“)”
```

§4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

定义及实现时：	调用时：
long f1() { ... } int max(int x, int y) { ... }	k = f1(); ✓ k = long f1(); ✗ k = max(i, j); ✓ k = int max(i, j); ✗

★ 无参函数调用时，参数位置不能写void

★ 有参函数调用时，实参不能写类型

定义及实现时：	调用时：
int max(int x, int y) { ... }	int i=10, j=15; k=max(i, j); ✓ k=max(int i, int j); ✗

问题：其它函数的返回值
可由调用函数使用，
main的返回值给谁？

定义及实现时：	调用时：

int fun() //空
{ ...
}
int fun(void) //写void
{ ...
}
	k = fun(); ✓
	k = fun(void); ✗

§4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对库函数，加相应的头文件说明

```
#include <stdio.h>   输入输出函数
#include <math.h>    数学运算函数
#include <string.h>  字符串运算函数

#include <cstdio>   输入输出函数
#include <cmath>    数学运算函数
#include <cstring>  字符串运算函数
```

C方式
仅此种

C++方式
两种均可

注意：<cstdio>和<cmath>这两个头文件在VS中缺省可以不加，其它编译器一般需要加

§4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前

两种方法：

返回类型 函数名(形参类型);

返回类型 函数名(形参类型 形参表);

```
int my_max(int, int);
int main()
{
    k=my_max(i,j);
}
int my_max(int x, int y)
{
    ...
}
```

```
int my_max(int x, int y);
int main()
{
    k=my_max(i,j);
}
int my_max(int x, int y)
{
    ...
}
```

```
int my_max(int p, int q);

int main()
{
    k=my_max(i,j);
}
int my_max(int x, int y)
{
    ...
}
```

pq不要求与实现中的xy一致

§4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对库函数，加相应的头文件说明

★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前

★ 若被调用函数出现在调用函数之前，可以不加说明(有些编译器可能必须加)

```
//可以没有说明
float fun()
{   ...
}
int main()
{
    float k;
    k=fun();
    return 0;
}
```

```
float fun(); //必须有说明
int main()
{
    float k;
    k=fun();
    return 0;
}
float fun()
{   ...
}
```

问：编译器的思维是怎样的？
为什么实现后面必须加说明？

§4. 函数

4.5. 函数的调用

4.5.3. 对被调用函数的说明

★ 调用说明可以在函数外，针对后面所有函数均适用；也可在函数内部，只对本函数有效

```
int my_max(int x,int y);
int main()
{
    ..my_max(...); ✓
}
int f1()
{
    ..my_max(...); ✓
}
int my_max(int x, int y)
{
    ....
}
```

```
int main()
{
    int my_max(int,int);
    ..my_max(...); ✓
}
int f1()
{
    ..my_max(...); ×
}
int my_max(int x, int y)
{
    ....
}
```

error C3861: “my_max”: 找不到标识符

```
int main()
{
    ..my_max(...); ?
}
int my_max(int x, int y);
int f1()
{
    ..my_max(...); ?
}
int my_max(int x, int y)
{
    ....
}
```

```
int main()
{
    int my_max(int,int);
    ..my_max(...); ?
}
int my_max(int x, int y)
{
    ....
}
int f1()
{
    ..my_max(...); ?
}
```

§4. 函数

4.5. 函数的调用

4.5.4. 实例

例1：求四个整数的最大值

```
//方法1
int max2(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}

int max4(int a, int b, int c, int d)
{
    int m;
    m = max2(a, b);
    m = max2(m, c);
    m = max2(m, d);
    return m;
}

int main()
{
   int a,b,c,d,m;

   ...输入a/b/c/d四个数字
   m = max4(a,b,c,d);
   ...输出最大值

   return 0;
}
```

```
//方法2
int max2(int a, int b)
{
    return (a>b ? a : b);
}

int max4(int a, int b, int c, int d)
{
    int m1, m2, m;
    m1 = max2(a, b);
    m2 = max2(c, d);
    m  = max2(m1, m2);
    return m;
}

int main()
{
   int a,b,c,d,m;

   ...输入a/b/c/d四个数字
   m = max4(a,b,c,d);
   ...输出最大值

   return 0;
}
```

```
//方法3
int main()
{
   ...
   m = max2( max2( max2(a,b), c), d);
   ...
}
```

```
//方法4
int main()
{
   ...
   m = max2( max2(a,b), max2(c,d) );
   ...
}
```

一个函数的返回值做为另一个函数的参数
(本例中函数名相同)

§4. 函数

4.5. 函数的调用

4.5.4. 实例

例2：写一个函数，判断某正整数是否素数

```
#include <iostream>
#include <cmath>
using namespace std;

int prime(int n)
{
    int i;
    int k = int(sqrt(n));

    for(i=2; i<=k; i++)
       if (n%i == 0)
          break;

    return i<=k ? 0 : 1;
}

int main()
{
    int n;
    cin >> n; //为简化讨论，此处假设输入正确
    cout << n << (prime(n) ? "是":"不是") << "素数" <<endl;
    return 0;
}
```

循环的结束有两个可能性：
1、表达式2（i<=k）不成立（是素数）
2、因为 break 而结束（不是素数）

```
//03模块例：求100~200间的素数
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int m, k, i, line=0;

    for(m=101; m<=200; m+=2) {
        k=int(sqrt(m));

        for(i=2; i<=k; i++)
            if (m%i==0)
                break;

        if (i>k) {
            cout << setw(5) << m;
            line++;
            if (line%10==0)
                cout << endl;
            }
        } //end of for

    return 0;
}
```

改写为用prime函数

```
//03模块例：求100~200间的素数
#include <iostream>
#include <iomanip>
using namespace std;
int prime(int n)
{   int i;
    int k = int(sqrt(n));

    for(i=2; i<=k; i++)
       if (n%i == 0)
          break;

    return (i<=k ? 0 : 1);
}
int main()
{
    int m, line = 0;
    for(m=101; m<=200; m+=2) {
        if (prime(m)) {
            cout << setw(5) << m;
            line++;
            if (line%10==0)
                cout << endl;
            }
        }
    return 0;
}
```

§4. 函数

4.5. 函数的调用

4.5.4. 实例

例3：验证哥德巴赫猜想

一道题目的解可用于另一题中
强调过程的积累、经验的积累

```
#include <iostream>
#include <cmath>
using namespace std;
int prime(int n)
{
    int i;
    int k = int(sqrt(n));
    for(i=2; i<k; i++)
       if (n%i == 0)
          break;
    return i<k ? 0 : 1;
}
void gotbaha(int even)
{
    int x;
    for (x=3; x<=even/2; x+=2)
        if ( prime(x) + prime(even-x) == 2) {
            cout << x << "+" << even-x << "=" << even <<endl;
            break; //不要break则求出全部组合
         }
}
int main()
{
    int n;
    cin >> n; //为简化讨论，此处假设输入正确
    gotbaha(n);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
18
5+13=18
```

Microsoft Visual Studio 调试控制台

```
18
5+13=18
7+11=18
```

§4. 函数

4.6. 函数的递归调用

4.6.1. 含义

函数直接或间接地调用本身

直接递归

```
f1()
{
...
f1()
...
}
```

间接递归

```
f1()      f2()
{         {
...       ...
f2()      f1()
...       ...
}         }
```

必然有条件判断是否进行下次递归调用!!!

★ 递归是指函数体中调用自己
★ 函数的返回值做本函数的参数，是嵌套，不是递归

```
int main()
{
    ...
    m = max2( max2( max2(a,b), c), d);

    m = max2( max2(a,b), max2(c,d) );
    ...
}
```

4.6.2. 递归的求解过程

回推：到一个确定值为止(递归不再调用)

递推：根据回推得到的确定值求出要求的解

例：求解第5个学生的年龄
题目描述：共5个学生
问第5个学生几岁，答：我比第4个大2岁；
问第4个学生几岁，答：我比第3个大2岁；
问第3个学生几岁，答：我比第2个大2岁；
问第2个学生几岁，答：我比第1个大2岁；
问第1个学生几岁，答：我10岁；

回溯

```
age(5) = age(4) + 2;
age(4) = age(3) + 2;
age(3) = age(2) + 2;
age(2) = age(1) + 2;
age(1) = 10;
```

递推

§4. 函数

4.6. 函数的递归调用

4.6.3. 如何写递归函数

★ 确定递归何时终止

★ 假设第n-1次调用已求得确定值，确定第n次调用和第n-1次调用之间存在的逻辑关系

=> 不要全面考虑1..n之间的变换关系，而应理解为只有n和n-1两层，且第n-1层数据已求得

例1：求解5个学生的年龄

```
int age(int n)
{
    if (n==1)
        return 10;
    else
        return age(n-1)+2;
}
```

```
int main()
{
    cout << age(5) << endl;
    return 0;
}
```

```
age(5) = age(4) + 2;
age(4) = age(3) + 2;
age(3) = age(2) + 2;
age(2) = age(1) + 2;
age(1) = 10;
```

§4. 函数

4.6. 函数的递归调用

4.6.3. 如何写递归函数

例2：采用非递归法和递归法两种方式求解n!

非递归法：

全面考虑1-n的关系，

可得出下列公式：

n! = 1*2*...*n;

```
int fac(int n)
{
    int s=1, i;
    for(i=1; i<=n; i++)
        s = s * i;
    return s;
}
```

递归法：

不全面考虑1-n的关系，

仅考虑n和n-1两层，

且假设n-1层已知

n!　　= n * (n-1)!

(n-1)! = n-1 *(n-2)!

...

1!　　= 1

0!　　= 1;

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

```
int main() //也可以由键盘输入n值，此处略
{   int n = 5;
    cout << n << "!=" << fac(5) << endl;
}
```

§4. 函数

4.6. 函数的递归调用

4.6.3. 如何写递归函数

例3：汉诺塔问题

§4. 函数

4.6. 函数的递归调用

4.6.3. 如何写递归函数

4.6.4. 如何读递归函数

★ 每次递归调用时，借助栈来记录调用的层次

★ 栈初始为空，每次递归函数被调用时在栈中增加一项，递归函数运行结束后栈中减少一项

★ 本次调用结束后，返回上次的调用位置，继续执行后续的语句

★ 重复操作至栈空为止

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(5)

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(4)	5
fac(5)	

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(3)	4
fac(4)	5
fac(5)	

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(1)	2
fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(1)	2	1
fac(2)	3	
fac(3)	4	
fac(4)	5	
fac(5)		

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(2)	3	2
fac(3)	4	
fac(4)	5	
fac(5)		

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(3)	4	6
fac(4)	5	
fac(5)		

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(4)	5	24
fac(5)		

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(5)		120

例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{   if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}
int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(5)=120

fac(1)	2	1
fac(2)	3	2
fac(3)	4	6
fac(4)	5	24
fac(5)		120

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k'); //VS中main无return不报错
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

7,k

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2, ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

例2：写出程序的运行结果

```
void f(int n, char ch)
{   if (n==0)
        return;
    if (n>1)
        f(n-2,ch);
    else
        f(n+1, ch);
    cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

mlnpr

0,k
2,k
1,k
3,k
5,k
7,k

例3：写出程序的运行结果及功能

```
void f(int n, int k)
{   if (n>=k)
        f(n/k, k);
    cout << n%k;
}
int main()
{   f(14, 2);      1110
    cout << endl;
    f(65, 8);      101
    return 0;
}
```

请用栈的方式
自行画图理解

§4. 函数

4.6. 函数的递归调用

4.6.5. 不设定终止条件的递归函数(错误的用法)

```
#include <iostream>
using namespace std;
int num = 0; //全局变量，后面4.11中详述
void fun()
{   num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

多编译器/多种模式，观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit

1、为什么会运行崩溃？
答：

2、不定义变量、定义10个int、10个double的情况下崩溃时打印的num值不同，为什么？
答：

3、有兴趣自行研究各编译器如何改变堆栈大小

```
#include <iostream>
using namespace std;
int num = 0; //全局变量，后面4.11中详述
void fun()
{   int a,b,c,d,e,f,g,h,i,j;
    a=b=c=d=e=f=g=h=i=j=10;
    num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

多编译器/多种模式，观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit

```
#include <iostream>
using namespace std;
int num = 0; //全局变量，后面4.11中详述
void fun()
{   double a,b,c,d,e,f,g,h,i,j;
    a=b=c=d=e=f=g=h=i=j=10;
    num ++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{   fun();
    return 0;
}
```

多编译器/多种模式，观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第02模块中：变量不能同名，不够准确)

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
...	...	...	...
}	a=15;	a=70000;	a=23;
	f2();	f3();	...
	}	}	}

● 在f3()执行时，三个a占用不同的内存空间(其中f1/f2中的int a/long a在"现场栈"中)，在f3中只能访问short a;

f2-f3
(a=70000)

f1-f2
(a=15)

main-f1

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
...	...	...	...
}	a=15;	a=70000;	a=23;
	f2();	f3();	...
	}	}	}

● 在f2()执行时，f1()/f2()的两个a占用不同内存空间(其中f1中的int a在"现场栈"中)，在f2中只能访问long a，而f3()中的a不占空间
(调用f3前则未分配，调用f3后则已释放)

f1-f2
(a=15)

main-f1

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第02模块中：变量不能同名，不够准确)

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
f2();	...	...	...
f3();	a=15;	a=70000;	a=23;
}	...	...	...
	}	}	}

● f1()/f2()/f3()中的三个a依次分配/释放，在不同时刻占用不同/相同(不保证)的内存空间，互不干扰

★ 形参等同于局部变量

int f1(int x)	int f2(long x)	int f3(int x)
{	{	{
...	...	...
}	}	}

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 复合语句内的变量，只在复合语句中有效(包括循环)

允许多层嵌套下各自定义属于自己作用范围的变量

```
void fun()
{
    int i,a;
    a=15;
    for(i=0;i<10;i++) {
        int y;
        y=11; ✓
        a=16; ✓
        }
    y=12; ×(超出复合语句的范围)
    a=17; ✓
}
```

error C2065: “y”: 未声明的标识符

```
void fun()
{
    int i,a;
    a=15;
    {
        int y;
        y=11; ✓
        a=16; ✓
    }
    y=12; ×(超出复合语句的范围)
    a=17; ✓
}
```

```
void fun()
{
    int i,a=15;
    {   int y;
        y=11; ✓
        a=16; ✓
        {   int w=10;
            y=12; ✓
            a=13; ✓
            w=14; ✓
        }
        w=15; ×(超出复合语句的范围)
    }
    y=12; ×(超出复合语句的范围)
    a=17; ✓
}
```

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名

★ 形参是局部变量

★ 复合语句内的变量，只在复合语句中有效(包括循环)

★ 在该函数的被调用函数内也无效(不可访问)

```
void f1()
{
    a=14;  ×
}
int main()
{   int a;
    a=15;
    f1();
    a=16;
}
```

```
void f1()
{ int a;
  a=14;  ✓
}
int main()
{ int a;
  a=15;
  f1();
  a=16;
}
```

两个a都是局部变量，分占不同的内存空间，当f1执行时，main中的a在____？

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

★ 从定义点到源文件结束之间的所有函数均可使用

```
int f1()
{ a=15; ×
}
int a;
int main()
{ a=16; ✓
}
int f2()
{ a=17; ✓
}
```

```
int a;
int f1()
{ a=15; ✓
}
int main()
{ a=16; ✓
}
int f2()
{ a=17; ✓
}
```

=> 递归函数中的局部变量/形参只能在本层被访问

```
#include <iostream>
using namespace std;
void fun(int n)
{
    int a = 10;
    cout << ++a << endl;
    if (n > 0)
        fun(--n);
}
int main()
{
    fun(3);
}
```

Microsoft Visual Studio 调试控制台

```
11
11
11
11
```

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

★ 从定义点到源文件结束之间的所有函数均可使用

★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果

```
#include <iostream>
using namespace std;

int a;
void f1()
{
    a=15;
}

int main()
{
    a=10;
    cout << "a=" << a << endl;  a=10
    f1();
    cout << "a=" << a << endl;  a=15
    return 0;
}
```

a | ~~10~~ 15

main()和f()访问的是同一个a，内存空间相同

```
#include <iostream>
using namespace std;

void f1(int a)
{
    a=15;
}

int main()
{
    int a =10;
    cout << "a=" << a << endl;  a=10
    f1(a);
    cout << "a=" << a << endl;  a=10
    return 0;
}
```

实参向形参单向传值的例子不一样!!!

a | 10 → a | ~~10~~ 15

main()和f()访问的是不同的a，占用不同的内存空间，单向传值

```
#include <iostream>
using namespace std;

int f1(int a)
{
    a=15;
    return a;
}

int main()
{
    int a =10;
    cout << "a=" << a << endl;  a=10
    a = f1(a);
    cout << "a=" << a << endl;  a=15
    return 0;
}
```

用函数返回值改变实参的值不一样!!!

a | ~~10~~ 15 → a | ~~10~~ 15

main()和f()访问的是不同的a，实参a是因为赋值语句而改变的

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

★ 从定义点到源文件结束之间的所有函数均可使用

★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果

=> 全局变量不在某函数被调用时被保存的"现场栈"中

=> 递归函数的各层均可以访问同一全局变量

```
#include <iostream>
using namespace std;
int a = 10;
void fun(int n)
{
    cout << ++a << endl;
    if (n > 0)
        fun(--n);
}

int main()
{
   fun(3);
}
```

Microsoft Visual Studio 调试控制台

```
11
12
13
14
```

§4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量

使用：

★ 在使用全局变量时应加以限制，提高程序的通用性和可靠性(别处的无意修改会导致结果变化)

=>本课程禁用全局变量(特别声明除外)

★ 若全局变量与局部变量同名，按"低层屏蔽高层"的原则处理(应尽量避免，以免理解错误)

全局变量和局部变量分别占用不同的内存空间

能否在f1()中访问全局变量a ？

C ：不能

C++：可以(后续模块内容)

```
#include <iostream>
using namespace std;
int a=10;
void f1()
{
    cout << "a=" << a << endl;
    int a=5;
    cout << "a=" << a << endl;
}
int main()
{
    f1();
    return 0;
}
```

全局a	10

局部a	5

Microsoft Visual Studio 调试控制台

```
a=10
a=5
```

```
#include <iostream>
using namespace std;
int a=10;
void f1()
{   int a=5;
    cout << "a1=" << a << endl;
}
void f2()
{   cout << "a2=" << a << endl;
}
int main()
{   f1();
    f2();
}
```

局部a	5

全局a	10

Microsoft Visual Studio 调试控制台

```
a1=5
a2=10
```

§4. 函数

4.7. 局部变量和全局变量

使用：

★ 若全局变量与局部变量同名，按"低层屏蔽高层"的原则处理(应尽量避免，以免理解错误)

=> "低层屏蔽高层"的规则同样适用于局部变量和复合语句内的局部变量同名

=> 在多层次嵌套的情况下允许不同层次的变量同名，遵循的基本规则是"低层屏蔽高层"

```
void f1()
{
  int a=5, i;
  for(i=0;i<10;i++) {
      int a=10;
      cout << "a=" << a;   a=10
      }
  cout << "a=" << a;       a=5
}
```

```
inline int f()
 {
     int a=5;
     cout << "fa=" << a << endl;
 }
 int main()
 {
     int a=10;
     f();
     cout << "ma=" << a << endl;
 }
```

问：inline应该这么理解吗？

```
int a=15;
void f1()
{
   int a=5, i;
   for(i=0;i<10;i++) {
      int a=10;
      if (i==5) {
         long a=20;
         cout << "a=" << a;    a=20
         }
      cout << "a=" << a;       a=10
      }
   cout << "a=" << a;          a=5
}
```

```
int main()
{
   int a=10;
   int a=5;
   cout << "fa=" << a << endl;
   cout << "ma=" << a << endl;
}
```

§4. 函数

4.8. 变量的存储类别

4.8.1. 应用程序执行时的内存分布

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中，变量占固定的存储空间
动态存储区	程序执行中，变量根据需要分配不同位置的存储空间

4.8.2. 局部变量的存储

4.8.2.1. 分类

- 自动变量：函数进入后，分配空间，函数运行结束后，释放空间（重复进行）

  1、假设main()中调用10000次f1()，则x,a的分配释放会重复10000次
  2、不保证每次x/a的空间与上次相同

```
int main()      void f1(int x)
{  ...          {
   f1(..);          int a;
   ...              ...
   f1(..);      }
   ...
}  //假设调用10000次f1()
```

  1、假设main()中调用10000次f1()，则a的分配释放只有1次(x仍为10000次)
  2、每次进入f1中，a都保持上次的值不变

```
int main()      void f1(int x)
{  ...          {
   f1(..);          static int a;
   ...              ...
   f1(..);      }
   ...
}  //假设调用10000次f1()
```

- 静态局部变量：变量所占存储单元在程序的执行过程中均不释放（无论函数体内外）

§4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.1. 分类

自动变量：函数进入后，分配空间，函数运行结束后，释放空间（重复进行）

★ 关于自动变量(auto)的新旧标准

- C++新标准中，缺省不写就是自动变量，而auto用来表示自动存储类型的变量
  =>新标准中，自动变量/auto变量是不同的变量
- C++旧标准中，缺省不写就是自动变量，也可以加auto来表示
  =>旧标准中，自动变量/auto变量是相同的变量

1、为适应多编译器，函数内的局部变量按正常定义，不加auto前缀
2、不准使用新标准的auto型变量(看得懂)
3、某些编译器默认使用旧标准，可通过加编译参数的方式使用新标准，具体方法略

```
#include <iostream>
using namespace std;
int main()
{
    auto int a;      //auto+类型
    int b=10;
    auto char c=2.1;//auto+类型

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

VS+Dev编译

```
#include <iostream>
using namespace std;
int main()
{
    auto a = 1;    //仅auto
    auto b = 'A'; //仅auto
    auto c=2.1;    //仅auto

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

VS+Dev编译

旧标准

```
int main()
{   auto int a;       //int 型自动变量
    int b=10;         //int 型自动变量(未加auto)
    auto char c=2.1; //char型自动变量
}
```

新标准

```
//auto变量不允许跟类型，定义时
  必须初始化，根据初始化值决定类型
int main()
{  auto int x;   //错误
   auto a=1;     //int型(换为1U,如何证明类型)
   auto b='A';   //char型
   auto f=1.0;   //double型(换为1.0F)
}
```

如果想使用auto自动类型，要对1/1LU/1.0/1.0F等常量的含义非常清晰，因此本课程禁止使用

静态局部变量：变量所占存储单元在程序的执行过程中均不释放（无论函数体内外）

§4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

★ 自动变量占动态存储区，静态局部变量占静态存储区，缺省声明为自动变量

★ 若定义时赋初值，自动变量在函数调用时执行，每次调用均重复赋初值；
静态局部变量在第一次调用时执行，以后每次调用不再赋初值，保留上次调用结束时的值

自动变量

```
#include <iostream>
using namespace std;
void f1()
{
    int a=1; //正常写，不加auto
    a++;
    cout << "a=" << a << endl;
}
int main()
{
    f1();   a=2
    f1();   a=2
    f1();   a=2
}
```

若定义时赋初值，自动变量在函数调用时执行，每次调用均重复赋初值

1、a的分配/释放重复了3次
2、3次的a不保证分配同一空间

静态局部变量

```
#include <iostream>
using namespace std;
void f1()
{
    static int a=1;
    a++;
    cout << "a=" << a << endl;
}
int main()
{
    f1();   a=2
    f1();   a=3
    f1();   a=4
}
```

静态局部变量赋初值在第一次调用时执行，以后每次调用不再赋初值，而保留上次调用结束时的值

1、a在第一次调用时分配空间并进行初始化，在3次退出/后2次调用中未再进行分配/释放
2、每次进入，a都是同一空间
3、在f1()内部，a可被访问，在f1()外部，a不能访问(但存在)

§4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

★ 自动变量占动态存储区,静态局部变量占静态存储区，缺省声明为自动变量

★ 若定义时赋初值，自动变量在函数调用时执行，每次调用均重复赋初值；
静态局部变量在第一次调用时执行，以后每次调用不再赋初值，保留上次调用结束时的值

```
#include <iostream>
using namespace std;
int f(int n)
{
    int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

?

```
#include <iostream>
using namespace std;
int f(int n)
{
    static int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

?

§4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

★ 自动变量占动态存储区，静态局部变量占静态存储区，缺省声明为自动变量

★ 若定义时赋初值，自动变量在函数调用时执行，每次调用均重复赋初值；
静态局部变量在第一次调用时执行，以后每次调用不再赋初值，保留上次调用结束时的值

★ 若定义时不赋初值，则自动变量的值不确定，静态局部变量的值为0（'\0'）

```
#include <iostream>
using namespace std;
int main()
{
    short a;
    static short b;
    static char c;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "c=" << (int)c << endl; //问：为什么要int？
    return 0;
}
```

a值：VS ：编译报错
Dev：不可预知值

b=0
c=0

VS	error C4700: 使用了未初始化的局部变量“a”

Dev

```
D:\WorkSpace\VS2019-Demo\cpp-demo\cpp-demo.exe
a=64
b=0
c=0
```

★ 函数的形参同自动变量

§4. 函数

4.8. 变量的存储类别

4.8.3. 寄存器变量

含义：对一些频繁使用的变量，可放入CPU的寄存器中，提高访问速度

(CPU访问寄存器比内存快一个数量级10^{-10}s vs 10^{-9}s)

```
register int a;
```

★ 仅对自动变量和形参有效(隐含含义：不能长期占用)

★ 编译系统会自动判断(即使定义了register，最终是否放入寄存器中，仍需要编译系统决定)

§4. 函数

4.8. 变量的存储类别

4.8.4. 用extern扩展全局变量的使用范围

原因：全局变量从定义点到源文件结束之间的所有函数均可使用，为了能在其它部分使用变量，需要进行使用范围的扩展

方法：在定义范围外使用全局变量时，应加上extern的说明，extern不分配存储空间，只说明对应关系

```
int f1()
{
    a=15;  ×
}
int a;
int main()
{
    a=16;  ✓
}

int f2()
{
    a=17;  ✓
}
```

```
extern int a;
int f1()
{
    a=15;  ✓
}
int a;
int main()
{
    a=16;  ✓
}
int f2()
{
    a=17;  ✓
}
```

不分配空间
说明对应关系

分配4字节空间

源程序文件ex1.cpp、ex2.cpp共同构成一个程序

ex1.cpp

```
int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
int f1()
{
    a=18; ×
}
int f3()
{
    a=19; ×
}
```

ex1.cpp

```
int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
extern int a;
int f1()
{
    a=18; ✓
}
int f3()
{
    a=19; ✓
}
```

分配4字节空间

不分配空间
说明对应关系

例：源程序文件ex1.cpp、ex2.cpp共同构成一个程序

ex1.cpp

```
int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
int f1()
{
    extern int a;
    a=18; ✓
}
int f3()
{
    a=19; ×
}
```

分配4字节空间

不分配空间
说明对应关系

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
int a;
int main()
{
    a=16;
}
int f2()
{
    a=17;
}
```

ex2.cpp

```
extern int a;
int f1()
{
    a=18;
}
int f3()
{
    a=19;
}
```

ex3.cpp

```
int a;
int f4()
{
    a=20;
}
```

对应哪个a?

§4. 函数

4.8. 变量的存储类别

4.8.5. 全局变量的存储

外部全局变量：所有源程序文件中的函数均可使用

(其它源程序文件中加extern说明)

静态全局变量：只限本源程序文件的定义范围内使用

(static)

★ 两者均在静态数据区中分配，不赋初值则自动为0

★ 不同源程序文件中的静态全局变量允许同名

★ 静态全局变量可与其它源程序文件中的外部全局变量同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
extern int a;
int f1()
{
    a=18; ×
}
int f3()
{
    a=19; ×
}
```

ex3.cpp

```
int f4()
{
    a=20; ×
}
```

分配4字节空间

无法与静态全局建对应关系

外部源程序文件无法访问静态全局变量

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;
int main()
{
    a=16;  ✓
}
int f2()
{
    a=17;  ✓
}
```

ex2.cpp

```
static int a;
int f1()
{
    a=18;  ✓
}
int f3()
{
    a=19;  ✓
}
```

ex3.cpp

```
int f4()
{
    a=20;
×
}
```

不同源程序文件中的静态全局变量允许同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;
int main()
{
    a=16;  ✓
}
int f2()
{
    a=17;  ✓
}
```

ex2.cpp

```
extern int a;
int f1()
{
    a=18;  ✓
}
int f3()
{
    a=19;  ✓
}
```

ex3.cpp

```
int a;
int f4()
{
    a=20;  ✓
}
```

静态全局变量可与其它源程序文件中的外部全局变量同名

分配4字节空间

分配4字节空间

不分配空间
说明对应关系

例：源程序ex1.cpp-ex4.cpp共同构成一个程序

```
ex1.cpp
static int a;
int main()
{
    a=16;
}
int f2()
{
    a=17;
}
```

```
ex2.cpp
extern int a;
int f1()
{
    a=18;
}
int f3()
{
    a=19;
}
```

```
ex3.cpp
int a;
int f4()
{
    a=20;
}
```

```
ex4.cpp
int a;
int f5()
{
    a=21;
}
```

情况1：正确/错误 ？

```
ex1.cpp
static int a;
int main()
{
    a=16;
}
int f2()
{
    a=17;
}
```

```
ex2.cpp
extern int a;
int f1()
{
    a=18;
}
int f3()
{
    a=19;
}
```

```
ex3.cpp
int a;
int f4()
{
    a=20;
}
```

```
ex4.cpp
static int a;
int f5()
{
    a=21;
}
```

情况2：正确/错误 ？

§4. 函数

4.9. 变量属性小结

4.9.1. 变量的分类

按类型：字符型、整型、浮点型等

按作用域
- 局部变量
- 全局变量

按存储方式（生存期）
- 动态存储变量
- 静态存储变量

按存储位置
- 内存变量
- 寄存器变量

§4. 函数

4.9. 变量属性小结

4.9.2. 不同类型变量对应的存储区

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中，变量占固定的存储空间
动态存储区	程序执行中，变量根据需要分配不同位置的存储空间

静态存储区	动态存储区	CPU寄存器
● 外部全局变量 ● 静态全局变量 ● 静态局部变量 ● 常量/常变量	● 自动变量 ● 函数形参 ● 堆(动态申请用，后续荣誉课)	● 寄存器变量

§4. 函数

4.9. 变量属性小结

4.9.3. 变量的生存期、作用域与链接性

★ 生存期：在什么时间存在，也叫持续性（时间概念）

- 存放在动态存储区的变量（动态存储）
- 存放在静态存储区的变量（静态存储）

★ 作用域：在什么范围内可以访问（空间概念）

- 只能在某个函数中被访问的变量（局部变量）
- 能够在多个函数中被访问的变量（全局变量）

★ 链接性：全局变量如何在不同单元间共享（共享概念）

- 在一个源程序文件的不同函数间共享（静态全局）
- 在多个源程序文件的不同函数间共享（外部全局）

	生存期	作用域	存储区
自动变量	本函数	本函数	动态数据区
形参	本函数	本函数	动态数据区
寄存器	本函数	本函数	CPU的寄存器
静态局部	程序执行中	本函数	静态数据区
静态全局	程序执行中	本源程序文件	静态数据区
外部全局	程序执行中	全部源程序文件	静态数据区

§4. 函数

4.10. 变量的声明与定义

定义：指定变量的类型，名称并分配存储空间

声明：指明变量的相互关系，不分配存储空间

```
int a;          定义
extern int a;   声明
```

§4. 函数

4.11. 内部函数和外部函数

内部函数：仅能在本源程序中被调用的函数

static 返回类型 函数名（形参表）

★ 不同的源程序文件中可以同名

外部函数：可以在所有的源程序文件中被调用

★ 本源程序文件中直接使用

★ 其它源程序文件中加函数说明

（可以加extern，也可以不加）

例：源程序文件ex1.cpp、ex2.cpp共同构成一个程序

外部源程序文件
无法访问内部函数

ex1.cpp

```
static float f2();
int main()
{
    f2();    ✓
}
static float f2()
{
    ...
}
int f1()
{
    f2();    ✓
}
```

ex2.cpp

```
int f3();
{
    f2();  ×
}
```

源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static float f2();
int main()
{
    f2();    ✓
}
static float f2()
{
    ...
}
int f1()
{
    f2();    ✓
}
```

ex2.cpp

```
int f3();
{
    f2(); ×
}
```

ex3.cpp

```
static char f2();
int f4()
{
    f2();    ✓
}
static char f2()
{
    ...
}
```

不同的源程序文件中的内部函数可以同名

ex1.cpp

```
float f2();
int main()
{
    f2();    ✓
}
float f2()
{
    ...
}
int f1()
{
    f2();    ✓
}
```

ex2.cpp

```
int f3();
{
    f2(); ×
}
```

ex3.cpp

```
extern float f2();
int f4()
{
    f2();    ✓
}
```

在其它源程序文件中加函数说明可以访问外部函数

extern可要可不要

例：源程序ex1.cpp-ex3.cpp共同构成一个程序

```
ex1.cpp
float f2();
int main()
{
    f2();
}
float f2()
{
    ...
}
```

```
ex2.cpp
static float f2()
int f3();
{
    f2();
}
float f2()
{
    ...
}
```

```
ex3.cpp
extern float f2();
int f4()
{
    f2();
}
```

情况1：正确/错误?

```
ex1.cpp
float f2();
int main()
{
    f2();
}
float f2()
{
    ...
}
```

```
ex2.cpp
float f2()
int f3();
{
    f2();
}
float f2()
{
    ...
}
```

```
ex3.cpp
extern float f2();
int f4()
{
    f2();
}
```

情况2：正确/错误?

§4. 函数

4.12. 头文件

4.12.1. 头文件的内容及作用

头文件的内容：

- ★ 结构体类型(struct-后续模块)及类(class-后续模块)的声明
- ★ 函数的声明
- ★ inline函数的定义与实现
- ★ 符号常量的定义及常变量的定义
- ★ 全局变量的extern声明
- ★ 其它需要的头文件

例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//引用100个函数
#include <iostream>
using namespace std;

void f1();
...
void f100();

int main()
{
   f1();
   ...
   f100();
}
```

#include中<>和""的区别先忽略

```
//ex2.cpp
//共100个函数

void f1()
{
   ...
}
...
...
void f100()
{
   ...
}
```

```
//ex3.cpp
//引用100个函数
#include <iostream>
using namespace std;

void f1();
...
void f100();

void fun()
{
   f1();
   ...
   f100();
}
```

问题：
函数定义的声明被多处重复，若修改了某个函数的定义，则需要修改多处，会造成不一致

```
//ex1.cpp
//引用100个函数
#include <iostream>
using namespace std;
#include "ex.h"

int main()
{
   f1();
   ...
   f100();
}
```

程序由ex1.cpp、ex2.cpp、ex3.cpp、ex.h（新增）组成

```
//ex.h
//100个函数的声明
void f1();
...
...
void f100();
```

```
//ex3.cpp
//引用100个函数
#include <iostream>
using namespace std;
#include "ex.h"

void fun()
{
   f1();
   ...
   f100();
}
```

通过头文件使维护简单，避免多处修改导致的不一致性

例：程序由ex1.cpp、ex2.cpp共同构成

```cpp
//ex1.cpp
#include <iostream>
using namespace std;
inline void f1()
{
   ...
}

int main()
{
   f1();
   ...
   f1();
}
```

```cpp
//ex2.cpp
#include <iostream>
using namespace std;
inline void f1()
{
   ...
}

void fun()
{
   f1();
   ...
   f1();
}
```

问题：
因为inline函数必须和调用函数处在同一个源文件中，导致多处重复

```cpp
//ex1.cpp
#include <iostream>
using namespace std;

#include "ex.h"

int main()
{
   f1();
   ...
   f1();
}
```

程序由ex1.cpp、ex2.cpp、ex.h（新增）组成

```cpp
//ex.h

inline void f1()
{
   ...
}
```

```cpp
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
   f1();
   ...
   f1();
}
```

通过头文件使维护简单，避免多处修改导致的不一致性

例：程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

int main()
{
   ...pi...
   ...
   ...x...
}
```

```
//ex2.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

void fun()
{
   ...pi...
   ...
   ...x...
}
```

问题：
符号常量及常变量在多处定义，导致重复定义以及维护困难

```
//ex1.cpp
#include <iostream>
using namespace std;

#include "ex.h"

int main()
{
   ...pi...
   ...
   ...x...
}
```

程序由ex1.cpp、ex2.cpp、ex.h（新增）组成

```
//ex.h

#define pi 3.14159
const int x=10;
```

```
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
   ...pi...
   ...
   ...x...
}
```

通过头文件使维护简单，避免多处修改导致的不一致性

例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
   ...x...
}
```

```
//ex2.cpp
//引用全局变量

extern int x;
void f1()
{
   ...x...
}

void f2()
{
   ...x...
}
```

```
//ex3.cpp
//引用全局变量

extern int x;
void fun()
{
   ...x...
}
```

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
   ...x...
}
```

```
//ex2.cpp
//引用全局变量

#include "ex.h"
void f1()
{
   ...x...
}

void f2()
{
   ...x...
}
```

```
//ex3.cpp
//引用全局变量

#include "ex.h"
void fun()
{
   ...x...
}
```

程序由ex1.cpp、ex2.cpp、ex3.cpp、ex.h（新增）组成

```
//ex.h
//全局变量声明

extern int x;
```

例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

int x=10;

int main()
{
   ...x...
}
```

```
//ex2.cpp
//引用全局变量

extern int x;
void f1()
{
   ...x...
}

void f2()
{
   ...x...
}
```

```
//ex3.cpp
//引用全局变量

extern int x;
void fun()
{
   ...x...
}
```

```
//ex1.cpp
//定义全局变量

#include <iostream>
using namespace std;

#include "ex.h"

int main()
{
   ...x...
}
```

```
//ex2.cpp
//引用全局变量

#include "ex.h"
void f1()
{
   ...x...
}

void f2()
{
   ...x...
}
```

```
//ex3.cpp
//引用全局变量

#include "ex.h"
void fun()
{
   ...x...
}
```

程序由ex1.cpp、ex2.cpp、ex3.cpp、ex.h（新增）组成

```
//ex.h
//全局变量定义
int x;   //错误
```

注:1.若头文件中包含全局变量定义，则被多个文件包含会导致重复定义

2.头文件中可包含静态全局/只读变量，但不建议

```
const int x = 10;
static int x = 15;
```

例：程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp

#include <iostream>
using namespace std;


int main()
{
   ...
}
```

```
//ex2.cpp

#include <iostream>
using namespace std;


void fun()
{
   ...
}
```

程序由ex1.cpp、ex2.cpp、ex.h（新增）组成

```
//ex1.cpp

#include "ex.h"

int main()
{
   ...
}
```

```
//ex.h

#include <iostream>
using namespace std;
```

```
//ex2.cpp

#include "ex.h"

void fun()
{
   ...
}
```

§4. 函数

4.12. 头文件

4.12.1. 头文件的内容及作用

头文件的作用：

★ 将编程者需要的在不同源程序文件传递的各种信息归集在一起，方便多次调用以及集中修改

★ 在一个源程序文件中包含头文件时，头文件的所有内容会被理解为包含到 #include 位置处，编译时（变量的定义及函数作用域等）均当作一个文件进行处理

头文件的包含方式：

#include <文件名>：直接到系统目录中寻找，找到则包含进来，找不到则报错

#include "文件名"：先在当前目录中寻找，找到则包含进来，
找不到则再到系统目录中寻找，找到则包含进来，找不到则报错

VS2022如果缺省安装，则头文件的目录为

64位Windows操作系统：

C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\xx.xx.xxxxx\include

具体版本号

§4. 函数

例1:理解<>和""的差别

例：在当前目录下有demo.h文件
内容：
```
int a=10;
```

源程序文件demo.c的内容
```
#include <iostream>
using namespace std;

#include <demo.h>
int main()
{
    cout << a << endl;
    return 0;
}
```
编译报错,因为<>不寻找当前目录中是否有demo.h

源程序文件demo.c的内容
```
#include <iostream>
using namespace std;

#include "demo.h"
int main()
{
    cout << a << endl;
    return 0;
}
```
编译正确

例2:理解<>和""的差别

例：在当前目录下有demo.h文件
内容：
```
int a=10;
```

例：在系统目录下有demo.h文件
内容：
```
int b=10;
```

源程序文件demo.c的内容
```
#include <iostream>
using namespace std;

#include <demo.h>
int main()
{
    cout << b << endl;
    return 0;
}
```
编译正确

源程序文件demo.c的内容
```
#include <iostream>
using namespace std;

#include "demo.h"
int main()
{
    cout << b << endl;
    return 0;
}
```
编译报错,因为""方式找到的是当前目录,无b的定义

§4. 函数

4.12. 头文件

4.12.1. 头文件的内容及作用

4.12.2. C++的标准库及头文件

C++包含系统头文件的两种形式：

```
#include <math.h>  ：C形式
#include <cmath>   ：C++形式
```

两种方式都是指编译系统的include目录的math.h

VS2022如果缺省安装，则头文件的目录为

64位Windows操作系统：

C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\xx.xx.xxxxx\include

具体版本号

附：关于全局变量使用的基本原则(实际工作中)

1、尽量不用

2、如果实在需要，尽量使用静态全局

3、如果静态全局不能满足要求，尽量在调用函数中进行extern声明

这三点，可理解为
权限最小化原则

```
//ex1.cpp
int a;
fun1()
{
    ...
}

fun2()
{
    ...
}
```

```
//ex2.cpp
f1()
{
}
...
f34()
{
    extern int a;
}
...
f173()
{
    extern int a;
}
...
f1000()
{
}
```

宁可多处重复，也不要直接放在最前面

4、给全局变量特殊的命名规则

例：_**　　//下划线开始

_**_　　//下划线开始+结尾

_zs_**_　　//特定串做前缀（假设姓名为张三）

5、#define的宏定义及const常变量不在限制范围内，鼓励多用(多处使用相同值时尽量使用)

§4. 函数

4.13. 内联函数

形式：

```
inline 返回类型 函数名（形式参数表）
{
   函数体
}

inline int max(int x, int y)
{
    return x>y?x:y;
}
```

§4. 函数

4.13. 内联函数

使用：

★ 不单独编为一段代码，而是直接插入每个调用处，调用时不按函数调用过程执行，而是直接将该函数的代码放在调用处顺序执行

```
int main()          void f(void)
{  ...              {
   f();                ***
   ...                 ***
}                   }
```

在可执行代码中有f()及main()的存在，按正常函数调用方式进行

```
int main()          inline void f(void)
{  ...              {
   ***                 ***
   ***                 ***
   ...              }
}
```

可执行代码中无f(),仅有main(),main()代码长度增加，但没有保存及恢复现场的消耗

§4. 函数

4.13. 内联函数

```
int main()
{   ...
    f();
    ...
    f();
    ...
    f();
    ...
    f();
    ...
}
```

```
inline void f(void)
{
    cout ...;
}
```

假设f()被调用了10000次

```
int main()
{   ...
    cout ...
    ...
    cout ...
    ...
    cout ...
    ...
    cout ...
    ...
}
```

相当于

```
inline void f(void)
{
    cout ...;
}
```

执行代码中f()已不存在

可执行代码中无f(),仅有main(),main()代码中包含了10000份f()的代码，长度增加，但没有保存及恢复现场的消耗

以空间的增加换取时间的加快

问：为什么不去掉f()，直接在main()中写10000次cout？

答：和前面定义符号常量一样
#define pi 3.14159
便于源程序的修改和维护

§4. 函数

4.13. 内联函数

使用：

★ 不单独编为一段代码，而是直接插入每个调用处，调用时不按函数调用过程执行，而是直接将该函数的代码放在调用处顺序执行

★ 可执行程序的代码长度增加，但执行速度加快，适用于函数体短小且调用频繁的情况（1-5行）
(保存/恢复现场的代价超过函数体自身代价的情况)

★ 不能包含分支、循环等复杂的控制语句

★ 系统编译时会自动判断是否需要真正采用内联方式
(写了inline，最终也不一定真正成为内联函数)

★ 递归函数不能内联(递归必须要保存/恢复现场)

★ 允许只在函数声明或函数定义中加inline，也可以同时加

不同编译器，三种情况可能都正确/部分正确(VS/Dev中三种都正确)		

inline void fun();
int main()
{
...
}
inline void fun()
{
...
}
	inline void fun();
	int main()
	{
	...
	}
	~~inline~~ void fun()
	{
	...
	}
		~~inline~~ void fun();
		int main()
		{
		...
		}
		inline void fun()
		{
		...
		}

§4. 函数

4.13. 内联函数

使用：

★ inline函数及调用函数必须在同一个源程序文件中，否则编译出错

（普通函数可以放在不同源程序文件中）

//ex1.cpp	//ex2.cpp

inline void fun();
int main()
{
...
}
	inline void fun()
	{
	...
	}

假设ex1.cpp和ex2.cpp共同构成一个可执行文件，则编译出错

//ex1.cpp	//ex2.cpp

void fun();
int main()
{
...
}
	void fun()
	{
	...
	}

假设ex1.cpp和ex2.cpp共同构成一个可执行文件，则编译正确

§4. 函数

4.14. 函数的重载

重载：同一作用域中多个函数使用相同的名称

引入：对同一类功能的实现，仅参数的个数或类型不同，希望采用相同的函数名

C不允许
C++允许

```
      int   imax(int x,   int y);
      float fmax(float x, float y);
      long  lmax(long x,  long y);
====> 希望 imax/fmax/lmax 都叫 max ?
      int   max(int x,   int y);
      float max(float x, float y);
      long  max(long x,  long y);
```

```
      int max2(int x, int y);
      int max3(int x, int y, int z);
      int max4(int x, int y, int z, int w);
====> 希望 max2/max3/max4 都叫 max ?
      int max(int x, int y);
      int max(int x, int y, int z);
      int max(int x, int y, int z, int w);
```

例：分别求两个int和double型数的最大值

```
int max(int x, int y)
{   cout << sizeof(x) << endl;
    return (x > y ? x : y);
}
double max(double x, double y)
{   cout << sizeof(x) << endl;
    return (x > y ? x : y);
}
int main()
{   cout << max(10,   15)   << endl;
    cout << max(10.2, 15.3) << endl;
}
```

?

例：分别求两个/三个int数的最大值

```
int max(int x, int y)
{   cout << 2 << ' ';
    return (x > y ? x : y);
}
int max(int x, int y, int z)
{   cout << 3 << ' ';
    int t = (x > y ? x : y);
    return (t > z ? t : z);
}
int main()
{   cout << max(10, 17)    << endl;
    cout << max(23, 15, 8) << endl;
}
```

?

§4. 函数

4.14. 函数的重载

重载：同一作用域中多个函数使用相同的名称

引入：对同一类功能的实现，仅参数的个数或类型不同，希望采用相同的函数名

重载函数调用时的匹配查找顺序：

（1）寻找参数个数、类型完全一致的定义(严格匹配)

（2）通过系统定义的转换寻找匹配函数

（3）通过用户定义的转换寻找匹配函数

★ 若某一步匹配成功，则不再进行下一顺序的匹配

★ 若某一步中发现两个以上的匹配则出错

例：分别求两个int和double型数的最大值

```
#include <iostream>
using namespace std;
int max(int x, int y)
{   cout << sizeof(x) << ' ';
    return (x > y ? x : y);
}
double max(double x, double y)
{   cout << sizeof(x) << ' ';
    return (x > y ? x : y);
}
int main()
{   cout << max(10,    15)      << endl;    //int, int
    cout << max(10.2, 15.3)     << endl;    //double, double
    cout << max(10,    int(15.3)) << endl; //int,double
    cout << max(5+4i, 15.3)     << endl;    //复数，double
    return 0;
}
```

1

2

哪句语句编译会错?
其它正确语句的输出是什么？

复数形式目前编译会错，如何定义复数以及定义复数向double的转换，具体见荣誉课相关内容

严格匹配1
严格匹配2
系统转换1
需自定义转换

自行将max的参数换成U/L/F等不同组合，看是否报错，按什么类型做系统转换

§4. 函数

4.14. 函数的重载

使用：

★ 要求同名函数的参数个数、参数类型不能完全相同

void fun(int x, int y);	正确
void fun(int x, int y, int z);	参数个数不同,类型同
void fun(int x, int y);	正确
void fun(long x, long y);	参数个数同,类型不同
void fun(int x, int y);	正确
void fun(long x, long y, long z);	个数类型均不同
void fun(int x, int y);	错误
void fun(int x, int y);	个数类型均相同

★ 返回类型及参数名不做检查(仅这两个不同认为错)

int max(int x, int y);	错误,仅返回类型不同
long max(int x, int y);	参数类型个数完全相同
int max(int x, int y);	错误,仅参数名不同
int max(int p, int q);	参数类型个数完全相同

★ 若参数类型是由typedef定义的不同名称的相同类型，则会产生二义性

typedef INTEGER int;	相当于给int起个别名叫INTERGER，具体见第7章
int fun(int a);	错误
INTEGER fun(INTEGER a);	

★ 尽量使同名函数完成相同或相似的功能，否则可能导致概念混淆

§4. 函数

4.15. 函数模板

函数重载的不足：对于参数个数相同，类型不同，
而实现过程完全相同的函数，
仍要分别给出各个函数的实现

问题:两段一样的代码
能否合并为一段？

```
int max(int x, int y)
{
    return x>y?x:y;
}
double max(double x, double y)
{
    return x>y?x:y;
}
```

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为函数模板，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

一段代码,两个功能
1、两个int型求max
2、两个double型求max

问题1：如果传入两个unsigned int型数据，
T的类型被实例化为什么？如何证明？
问题2：如果x,y的类型不同，自行摸索转换规律

```
#include <iostream>
using namespace std;
template <typename T>
T max(T x, T y)
{   cout << sizeof(x) << ' ';
    return x>y?x:y;
}
int main()
{   int     a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << max(a, b) << endl;
    cout << max(f1, f2) << endl;
    return 0;
}
```

```
4 15
8 23.45
```

§4. 函数

4.15. 函数模板

函数重载的不足：对于参数个数相同，类型不同，而实现过程完全相同的函数，仍要分别给出各个函数的实现

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为函数模板，
调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

使用：

★ 仅适用于参数个数相同、类型不同，实现过程完全相同的情况

★ typename可用class替代

★ 类型定义允许多个

```
template <typename T1, typename t2>
template <class T1, class t2>
```

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
char max(T1 x, T2 y)
{   cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}
int main()
{   int    a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << max(a, f1) << endl;
    cout << max(f2, b) << endl;
    return 0;
}
```

?

问：max(a,f1)时，T1/T2类型分别是?
max(f2,b)时，T1/T2类型分别是?

§4. 函数

4.16. 有默认参数的函数

引入：假设已经定义了某个函数，并进行了大量的应用后来随着要求的增加，需要扩充函数的功能并且增加相应的参数来满足扩充的功能

例：定义 circle(int x, int y)用于画圆心在(x,y)处半径为10的圆，并已被调用1000次

```
int main()
{
    ...
    circle(...);
    ...
    circle(...);
    ...
    circle(...);
    ...
    circle(...);
    ...
}   //有1000次调用

void circle(int x, int y)
{
    //具体实现过程
}
```

经过不断的测试，程序已稳定运行

例：增加要求，要求半径可变，前面已调用的1000次中900次维持半径为10不变，100次改为不同值，又新增调用1000次

```
int main()
{
    ...
    circle(.旧.);
    ...
    circle(.旧.);
    ...
    circle(.新.);
    ...
    circle(.新.);
    ...
}
```

```
void circle(int x,int y,int r)
{
    //具体实现过程
}
```

首先：修改circle的定义及实现

其次：修改旧的1000次调用语句，从两参改为三参

最后：新增1000次三参调用

§4. 函数

4.16. 有默认参数的函数

例：一个程序要求的不断演变

1、定义 circle(int x, int y)用于画圆心在(x,y)处半径为10的圆，并已被调用1000次

2、增加要求，要求半径可变，前面已调用的1000次中900次维持半径为10不变，100次改为不同值，又新增调用1000次

3、增加要求，要求指定不同的颜色，前面已调用过的2000次中1800次保持白色，200次改为其它颜色，又新增调用1000次

4、新增要求，要求指定空心还是实心，前面已调用过的3000次中2700次都是空心，300次改为实心，又新增调用1000次

工程思维的基本概念：

1、使程序稳定运行所需要的测试工作工作量很大

2、一旦修改了程序，原来稳定运行的部分也可能出现各种问题，需要重新测试

3、新功能的增加是必须的

问题：能否在功能增加的同时使程序的改动尽可能少？

§4. 函数

4.16. 有默认参数的函数

引入：假设已经定义了某个函数，并进行了大量的应用后来随着要求的增加，需要扩充函数的功能并且增加相应的参数来满足扩充的功能

含义：对函数的某一形参，大部分情况下都对应同一个实参值时，可以采用默认参数(默认值建议为常量)

形式：

```
返回类型 函数名(无默认参数形参,有默认参数形参)
{
    函数体
}

void circle(int x, int y, int r=10)
{
    ...
}
```

调用：circle(0,0); ⇔ circle(0,0,10);

circle(5,8,12);

§4. 函数

4.16. 有默认参数的函数

使用：

★ 便于函数功能的扩充，减少代码维护，修改的数量

针对刚才的例子：

=> 1、两个参数的原始程序完成，调用1000次

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

=> 2、加半径参数，不变900处，改100处，增1000处

```
void circle(int x, int y, int r=10)
```

=> 3、加颜色参数，不变1800处，改200处，增1000处

```
void circle(int x, int y, int r=10, int color=WHITE)
```

=> 4、加填充参数，不变2700处，改300处，增1000处

```
void circle(int x, int y, int r=10, int color=WHITE, int filled=NO)
```

结论：

1、有默认参数的函数，能有效地减少了修改次数，减少了工作量

2、最好的方法，是在初始设计函数时，就考虑到更多可能的因素（包括客户暂时未想到的问题）

§4. 函数

4.16. 有默认参数的函数

使用：

★ 便于函数功能的扩充，减少代码维护，修改的数量

★ 允许有多个默认参数，但必须是连续的最后几个

```
void circle(int y, int x=0, int r=5)(对)
void circle(int x=0, int y, int r=5)(错)
```

★ 若有多个默认参数，调用时，前面使用缺省值，后面不使用缺省值，则前面也要加上

```
void circle(int x,int y,int r=5,int c=WHITE)
circle(10, 15);
circle(10, 15, 10);
circle(10, 15, 12, BLUE);
circle(10, 15, 5, BLUE);
```

虽然是缺省，也要加

§4. 函数

4.16. 有默认参数的函数

使用：

★ 若函数定义在调用函数之后，则声明时必须给出默认值，定义时不再给出

```
void circle(int x, int y, int r=10);
int main()
{   ...
}
void circle(int x, int y, int r)
{   ...
}
```

正确

```
void circle(int x, int y, int r=10);
int main()
{   ...
}
void circle(int x, int y, int r=10)
{   ...
}
```

错误，即使相同

```
void circle(int x, int y, int r);
int main()
{   ...
}
void circle(int x, int y, int r=10)
{   ...
}
```

错误

```
void circle(int x, int y, int r=10);
int main()
{   ...
}
void circle(int x, int y, int r=5)
{   ...
}
```

错误

★ 重载与带默认参数的函数一起使用时，可能会产生二义性

```
int fun(int a, int b=10);
int fun(int a);
```

若调用为：fun(10，20)　正确

fun(50)　二义性

§4. 函数

4.17. 带参数的main函数

4.17.1. 引入

可执行文件运行时，目前只能简单的运行，如果能加上参数，则使用中可以更灵活

例1：两数交换(常规方法)

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, t;
    cout << "请输入两个整数" << endl;
    cin >> a >> b;
    cout << "交换前：a=" << a << "  b=" << b << endl;
    t = a;
    a = b;
    b = t;
    cout << "交换后：a=" << a << "  b=" << b << endl;

    return 0;
}
```

§4. 函数

4.17. 带参数的main函数

4.17.1. 引入

可执行文件运行时，目前只能简单的运行，如果能加上参数，则使用中可以更灵活

4.17.2. 带参数的main函数的定义形式

```
int main(int argc, char **argv)
int main(int argc, char *argv[])
```

两者均可

★ 参数解释

argc：参数的个数，若不带参数，则为1(自身)

argv：参数的内容，用指针数组表示，每个元素是一个字符串(char *)，最后一个是 NULL

- argv数组共有argc+1个元素，下标[0]~[argc]（例如：argc为3，则argv[0]是自身，argv[3]是NULL）
- 参数名argc/argv可变，类型不能变（例如：int ac, char **av）
- VS系列可以 long ac,unsigned char **av, gcc系列不可以，因此不建议其它类型

§4. 函数

4.17. 带参数的main函数

4.17.3. 使用

例2：两数交换(main函数带参数方法)

```
#include <iostream>
#include <cstdlib>  //atoi函数用到
using namespace std;
int main(int argc, char *argv[])
{
    int a, b, t;

    cout << "argc="   << argc    << endl;
    cout << "argv[0]="<< argv[0] << endl;
    a = atoi(argv[1]); //atoi是将字符串转为整数的函数
    b = atoi(argv[2]);

    cout << "交换前:a=" << a << "  b=" << b << endl;
    t = a;
    a = b;
    b = t;
    cout << "交换后:a=" << a << "  b=" << b << endl;

    return 0;
}
```

假设编译后形成形成demo.exe

1、集成环境运行　　(出错,为什么？)

2、命令行运行

　　demo　　(出错,为什么？)

　　demo 10　　(出错,为什么？)

　　demo 10 15　　(正确)

　　demo 10 15 20 (正确)

§4. 函数

4.17. 带参数的main函数

4.17.3. 使用

例3：两数交换(main函数带参数方法 - 改进)

```
#include <iostream>
#include <cstdlib>  //atoi函数用到
using namespace std;
int main(int argc, char *argv[])
{
    int a, b, t;
    if (argc<3) { /* 参数不足3个则出现提示 */
        cout << "请带两个整数作为参数"<< endl;
        return -1;
        }
    for (t=0; t<argc; t++)    /* 打印所有的参数值 */
         cout << "argv[" << t << "]=" << argv[t] << endl;
    a = atoi(argv[1]); //atoi是将字符串转为整数的函数
    b = atoi(argv[2]);
    cout << "交换前：a=" << a << "  b=" << b << endl;
    t = a;
    a = b;
    b = t;
    cout << "交换后：a=" << a << "  b=" << b << endl;
    return 0;
}
```

假设编译后形成形成
demo.exe
1、集成环境运行
2、命令行运行
    demo
    demo 10
    demo 10 15
    demo 10 15 20

§4. 函数

4.17. 带参数的main函数

4.17.4. 综合应用

例4：作业相似度检查程序的参数设计

(1) 学生的匹配

要求能在两个特定的学生之间检查

某个特定学生和全体学生之间检查

全体学生之间相互检查

(2) 文件的匹配

要求既可以是单文件，也可以全部文件

(3) 相似度设置

要求值在60-100间浮动

(4) 输出方式

可选文件/屏幕

```
假设Linux下编译后形成形成 check，下列方式都正确
  ./check 2159999 2159998 12-b2.cpp 80
  ./check 2159999 2159998 all       80
  ./check 2159999 all     12-b2.cpp 75 result.txt
  ./check 2159999 all     all       85
  ./check all     all     all       85 final.txt
```

§4. 函数

4.17. 带参数的main函数

4.17.5. 参数个数不固定的带参main函数

例5：在Windows的命令行下输入 ping，可以看到ping 命令的很多选项，下列命令都是正确的

```
ping 10.10.108.117
ping -t 10.10.108.117
ping -n 10 10.10.108.117
ping -n 10 -l 50000 192.168.80.230
ping -t -l 50000 192.168.80.230
ping -l 50000 -t 192.168.80.230
```

★ 参数个数不固定，且部分参数要2个一组

★ 参数出现顺序任意

思考：如果输入ping后用人机交互形式，该如何做？从用户操作方便性角度而言，可行吗？

cmd.exe

```
D:\>ping

用法: ping [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS]
           [-r count] [-s count] [[-j host-list] | [-k host-list]]
           [-w timeout] [-R] [-S srcaddr] [-c compartment] [-p]
           [-4] [-6] target_name

选项:
    -t             Ping 指定的主机，直到停止。
                   若要查看统计信息并继续操作，请键入 Ctrl+Break；
                   若要停止，请键入 Ctrl+C。
    -a             将地址解析为主机名。
    -n count       要发送的回显请求数。
    -l size        发送缓冲区大小。
    -f             在数据包中设置“不分段”标记(仅适用于 IPv4)。
    -i TTL         生存时间。
    -v TOS         服务类型(仅适用于 IPv4。该设置已被弃用，
                   对 IP 标头中的服务类型字段没有任何
                   影响)。
    -r count       记录计数跃点的路由(仅适用于 IPv4)。
    -s count       计数跃点的时间戳(仅适用于 IPv4)。
    -j host-list   与主机列表一起使用的松散源路由(仅适用于 IPv4)。
    -k host-list   与主机列表一起使用的严格源路由(仅适用于 IPv4)。
    -w timeout     等待每次回复的超时时间(毫秒)。
    -R             同样使用路由标头测试反向路由(仅适用于 IPv6)。
                   根据 RFC 5095，已弃用此路由标头。
                   如果使用此标头，某些系统可能丢弃
                   回显请求。
    -S srcaddr     要使用的源地址。
    -c compartment 路由隔离舱标识符。
    -p             Ping Hyper-V 网络虚拟化提供程序地址。
    -4             强制使用 IPv4。
    -6             强制使用 IPv6。


D:\>_
```

§4. 函数

4.17. 带参数的main函数

4.17.6. 带参数的main函数的扩展形式(了解)

形式： int main(int argc, char **argv, char **env)

或： char *env[]

参数解释：
- argc：同前
- argv：同前
- env ：操作系统的环境变量，用指针数组来表示，每个元素是一个字符串(char *)，最后一个元素是NULL

使用：需要判断/取操作系统的某些设置时才用到

例6：取操作系统的环境变量(在Windows/Linux下分别运行)

```
#include <iostream>
using namespace std;

int main(int argc, char **argv, char **env)
{
    int i;
    for (i=0; env[i]; i++)
        cout<< "env[" << i << "]=" << env[i] <<endl;

    return 0;
}
```

拓展问题：如何在Windows/Linux下增加一个环境变量？

§4. 函数

4.17. 带参数的main函数

4.17.7. 带参数main函数的作用

	带参main函数方式	运行时键盘交互方式
运行方法	运行命令后直接跟各参数，不再进行人机交互	运行命令后进入人机交互
是否需要人机交互	不需要人机交互	需要人机交互 (可用输入重定向方式取消人机交互，但不方便)
适用程序	1、守护进程(开机自启动) 2、后台运行程序 3、类似ping的不定参数形式命令，必须用此形式	前端程序

§5. 数组

5.1. 基本概念

5.1.1. 引入

在表示若干相同类型、含义的元素时，引入数组，避免了用多个变量表示的不统一性

5.1.2. 数组的组成

数组名+下标

（标识在整个数组中所处的位置，即序号）

5.1.3. 要求

数组中元素的类型相同

例1：输入3个人的成绩，打印平均值

S中只保留了最后一次的输入

```
int main()
{   int i, s, sum = 0;
    for(i=0; i<3; i++) {
       cin >> s;  //假设输入都正确
       sum += s;
       }
    cout << "avg=" << sum/3.0 << endl;
    return 0;
}
```

例2：输入3个人的成绩，打印每个人的成绩及平均值

S中只保留了最后一次的输入

```
int main()
{   int i, s, sum = 0;
    for(i=0; i<3; i++) {
       cin >> s;  //假设输入都正确
       cout << "第" << i+1 << "个=" << s << endl;
       sum += s;
       }
    cout << "avg=" << sum/3.0 << endl;
    return 0;
}
```

例3：输入3个人的成绩，先打印平均值再打印成绩

输入/输出无法用循环，因为每次变量不同

```
int main()
{   int s1, s2, s3;
    cin >> s1;   //假设输入正确
    cin >> s2;   //假设输入正确
    cin >> s3;   //假设输入正确
    cout << "avg=" << (s1+s3+s3)/3.0 << endl;
    cout << "第1个:" << s1 << endl;
    cout << "第2个:" << s2 << endl;
    cout << "第3个:" << s3 << endl;
    return 0;
}
```

10000人 ？

人数不定 ？

§5. 数组

5.2. 一维数组的定义和引用

5.2.1. 定义

数据类型　数组名[正整型常量表达式]

```
int student[10];
long a[15*2];
```

★ 包括整型常量、整型符号常量和整型只读变量

```
#define N 10        const int n=10;
int a[N]; ✓         int a[n]; ✓

int k=10;
int a[k]; ×
```

● 说明：早期C/C++标准中，数组大小必须是常量，新标准已允许为变量，为统一，此处仍要求为常量

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "请输入数组大小" << endl;
    cin >> n;
    int a[n];
    return 0;
}
```

VS 编译：×
Dev编译：✓

```
已启动生成...
1>------ 已启动生成: 项目: demo-cpp, 配置: Debug Win32 ------
1>demo.cpp
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): error C2131: 表达式的计算结果不是常数
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): message : 因读取超过生命周期的变量而失败
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): message : 请参见"n"的用法
1>已完成生成项目"demo-cpp.vcxproj"的操作 - 失败。
========== 生成: 0 成功, 1 失败, 0 最新, 0 已跳过 ==========
```

```
编译结果...
--------
- 错误: 0
- 警告: 0
- 输出文件名: D:\WorkSpace\VS2022-demo\d
- 输出大小: 2.33393955230713 MiB
- 编译时间: 0.63s
```

D:\WorkSpace\VS2022-demo\demo-cpp\demo.exe

```
请输入数组大小
```

§5. 数组

5.2. 一维数组的定义和引用

5.2.2. 使用

★ 必须先定义，后使用，数组名的命名规则同变量名

★ 数组的大小在声明时应确定，不能动态定义，在内存中顺序存放，占用一块连续的空间

★ 若数组定义中整型常量表达式为n，表示有n个元素，则称数组长度应为n

★ 每个数组元素在内存中占用一个数据类型所占用的字节数，数组元素的表示方法为数组名[下标]，下标范围从[0..n-1]，表示为整型常量或整型表达式

```
#include <iostream>
using namespace std;
int main()
{  long a[10];
   cout << sizeof(a)     << endl;
   cout << sizeof(a[3]) << endl;
}
```

```
40
4
```

例：若有数组定义long a[10]
则：数组长度为10
数组下标表示范围[0..9]
每个数组元素占4个字节
sizeof(long)=4
数组共占用内存中连续的40个字节
内存存放为：

a[0]	2000 2003	
a[1]	2004 2007	
a[2]	2008 2011	
a[3]	2012 2015	
a[4]	2016 2019	
a[5]	2020 2023	
a[6]	2024 2027	
a[7]	2028 2031	
a[8]	2032 2035	
a[9]	2036 2039	

★ 即使允许使用变量定义数组，数组定义后大小仍为固定值，定义变量值的改变不影响数组

```
//用Dev编译运行
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;   //假设输入15
    int a[n];
    cout << sizeof(a) << endl;
    n = 10;
    cout << sizeof(a) << endl;
    return 0;
}
```

```
60

60
```

§5. 数组

5.2. 一维数组的定义和引用

5.2.1. 定义

5.2.2. 使用

★ 数组的使用只能逐个引用其中元素，不能整体使用，引用时数组下标的表示为常量或带变量的表达式

int a[10];	
a[0]=15;	✓
float c; long d; c+d*a[5];	✓
int k=3; a[k*4-3]=18;	✓
10+a[0]+a[1]-a[2];	✓

int a[5], i;	
× cout << a; printf("%d",a);	
× printf("%d%d%d%d%d",a);	
✓ cout<<a[0]<<a[1]<<a[2]<<a[3]<<a[4];	
for(i=0;i<5;i++)	
✓ cout << a[i];	

错误是指无法得到预期结果，编译本身没错，得到的是另外值(地址)

```
#include <iostream>
using namespace std;

int main()
{
    int a[10], i;
    for(i=0; i<10; i++)
        cout << a[i] << endl;
    cout << endl;
    cout << a << endl;

    return 0;
}
```

思考：将int a[10]改为全局变量，前10行输出分别是什么？

10个不确定值
一个6/8位的16进制数表示数组的首地址

Microsoft Visual Studio 调试控制台

```
-858993460
-858993460
-858993460
-858993460
-858993460
-858993460
-858993460
-858993460
-858993460
-858993460
0101F9F8
```

VS

D:\WorkSpace\VS2019-Demo\cpp-demo\cpp-demo.exe

```
-2
7929704
4199663
4242320
65535
0
0
47
8981760
7929704

0x78fe94
```

Dev

§5. 数组

5.2. 一维数组的定义和引用

5.2.1. 定义

5.2.2. 使用

★ 数组的使用只能逐个引用其中元素，不能整体使用，引用时数组下标的表示为常量或带变量的表达式

★ 引用时，若数组下标超出范围， C/C++不会报错，因此编程者要控制下标在合理的范围内

int a[10];	
3*5+a[10];	× （引用操作）
a[5+2*3]=16;	× （修改操作）

VS下均有warning
观察执行后的错误表现
其余编译器自行观察

```
#include <iostream>
using namespace std;
int main()
{
    int k, a[10];
    k = 10 + a[10] * 2;
    cout << k << endl;
    return 0;
}//能执行，不确定值
```

```
#include <iostream>
using namespace std;
int main()
{
    int k, a[10];
    k = 10+a[1000]*2;
    cout << k << endl;
    return 0;
} //不确定值 or 被系统终止
```

```
#include <iostream>
using namespace std;
int main()
{
    int a[10];
    a[10] = 123;
    cout<<a[10]<< endl;
    return 0;
}//输出123后被系统终止
```

```
#include <iostream>
using namespace std;
int main()
{
    int a[10];
    a[1000] = 123;
    cout <<a[1000]<< endl;
    return0;
}//直接被操作系统终止
```

注意：编译都不会报错，
是执行时错误
=> 数组下标越界即使执行时不错，
也是严重错误

§5. 数组

5.2. 一维数组的定义和引用

5.2.1. 定义

5.2.2. 使用

★ 数组的使用只能逐个引用其中元素，不能整体使用，引用时数组下标的表示为常量或带变量的表达式

★ 引用时，若数组下标超出范围， C/C++不会报错，因此编程者要控制下标在合理的范围内

int a[10];	
3*5+a[10];	× （引用操作）
a[5+2*3]=16;	× （修改操作）

```
#include <iostream>
using namespace std;

int main()
{
    int k=321, a[10];
    cout << "k=" << k << endl;
    a[12] = 123;
    cout << "a[12]=" << a[12] << endl;
    cout << "k=" << k << endl;
    return 0;
}
```

VS下：
a[12]实际占了k的位置，
但系统不报错，访问a[12]
也正确，但实际是严重错误
换为Dev，观察运行结果

VS有IntelliSense，
结果为：
k=321
a[12]=123
k=123

```
#include <iostream>
using namespace std;

int main()
{
    int k=321, a[10];
    cout << "k=" << k << endl;
    a[10] = 123;
    cout << "a[10]=" << a[10] << endl;
    cout << "k=" << k << endl;
    return 0;
}
```

用VS执行，观察运行结果

用Dev执行，观察运行结果

§5. 数组

5.2. 一维数组的定义和引用

5.2.3. 数组在定义时初始化

5.2.3.1. 初始化的作用

保证使用前有一个确定值

5.2.3.2. 全部初始化

A. int a[10]={1,2,3,4,5,6,7,8,9,10};

a[0]	2000 2003	1
a[1]	2004 2007	2
a[2]	2008 2011	3
a[3]	2012 2015	4
a[4]	2016 2019	5
a[5]	2020 2023	6
a[6]	2024 2027	7
a[7]	2028 2031	8
a[8]	2032 2035	9
a[9]	2036 2039	10

int a[10]={7,-3,9,64,76,-23,78,123,-76,263};

a[0]	2000 2003	7
a[1]	2004 2007	-3
a[2]	2008 2011	9
a[3]	2012 2015	64
a[4]	2016 2019	76
a[5]	2020 2023	-23
a[6]	2024 2027	78
a[7]	2028 2031	123
a[8]	2032 2035	-76
a[9]	2036 2039	263

int a[5]={1,2,3,4,5,6}

错误，因为初始化的个数超过了数组的大小

```
#include <iostream>
using namespace std;
int main()
{
    int a[5] = { 1,2,3,4,5,6 };
    return 0;
}
```

```
已启动生成…
1>------ 已启动生成: 项目: cpp-demo, 配置: Debug Win32 ------
1>cpp-demo.cpp
1>D:\WorkSpace\VS2019-Demo\cpp-demo\cpp-demo.cpp(5,25): error C2078: 初始值设定项太多
1>已完成生成项目"cpp-demo.vcxproj"的操作 - 失败。
========== 生成: 成功 0 个，失败 1 个，最新 0 个，跳过 0 个 ==========
```

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

数组长度自动定义为10（初始化元素的个数）

如何验证？ => sizeof(a) 期望值40

```
#include <iostream>
using namespace std;

int main()
{
    int a[] = { 0,1,2,3,4,5,6,7,8,9 };
    cout << sizeof(a) << endl;
    return 0;
}
```

Microsoft

40

§5. 数组

5.2. 一维数组的定义和引用

5.2.3. 数组在定义时初始化

5.2.3.1. 初始化的作用

5.2.3.2. 全部初始化

5.2.3.3. 部分初始化

```
int a[10]={5,-2, 7};
```

a[0]=5 a[1]=-2 a[2]=7

a[3]-a[9]自动为0

a[0]	2000 2003	5
a[1]	2004 2007	-2
a[2]	2008 2011	7
a[3]	2012 2015	0
a[4]	2016 2019	0
a[5]	2020 2023	0
a[6]	2024 2027	0
a[7]	2028 2031	0
a[8]	2032 2035	0
a[9]	2036 2039	0

```
int main()
{
    int i, a[10]={5,-2, 7 };
    for (i=0; i<10; i++)
        cout << a[i] << ' ';
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
5 -2 7 0 0 0 0 0 0 0
```

```
int a[1000]={0};    //全0
```

长度不可省略

```
#include <iostream>
using namespace std;
int main()
{
    int a[1000]={ 0 };
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[999] << endl;
    return 0;
}
```

Microsoft

```
0
0
0
```

```
int a[1000]={1};    //a[0]为1，其余全0
```

注意：不是全1

```
#include <iostream>
using namespace std;
int main()
{
    int a[1000]={ 1 };
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[999] << endl;
    return 0;
}
```

Microsoft

```
1
0
0
```

```
int a[1000];    希望a[0] - a[999] 为 0 - 999
```

方法1：`int a[1000] = {0,1,2,...,999};`　　语句太长，一般不用

方法2：

```
int a[1000], i;
for(i=0; i<1000; i++)
    a[i] = i;
```

不是定义时初始化，而是通过执行语句进行赋值操作

§5. 数组

5.2. 一维数组的定义和引用

5.2.3. 数组在定义时初始化

5.2.3.1. 初始化的作用

5.2.3.2. 全部初始化

5.2.3.3. 部分初始化

★ 若一个都不初始化，则数组元素的值，当数组为自动变量时：不确定

当数组为静态局部、静态全局、外部全局：0

```
int a[1000]; //全部为0
int main()
{   int b[100]; //不确定
}
```

```
#include <iostream>
using namespace std;
int a[1000];
int main()
{
    int b[100];
    cout << a[0] << ' ' << b[0] << endl;
    cout << a[9] << ' ' << b[9] << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0 -858993460
0 -858993460
```

§5. 数组

思考：
1、人数不定如何处理？
2、如何检查输入的合理性？

5.2. 一维数组的定义和引用

5.2.4. 应用

例3：输入3个人的成绩，先打印平均值再打印成绩

```
int main()
{   int s1, s2, s3;
    cin >> s1;   //假设输入正确
    cin >> s2;   //假设输入正确
    cin >> s3;   //假设输入正确
    cout << "avg=" << (s1+s3+s3)/3.0 << endl;
    cout << "第1个:" << s1 << endl;
    cout << "第2个:" << s2 << endl;
    cout << "第3个:" << s3 << endl;
    return 0;
}
```

输入/输出无法用循环，因为每次变量不同

例3：输入3个人的成绩，先打印平均值再打印成绩

```
int main()
{  int s[3], i, sum=0;
   for (i=0; i<3; i++) {
      cin >> s[i];  //假设输入正确
      sum += s[i];
      }
   cout << "avg=" << sum/3.0 << endl;
   for (i=0; i<3; i++)
      cout <<"第"<< i+1 << "个:" << s[i] << endl;
   return 0;
}
```

循环方式

例3：输入3个人的成绩，先打印平均值再打印成绩

```
int main()
{  int s[10000], i, sum=0;
   for (i=0; i<10000; i++) {
      cin >> s[i];  //假设输入正确
      sum += s[i];
      }
   cout << "avg=" << sum/10000.0 << endl;
   for (i=0; i<10000; i++)
      cout <<"第"<< i+1 << "个:" << s[i] << endl;
   return 0;
}
```

10000人

例3：输入3个人的成绩，先打印平均值再打印成绩

```
#define NUM 10000  /  const int NUM=10000;
int main()
{  int s[NUM], i, sum=0;
   for (i=0; i<NUM; i++) {
      cin >> s[i];  //假设输入正确
      sum += s[i];
      }
   cout << "avg=" << sum/float(NUM) << endl;
   for (i=0; i<NUM; i++)
      cout <<"第"<< i+1 << "个:" << s[i] << endl;
   return 0;
}
```

10000人，更合理方便修改及维护

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：求斐波那契数列

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    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 << endl;
        cout << setw(8) << f[i];
        }
    cout << endl;
    return 0;
}
```

和前例相比

1、前例：边计算边输出
　　本例：先计算后输出

2、前例：输出后值不保留，无法回溯
　　本例：值全部保留，可回溯

空行

1	1	2	3	5
8	13	21	34	55
89	144	233	377	610
987	1597	2584	4181	6765

Microsoft Visual Studio 调试控制台

1	1	2	3	5
8	13	21	34	55
89	144	233	377	610
987	1597	2584	4181	6765

思考：
如果不希望空第一行，
如何实现？

§3. 结构化程序设计基础

例：求 Fibonacci 数列的前40项

本程序的不足之处：无法回溯
（当f1表示第7个月后，第5个月的值无法再现）

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int i, f1=1, f2=1;    //初值
    for(i=1; i<=20; i++) { //每次2数，20次=40个
            cout << setw(12) << f1 << setw(12) << f2;
      if (i%2==0)
         cout << endl; //每2次(4个)加换行
      f1 = f1 + f2;     //f1为第3/5/7/...个月
      f2 = f1 + f2;     //f2为第4/6/8/...个月
      }
    return 0;
}
```

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

内外循环的关系：

外循环i=0，内循环j=0~8，比较9次

外循环i=1，内循环j=0~7，比较8次

...

...

外循环i=8，内循环j=0~0，比较1次

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环 0 1 2 3 4 5 6 7 8

0 3
1 7
2 29
3 23
4 12
5 82
6 1
7 72
8 8
9 5

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-							
2	29								
3	23								
4	12								
5	82								
6	1								
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-						
3	23								
4	12								
5	82								
6	1								
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29					
4	12								
5	82								
6	1								
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29				
5	82								
6	1								
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29	-			
5	82					-			
6	1								
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29	-			
5	82					-	1		
6	1						82		
7	72								
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29	-			
5	82					-	1		
6	1						82	72	
7	72							82	
8	8								
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29	-			
5	82					-	1		
6	1						82	72	
7	72							82	8
8	8								82
9	5								

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3	-							
1	7	-	-						
2	29		-	23					
3	23			29	12				
4	12				29	-			
5	82					-	1		
6	1						82	72	
7	72							82	8
8	8								82 5
9	5								82

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8	
0	3	-								3
1	7	-	-							7
2	29		-	23						23
3	23			29	12					12
4	12				29	-				29
5	82					-	1			1
6	1						82	72		72
7	72							82	8	8
8	8								82	5
9	5									82

第1次外循环后，最大数82在最后

§5. 数组

5.2. 一维数组的定义和引用

5.2.4. 应用

例：冒泡法排序

```
#include <iostream>
using namespace std;
#define  N  10
int main()
{   int a[N] = { 3,7,29,23,12,82,1,72,8,5 };
    int i, j, t;

    for (i=0; i<N-1; i++)
        for (j=0; j<N-(i+1); j++)
            if (a[j] > a[j+1]) {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
                }

    for (i=0; i<N; i++)
        cout << a[i] << ' ';
    cout << endl;
}
```

两数交换方法

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0

内循环	0	1	2	3	4	5	6	7	8
0	3								
1	7								
2	23								
3	12								
4	29								
5	1								
6	72								
7	8								
8	5								
9	82								

第1次外循环后，最大数82在最后，

第2次外循环就不必再比较该数了

§5. 数组

5.3. 二维数组的定义和引用

5.3.1. 定义

数据类型 数组名[正整型常量表达式1][正整型常量表达式2]

行 列

```
int a[5][7];
long s[2*8][9];
```

★ 对应为一张二维表

★ 包括整型常量、整型符号常量和整型只读变量

```
#define M 5          const int m=5,n=10;
#define N 10         int a[m][n]; ✓
int a[M][N] ✓

int i=5,j=10;
int a[i][j]; ×
```

● 说明：早期C/C++标准中，数组大小必须是常量，新标准已允许为变量，为统一，此处仍要求为常量

```
#include <iostream>
using namespace std;
int main()
{
    int m, n;
    cout << "请输入数组大小" << endl;
    cin >> m >> n;
    int a[m][n];
    return 0;
}
```

同一维数组

VS 编译：×
Dev编译：✓

```
已启动生成...
1>------ 已启动生成: 项目: demo-cpp, 配置: Debug Win32 ------
1>demo.cpp
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): error C2131: 表达式的计算结果不是常数
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): message : 因读取超过生命周期的变量而失败
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,11): message : 请参见“m”的用法
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,14): error C2131: 表达式的计算结果不是常数
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,14): message : 因读取超过生命周期的变量而失败
1>D:\WorkSpace\VS2022-demo\demo-cpp\demo.cpp(8,14): message : 请参见“n”的用法
1>已完成生成项目“demo-cpp.vcxproj”的操作 - 失败。
========== 生成: 0 成功，1 失败，0 最新，0 已跳过 ==========
```

```
编译结果...
--------
- 错误: 0
- 警告: 0
- 输出文件名: D:\WorkSpace\VS2022-demo\
- 输出大小: 2.33393955230713 MiB
- 编译时间: 0.61s
```

```
D:\WorkSpace\VS2022-demo\demo-cpp\demo.exe
请输入数组大小
```

§5. 数组

5.3. 二维数组的定义和引用

5.3.1. 定义

5.3.2. 使用

★ 必须先定义，后使用，数组名的命名规则同变量名

★ 数组的大小在声明时应确定，不能动态定义，在内存中存放时先行后列，占用一块连续的空间

★ 若数组定义中整型常量表达式为m,n，则表示有m*n个元素，数组长度应为m*n，也称数组有m行n列

★ 每个数组元素占用一个数据类型所占用的字节数，数组元素的表示方法为数组名[行下标][列下标]，行下标的范围从[0..m-1]，列下标的范围从[0..n-1]，表示为整型常量或整型表达式

```
#include <iostream>
using namespace std;
int main()
{
    int a[3][4];
    cout << sizeof(a) << endl;
    cout << sizeof(a[1][2]) << endl;
    return 0;
}
```

```
48
4
```

例：有数组定义int a[3][4]，则
数组有3行4列，元素排列为：

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]

共有3*4=12个数组元素，数组长度为12
每个数组元素占4个字节
整个数组占用3*4*sizeof(int)=48个字节
行下标的范围：0-2
列下标的范围：0-3
内存存放为：

a[0][0]	2000 2003	
a[0][1]	2004 2007	
a[0][2]	2008 2011	
a[0][3]	2012 2015	
a[1][0]	2016 2019	
a[1][1]	2020 2023	
a[1][2]	2024 2027	
a[1][3]	2028 2031	
a[2][0]	2032 2035	
a[2][1]	2036 2039	
a[2][2]	2040 2043	
a[2][3]	2044 2047	

§5. 数组

5.3. 二维数组的定义和引用

5.3.1. 定义

5.3.2. 使用

★ 二维数组可以看作是一个一维数组，它的每个数组元素都是一个一维数组

二维数组 int a[3][4],
理解为一维数组,有3(行)个元素
每个元素又是一维数组,有4(列)个元素
a是二维数组名
a[0],a[1],a[2]是一维数组名

理解1：a [3][4]

理解2：a[3] [4]

```
#include <iostream>
using namespace std;
int main()
{
    int a[3][4];
    cout << sizeof(a) << endl;
    cout << sizeof(a[0]) << endl;
    cout << sizeof(a[1][2]) << endl;
    return 0;
}
```

```
48
16
4
```

a[0]	a[0][0]	2000 2003	
	a[0][1]	2004 2007	
	a[0][2]	2008 2011	
	a[0][3]	2012 2015	
a[1]	a[1][0]	2016 2019	
	a[1][1]	2020 2023	
	a[1][2]	2024 2027	
	a[1][3]	2028 2031	
a[2]	a[2][0]	2032 2035	
	a[2][1]	2036 2039	
	a[2][2]	2040 2043	
	a[2][3]	2044 2047	

§5. 数组

5.3. 二维数组的定义和引用

5.3.1. 定义

5.3.2. 使用

★ 数组的使用只能逐个引用其中元素，不能整体使用，引用时数组下标的表示为常量或带变量的表达式

int a[10][20];	
a[0][7]=15;	✓
float c; long d; c+d*a[5][3];	✓
int k=3; a[k*4-3][k/2+4]=18;	✓
10+a[0][9]+a[1][3]-a[2][2];	✓

int a[3][4], i, j;	
×	cout << a;
×	cout << a[0];
✓	cout << a[0][0] << a[1][2] << a[2][3];
	for(i=0;i<3;i++) for(j=0;j<4;j++)
✓	cout << a[i][j];

错误是指无法得到预期结果，编译本身没错，得到的是另外值（数组的首地址）

★ 引用时，若数组下标超出范围，C/C++不会报错，因此编程者要控制下标在合理的范围内

int a[10][20];	
3*5+a[10][5];	× （引用操作）
3*5+a[5][20];	× （引用操作）
a[5*3][21]=16;	× （修改操作）

注意：编译都不会报错，是执行时错误

=> 数组下标越界即使执行时不错，也是严重错误

§5. 数组

5.3. 二维数组的定义和引用

5.3.3. 数组在定义时初始化

5.3.3.1. 全部初始化

A. int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

a[0][0]=1 a[0][1]=2 a[0][2]=3 a[0][3]=4
a[1][0]=5 a[1][1]=6 a[1][2]=7 a[1][3]=8
a[2][0]=9 a[2][1]=10 a[2][2]=11 a[2][3]=12

```
#include <iostream>
using namespace std;
int main()
{
    int i,j;
    int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
    for (i=0; i<3; i++) {
        for (j=0; j<4; j++)
            cout << a[i][j] << ' ';
        cout << endl;
        }
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4
5 6 7 8
9 10 11 12
```

a[0][0]	2000 2003	1
a[0][1]	2004 2007	2
a[0][2]	2009 2011	3
a[0][3]	2012 2015	4
a[1][0]	2016 2019	5
a[1][1]	2020 2023	6
a[1][2]	2024 2027	7
a[1][3]	2028 2031	8
a[2][0]	2032 2035	9
a[2][1]	2036 2039	10
a[2][2]	2040 2043	11
a[2][3]	2044 2047	12

int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12,13};

错误，元素的个数超过了数组的大小

cpp-demo.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[3][4]= { 1,2,3,4,5,6,7,8,9,10,11,12,13 };
6      return 0;
7  }
```

注意：波浪号的位置是第13个

error C2078: 初始值设定项太多

§5. 数组

5.3. 二维数组的定义和引用

5.3.3. 数组在定义时初始化

5.3.3.1. 全部初始化

B. int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

⇕

```
int a[3][4]={{1,  2,  3,  4},
             {5,  6,  7,  8},
             {9, 10, 11, 12}};
```

更直观

1	2	3	4
5	6	7	8
9	10	11	12

★ 如果采用双层括号方式，则内括号中元素的个数不超过列数，成对的内括号的总数不超过行数

int a[3][4]={{1,2,3,4,5},{5,6,7,8},{9,10,11,12}};

错误，元素的总个数超过了数组大小

int a[3][4]={{1,2,3,4,5},{6,7,8},{9,10,11,12}};

错误，元素的总个数虽然正确，但一行的个数超过了数组定义的行大小

§5. 数组

5.3. 二维数组的定义和引用

5.3.3. 数组在定义时初始化

5.3.3.1. 全部初始化

A. int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};

B. int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

C. int a[][4]={1,2,3,4,5,6,7,8,9,10,11,12};

(行缺省=3)(列不可省略)

×int a[3][]={1,2,3,4,5,6,7,8,9,10,11,12};

不允许缺省列

D. int a[][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

(行缺省=3)(列不可省略)

×int a[3][]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

不允许缺省列

1	2	3	4
5	6	7	8
9	10	11	12

```
#include <iostream>
using namespace std;
int main()
{
    int a[3][] = { 1,2,3,4,5,6,7,8,9,10,11,12 };
    return 0;
}
```

(5,16): error C2087: "a": 缺少下标
(5,26): error C2078: 初始值设定项太多

```
#include <iostream>
using namespace std;
int main()
{
    int a[3][] = { {1,2,3,4},{5,6,7,8},{9,10,11,12} };
    return 0;
}
```

(5,16): error C2087: "a": 缺少下标
(5,23): error C2078: 初始值设定项太多

问：为什么只能行缺省而不能列缺省？

答：二维表形式：{ 行缺省，行 = 总数/列
列缺省，列 = 总数/行 } 无法理解

元素是一维数组的一维数组：

行缺省：元素大小(一维数组)已知，省略个数

列缺省：元素个数已知，省略大小 ×

§5. 数组

5.3. 二维数组的定义和引用

5.3.3. 数组在定义时初始化

5.3.3.1. 全部初始化

5.3.3.2. 部分初始化

A. int a[3][4]={1,5,9};

a[0][0]=1 a[0][1]=5 a[0][2]=9，其余为0

1	5	9	0
0	0	0	0
0	0	0	0

B. int a[3][4]={{1},{5},{9}};

a[0][0]=1 a[1][0]=5 a[2][0]=9，其余为0

1	0	0	0
5	0	0	0
9	0	0	0

C. int a[3][4]={{1,2},{5,6,7},{9,10}};

1	2	0	0
5	6	7	0
9	10	0	0

D. int a[3][4]={{1},{5,6}}

1	0	0	0
5	6	0	0
0	0	0	0

E. int a[3][4]={{1},{},{5,6}};

1	0	0	0
0	0	0	0
5	6	0	0

§5. 数组

5.3. 二维数组的定义和引用

5.3.3. 数组在定义时初始化

5.3.3.2. 部分初始化

F. int a[][4]={{1},{5},{9}};

3(省略)

G. int a[][4]={{1},{5}};

2(省略)

H. int a[][4]={{1},{},{5}};

3(省略)

I. int a[][4]={1,2,3,4,5,6,7,8,9};

3(省略)

3 = 「初始化元素个数/列数」

J. int a[][4]={1,2,3,4,5};

2(省略) = 「初始化元素个数/列数」

如何验证？

```
#include <iostream>
using namespace std;
int main()
{
    int a1[][4]={{1}, {5}, {9}};      //3
    int a2[][4]={{1}, {5}};           //2
    int a3[][4]={{1}, {}, {5}};       //3
    int a4[][4]={1,2,3,4,5,6,7,8,9}; //3
    int a5[][4]={1,2,3,4,5};          //2

    cout << sizeof(a1) << endl;   //期望48
    cout << sizeof(a2) << endl;   //期望32
    cout << sizeof(a3) << endl;   //期望48
    cout << sizeof(a4) << endl;   //期望48
    cout << sizeof(a5) << endl;   //期望32
    return 0;
}
```

§5. 数组

5.3. 二维数组的定义和引用

5.3.4. 应用

例：矩阵转置（2x3 转置为 3x2）

```
#include <iostream>
using namespace std;

int main()
{
    int a[2][3] = { {1,2,3},{4,5,6} };
    int b[3][2], i, j;

    cout << "Matrix a:" << endl;
    for (i=0; i<2; i++) {
        for (j=0; j<3; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
    for (i=0; i<2; i++)
        for (j=0; j<3; j++)
            b[j][i] = a[i][j];  //转置

    cout << "Matrix b:" << endl;
    for (i=0; i<3; i++) {
        for (j=0; j<2; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }

    return 0;
}
```

```
#include <iostream>
using namespace std;
const int M=2, N=3;
int main()
{
    int a[M][N] = { {1,2,3},{4,5,6} };
    int b[N][M], i, j;

    cout << "Matrix a:" << endl;
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            b[j][i] = a[i][j];  //转置

    cout << "Matrix b:" << endl;
    for (i=0; i<N; i++) {
        for (j=0; j<M; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Matrix a:
1 2 3
4 5 6
Matrix b:
1 4
2 5
3 6
```

1. 用宏定义/常变量表示2/3更好
2. C/C++中的数组范围表示，一般遵循前闭后开的规则

   for(i=0; i<=1; i++)

   for(i=0; i<2; i++) √

§5. 数组

5.3. 二维数组的定义和引用

5.3.4. 应用

例：求最大值及所在行列(如有相同，按先行后列取第一个)

```
#include <iostream>
using namespace std;

const int M=3, N=4;
int main()
{   int a[M][N] = { {7, -3, 12, 19},
                    {-23, 17, 92, 78} ,
                    {2, -1, 5, 92} };
    int i, j;
    int max=a[0][0], row=0, col=0;//初始认为最大值为a[0][0]

    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            if (a[i][j] > max) {
                max = a[i][j]; //替换新的max
                row = i;       //同时记录下行、列值
                col = j;
            }

    cout << "max=" << max << endl;
    cout << "row=" << row << endl;
    cout << "col=" << col << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
max=92
row=1
col=2
```

例：求最大值及所在行列(如有相同，按先行后列取最后一个)

```
#include <iostream>
using namespace std;

const int M=3, N=4;
int main()
{   int a[M][N] = { {7, -3, 12, 19},
                    {-23, 17, 92, 78} ,
                    {2, -1, 5, 92} };
    int i, j;
    int max=a[0][0], row=0, col=0;//初始认为最大值为a[0][0]

    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            if (a[i][j] >= max) {
                max = a[i][j]; //替换新的max
                row = i;       //同时记录下行、列值
                col = j;
            }

    cout << "max=" << max << endl;
    cout << "row=" << row << endl;
    cout << "col=" << col << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
max=92
row=2
col=3
```

§5. 数组

5.3. 二维数组的定义和引用

补：多维数组的基本概念

★ 定义：数据类型 数组名[N1][..][..][..][..]

★ 内存中的排列：离数组名最远的维数变化最快，最近的维数变化最慢

★ 一般理解：三维/四维/…/N维空间

★ 专业理解：N维数组是元素是N-1维数组的一维数组

=> N维数组是基本元素的类型为数据类型的一维数组的一维数组的...一维数组

★ 定义时初始化：允许N层括号嵌套，每层括号内的元素数量受各层对应维数的限制，只能省略最靠近数组名的维数大小

```
int a[3][4][5][6][7];

内存：a[0][0][0][0][0],…,a1[0][0][0][0][6]
      a[0][0][0][1][0],…,a1[0][0][0][1][6]
                 …                    …
      a[0][0][0][5][0],…,a1[0][0][0][5][6]
                 …                    …
      a[2][3][4][5][0],…,a1[2][3][4][5][6]

定义时初始化：
int a[3][4][5][6][7] =  {{},{},{}};
                     => {{{},{},{},{}},…}
                     => {{{{},{},{},{},{}},…},…}
                     => {{{{{},{},{},{},{},{}},…},…},…}
                     => {{{{{2,6,1,3,9,8,12},…},…},…},…}
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

★ 形参为相应类型的简单变量

例：数组元素做参数求最大值及所在行列

同前例，将3/4换为M/N更合理

例：求最大值（如果最大值有相同，按先行后列取第一个）

```
int max_value(int x, int max)
{
    if (x > max)
        return x;
    else
        return max;
}

int main()
{
    int a[3][4] = { {7, -3, 12, 19},
                    {-23, 17, 92, 78},
                    {2, -1, 5, 92} };
    int i, j, max=a[0][0]; //初始最大值为a[0][0]

    for(i=0; i<3; i++)
       for(j=0; j<4; j++)
          max=max_value(a[i][j], max);

    cout << "max=" << max << endl;

    return 0;
}
```

实参:二维数组元素(int型)
形参:int型简单变量

例：求最大值及所在行列（如果最大值有相同，按先行后列取第一个）

```
int main()
{
    int a[3][4] = { {7, -3, 12, 19},
                    {-23, 17, 92, 78},
                    {2, -1, 5, 92} };
    int i, j, max=a[0][0], row=0, col=0; //初始最大值为a[0][0]
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            if (a[i][j] > max) {
                max = a[i][j]; //替换新的max
                row = i;       //同时记录行/列值
                col = j;
            }
    cout << "max=" << max << endl;
    cout << "row=" << row << endl;
    cout << "col=" << col << endl;
    return 0;
}
```

例：求最大值（如果最大值有相同，按先行后列取第一个）

```
int main()
{
    int a[3][4] = { {7, -3, 12, 19},
                    {-23, 17, 92, 78},
                    {2, -1, 5, 92} };
    int i, j, max=a[0][0]; //初始最大值为a[0][0]
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            if (a[i][j] > max) {
                max = a[i][j]; //替换新的max

            }
    cout << "max=" << max << endl;
    return 0;
}
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i, j, k, t;
    for (i=0; i<n-1; i++) {
        k = i;
        for (j=i+1; j<n; j++)
            if (array[j] < array[k])
                k = j;
       t = array[k];
       array[k] = array[i];
       array[i] = t;
       }
}

int main()
{
    const int N = 10;
    int a[N]={3,7,29,23,12,82,1,72,8,5};
    cout << "排序前：";
    for (i = 0; i < N; i++)
        cout << a[i] << ' ';
    cout << endl;

    select_sort(a, N);

    cout << "排序后：";
    for (i = 0; i < N; i++)
        cout << a[i] << ' ';
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
排序前：3 7 29 23 12 82 1 72 8 5
排序后：1 3 5 7 8 12 23 29 72 82
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

0	1	2	3	4	5	6	7	8	9
3	7	29	23	12	82	1	72	8	5

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2										
3										
4										
5										
6										
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3										
4										
5										
6										
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4										
5										
6										
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4	12	<	3	-						
5										
6										
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4	12	<	3	-						
5	82	<	3	-						
6										
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4	12	<	3	-						
5	82	<	3	-						
6	1	<	3	k=6						
7										
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4	12	<	3	-						
5	82	<	3	-						
6	1	<	3	k=6						
7	72	<	1	-						
8										
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	<	3	-						
2	29	<	3	-						
3	23	<	3	-						
4	12	<	3	-						
5	82	<	3	-						
6	1	<	3	k=6						
7	72	<	1	-						
8	8	<	1	-						
9										

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5

1	7	<	3	-
2	29	<	3	-
3	23	<	3	-
4	12	<	3	-
5	82	<	3	-
6	1	<	3	k=6
7	72	<	1	-
8	8	<	1	-
9	5	<	1	-

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0　k=0　内循环 1-9

	0	1	2	3	4	5	6	7	8	9
	3	7	29	23	12	82	1	72	8	5
1	7	< 3	–							
2	29	< 3	–							
3	23	< 3	–							
4	12	< 3	–							
5	82	< 3	–							
6	1	< 3	k=6							
7	72	< 1	–							
8	8	< 1	–							
9	5	< 1	–							

循环结束 k=6　a[0] ⇔ a[6]

1	7	29	23	12	82	3	72	8	5

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

例：选择法排序

```
void select_sort(int array[], int n)
{
    int i,j,k,t;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (array[j] < array[k])
                k=j;
        t=array[k];
        array[k]=array[i];
        array[i]=t;
        }
}

int main() { ... select_sort(a, N); ...};
```

实参:一维数组名
形参:一维数组

设a[10]为{3,7,29,23,12,82,1,72,8,5}

第1次，外循环i=0 k=0 内循环 1-9

0	1	2	3	4	5	6	7	8	9
3	7	29	23	12	82	1	72	8	5

内循环结束 a[0] ⇔ a[6]

1	7	29	23	12	82	3	72	8	5

第1次外循环结束，a[0]中已经是最小数

第2次:k=1 内循环:2-9 结束后a[1]次小

N个数，外循环N-1次即可

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

这些内容与第04模块中简单变量做实形参的概念不同，第06模块指针中再详细解释

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

★ 形参数组值的改变会影响到实参(与简单参数不同)

★ 因为形参数组不分配空间，因此数组大小可不指定

★ 因为形参数组不分配空间，因此实形参类型必须完全相同，否则编译错

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合

(实参占用空间，形参不占用空间)

```
//数组名/普通变量做函数参数，验证实形参地址是否相同
#include <iostream>
using namespace std;
void fun(int x[], int y)
{    cout << "x_size=" << sizeof(x) << endl;
     cout << "addr_x=" << x << endl; //数组名代表首地址
     cout << "addr_y=" << &y << endl; //普通变量加&
     cout << "x[2]=" << x[2] << endl;
}
int main()
{    int a[10] = {7, -2, 108, 25}, w=19;
     cout << "a_size=" << sizeof(a) << endl;
     cout << "addr_a=" << a << endl; //数组名代表首地址
     cout << "addr_w=" << &w << endl;//普通变量加&
     cout << "a[2]=" << a[2] << endl;
     fun(a, w);
}
```

```
a_size=40
addr_a=16进制地址a
addr_w=16进制地址w
a[2]=108
x_size=4
addr_x=16进制地址(与a相同)
addr_y=16进制地址(与w不同)
x[2]=108
```

说明

1. 证明了形参未分配40字节的数组空间
   (为什么是4第6章会解释)
2. 证明了实/形参内存地址重合
   (地址相同/[2]的值相同)

Microsoft Visual Studio 调试控制台

```
a_size=40
addr_a=0039F6B8
addr_w=0039F6AC
a[2]=108
x_size=4
addr_x=0039F6B8
addr_y=0039F5D8
x[2]=108
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

★ 形参数组值的改变会影响到实参(与简单参数不同)

```
//数组名及简单变量做函数参数，验证形参是否影响实参
#include <iostream>
using namespace std;
void fun(int x[], int y)
{    x[2]=37; //修改形参数组某元素的值
     y=45;    //修改简单变量形参的值
}
int main()
{    int a[10] = {7, -2, 18, 25}, w=19;
     cout << a[2] << ' ' << w << endl;
     fun(a, w);
     cout << a[2] << ' ' << w << endl;
}
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

★ 形参数组值的改变会影响到实参(与简单参数不同)

★ 因为形参数组不分配空间，因此数组大小可不指定

```
//形参数组不指定大小/指定大小与实参数组相同/不同，验证sizeof的值是否相同
#include <iostream>
using namespace std;
void f1(int x1[])      //形参数组不指定大小
{   cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[10])    //形参数组大小与实参相同
{   cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[1234]) //形参数组大小与实参不同
{   cout << "x3_size=" << sizeof(x3) << endl;
}
int main()
{   int a[10];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);     f2(a);     f3(a);
}
```

Microsoft Visual Studio 调试控制台

```
a_size=40
x1_size=4
x2_size=4
x3_size=4
```

(为什么是4后续会解释)

```
void select_sort(int array[10], int n);
void select_sort(int array[], int n);
void select_sort(int array[5],int n);
```

形参数组 { 和实参数组大小相同 / 不指定大小 / 和实参数组大小不同 } 均认为是正确的

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

★ 形参为相应类型的一维数组

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

★ 形参数组值的改变会影响到实参(与简单参数不同)

★ 因为形参数组不分配空间，因此数组大小可不指定

★ 因为形参数组不分配空间，因此实形参类型必须完全相同，否则编译错

```
#include <iostream>
using namespace std;
void f1(short b[])
{   return ;
}
void f2(long b[])
{   return ;
}
void f3(unsigned int b[])
{   return ;
}
int main()
{   int a[10];
    f1(a); //编译报错
    f2(a); //编译报错
    f3(a); //编译报错
}
```

```
error C2664: "void f1(short [])": 无法将参数 1 从"int [10]"转换为"short []"
 message : 与指向的类型无关；强制转换要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
message : 参见"f1"的声明
 error C2664: "void f2(long [])": 无法将参数 1 从"int [10]"转换为"long []"
 message : 与指向的类型无关；强制转换要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
message : 参见"f2"的声明
 error C2664: "void f3(unsigned int [])": 无法将参数 1 从"int [10]"转换为"unsigned int []"
 message : 与指向的类型无关；强制转换要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
 message : 参见"f3"的声明
```

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

5.4.3. 用多维数组名做函数实参

★ 形参为相应类型的多维数组

```
例：求最大值（如果最大值有相同，按先行后列取第一个）
int max_value(int array[][4])
{
    int i, j, max=array[0][0]; //初始认为最大值为a[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;
}
int main()
{   int a[3][4] = { {7, -3, 12, 19}, {-23, 17, 92, 78} ,{2, -1, 5, 92} };
    int max = max_value(a);
    cout << "max=" << max <<endl;
}
```

实参:二维数组名
形参:二维数组

★ 实、形参数组的列必须相等，形参的行可以不指定，或为任意值

```
int f(int x[3][4]);
int f(int x[][4]);
int f(int x[8][4]);//不推荐
int main() { int a[3][4];  f(a);  }
```

三种方式都正确

```
//形参数组不指定大小/指定大小与实参数组相同/不同，
//验证sizeof的值是否相同
#include <iostream>
using namespace std;
void f1(int x1[][4])     //形参数组不指定行大小
{
    cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[3][4])    //形参数组行大小与实参相同
{
    cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{
    cout << "x3_size=" << sizeof(x3) << endl;
}
int main()
{
    int a[3][4];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);
    f2(a);
    f3(a);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
a_size=48
x1_size=4
x2_size=4
x3_size=4
```

（为什么是4后续会解释）

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

5.4.3. 用多维数组名做函数实参

★ 形参为相应类型的多维数组

```
例：求最大值（如果最大值有相同，按先行后列取第一个）
int max_value(int array[][4])
{
    int i, j, max=array[0][0]; //初始认为最大值为a[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;
}
int main()
{   int a[3][4] = { {7, -3, 12, 19}, {-23, 17, 92, 78} ,{2, -1, 5, 92} };
    int max = max_value(a);
    cout << "max=" << max <<endl;
}
```

实参:二维数组名
形参:二维数组

★ 实、形参数组的列必须相等，形参的行可以不指定，或为任意值

```
int f(int x[3][4]);
int f(int x[][4]);
int f(int x[8][4]);//不推荐
int main() { int a[3][4];  f(a);  }
```

三种方式都正确

问：如何用一维数组的知识推导出本结论？
（如何做到知识的融会贯通？）

答：=> 一维数组做参数要求实形参元素类型完全一致
=> 二维数组理解为元素是一维数组的一维数组
=> 元素类型完全一致
=> 做元素的一维数组完全一致
=> 实、形参数组的列必须相等
=> 形参是一维数组时，数组大小可以不指定
=> 形参的行可以不指定

§5. 数组

5.5. 字符数组

第02模块中的概念：
1、char/unsigned char定义字符变量，代表一个字符
2、一对单引号可表示字符常量,可以字符/转义符形式
'A'　'\n'　'\x41'　'\101'
3、字符变量/常量在内存中占一个字节,存储为该字符的ASCII码，可当做1字节整数与整数通用
4、一对双引号可表示字符串常量，字符串中字符的个数称为字符串的长度，字符串的存储形式为每个字符的ASCII码+'\0'(尾零)
5、C++中无字符串变量，可用一维字符数组来表示字符串变量
6、暂不讨论字符串中含'\0'的情况("abc\0def")

5.5.1. 含义

数据类型为字符型的数组

5.5.2. 定义

char 数组名[正整型常量表达式]　　一维数组
unsigned char 数组名[正整型常量表达式]

char 数组名[正整型常量表达式1][正整型常量表达式2]　　二维数组
unsigned char 数组名[正整型常量表达式1][正整型常量表达式2]

★ 包括整型常量、整型符号常量和整型只读变量(部分编译器允许用变量定义数组，不讨论)

§5. 数组

5.5. 字符数组

5.5.3. 字符数组的初始化

5.5.3.1. 全部初始化

char a[5]={'c','h','i','n','a'};　　字符常量形式

⇕

char a[5]={99, 104, 105, 110, 97};　　整数形式(十进制)

char a[5]={'\143','\150','\151','\156','\141'};　8进制转义符形式

char a[5]={'\x63','\x68','\x69','\x6e','\x61'};　16进制转义符形式

char a[5]={0x63, 104, 105, 110, 0141};　　整数形式(多进制)

多种表示方法等价，内存占5字节，表示如下：

0110 0011	0110 1000	0110 1001	0110 1110	0110 0001

unsigned char a[4]={'c','h','i','n','a'};

若个数多，则错误

```
cpp-demo.cpp
cpp-demo
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      unsigned char a[4] = { 'c','h','i','n','a' };
6      return 0;
7  }
(5,44): error C2078: 初始值设定项太多
```

char a[]={'c','h','i','n','a'};

缺省为5

```
cpp-demo.cpp
cpp-demo
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char a[] = { 'c','h','i','n','a' };
6      cout << sizeof(a) << endl;
7      return 0;
8  }
Microsoft
5
```

```
int main()
{
    char a1[5] = { 'c','h','i','n','a' };
    char a2[5] = { 99, 104, 105, 110, 97 };
    char a3[5] = { '\143','\150','\151','\156','\141' };
    char a4[5] = { '\x63','\x68','\x69','\x6e','\x61' };
    char a5[5] = { 0x63, 104, 105, 110, 0141 };

    for (int i = 0; i < 5; i++) {
        cout << a1[i] << ' ';
        cout << a2[i] << ' ';
        cout << a3[i] << ' ';
        cout << a4[i] << ' ';
        cout << a5[i] << endl;
        }
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
c c c c c
h h h h h
i i i i i
n n n n n
a a a a a
```

```
int main()
{
    char a1[5] = { 'c','h','i','n','a' };
    char a2[5] = { 99, 104, 105, 110, 97 };
    char a3[5] = { '\143','\150','\151','\156','\141' };
    char a4[5] = { '\x63','\x68','\x69','\x6e','\x61' };
    char a5[5] = { 0x63, 104, 105, 110, 0141 };
    for (int i = 0; i < 5; i++) {
        cout << int(a1[i]) << ' ';
        cout << int(a2[i]) << ' ';
        cout << int(a3[i]) << ' ';
        cout << int(a4[i]) << ' ';
        cout << int(a5[i]) << endl;
        }
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
99 99 99 99 99
104 104 104 104 104
105 105 105 105 105
110 110 110 110 110
97 97 97 97 97
```

§5. 数组

5.5. 字符数组

5.5.3. 字符数组的初始化

5.5.3.1. 全部初始化

```
char b[3][3]={'a','b','c','d','e','f','g','h','i'};
```

a	b	c
d	e	f
g	h	i

```
char b[][3]={'a','b','c','d','e','f','g','h','i'};
```

缺省为3

```
char b[3][3]={{'a','b','c'},{'d','e','f'},{'g','h','i'}};

char b[][3]= {{'a','b','c'},{'d','e','f'},{'g','h','i'}};
```

缺省为3

§5. 数组

5.5. 字符数组

5.5.3. 字符数组的初始化

5.5.3.1. 全部初始化

5.5.3.2. 部分初始化

```
char a[10]= {'c','h','i','n','a'};
```

```
char a[10]={99, 104, 105, 110, 97};
```

对应a[0]-a[4]　　a[5]-a[9]为'\0'

0	99
1	104
2	105
3	110
4	97
5	0
6	0
7	0
8	0
9	0

```
char a[3][3]= {'a','b','c'};
```

a	b	c
\0	\0	\0
\0	\0	\0

```
char a[3][3]= {{'a'},{'b'},{'c'}};
```

a	\0	\0
b	\0	\0
c	\0	\0

再次明确：　　ASCII码

1字节整数　　0　　: 0000 0000

字符常量 '\0' : 0000 0000

字符常量 '0' : 0011 0000

```
#include <iostream>
using namespace std;
int main()
{
    int i, j;
    char a[3][3] = { 'a','b','c' };
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++)
            cout << int(a[i][j]) << ' ';
        cout << endl;
    }
}
```

Microsoft Visual Studio 调试控制台

```
97 98 99
0 0 0
0 0 0
```

```
#include <iostream>
using namespace std;
int main()
{
    int i, j;
    char a[3][3] = {{'a'},{'b'},{'c'}};
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++)
            cout << int(a[i][j]) << ' ';
        cout << endl;
    }
}
```

Microsoft Visual Studio 调试控制台

```
97 0 0
98 0 0
99 0 0
```

§5. 数组

5.5. 字符数组

5.5.4. 字符串

5.5.4.1. 含义

一串连续的字符

5.5.4.2. 字符串的表示与存放

★ 一对双引号可表示字符串常量，字符串中字符的个数称为字符串的长度，字符串的存储形式为每个字符的ASCII码+'\0'(尾零)

问：常量"china"，想变为"China"，能否做到？

★ C++中无字符串变量，可用一维字符数组来表示字符串变量

=> 以一维字符数组方式表示，最后自动加一个'\0'

'\0':字符串结束标志(尾0)

5.5.4.3. 字符数组与字符串的区别

字符串：最后一个字符必须为'\0'

字符数组：最后一个字符不必为'\0'

若无'\0'，不可与字符串相互替代

若有'\0'，可以与字符串相互表示

5.5.4.4. 字符串的长度与字符数组的长度

字符串的长度:在'\0'之前的实际长度

字符数组的长度:数组的大小

```
char a[10]={'c','h','i','n','a','\0','a','b','c','d'};
   字符串长度：5
   字符数组长度：10
   字符数组a表示字符串"china"
   令 a[0] = 'C';
   则字符数组a所表示的字符串变为"China" =>(体现出变量特性)

char a[10]={'c','h','i','n','a','a','b','c','d','\0'};
   字符串长度：9
   字符数组长度：10

char a[10]={'c','h','i','n','a','a','b','c','d','e'};
   字符串长度：不是字符串
   字符数组长度：10
```

§5. 数组

5.5. 字符数组

5.5.4. 字符串

5.5.4.5. 用字符串常量的方式初始化字符数组

A. 全部初始化

```
char a[6]={"china"}; //双引号方式，数组长度为字符串长度+1
char a[6]="china";   //大括号可省略
char a[5]="china";   //数组大小未计算尾零位置则编译报错
```

cpp-demo.cpp

cpp-demo (全局范围)

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     char a[5] = "china";
6     return 0;
7 }
```

(5,24): error C2117: "a": 数组界限溢出

(5): message : 参见"a"的声明

若初始化时不给出数组大小，则字符串方式要算\0位置(字符方式不需要)

```
char a[]="china";                //缺省为6
char a[]={'c','h','i','n','a'};  //缺省为5
```

```
#include <iostream>
using namespace std;
int main()
{
    char a[]="china";
    char b[]={'c','h','i','n','a'};
    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
6
5
```

§5. 数组

5.5. 字符数组

5.5.4. 字符串

5.5.4.5. 用字符串常量的方式初始化字符数组

B. 部分初始化

char a[10]="china";

a[5]-a[9]为'\0'

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    char a[10] = "china";
    for (i = 0; i < 10; i++)
        cout << int(a[i]) << ' ';
    cout << '#' << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
99 104 105 110 97 0 0 0 0 0 #
```

C. 多个字符串初始化二维字符数组

```
char a[3][6] = {"hello", "tong", "ji"};
char a[3][6] = {{"hello"}, {"tong"}, {"ji"}};
char a[][6] = {"hello", "tong", "ji"};
char a[][6] = {{"hello"}, {"tong"}, {"ji"}};
```

h	e	l	l	o	\0
t	o	n	g	\0	\0
j	i	\0	\0	\0	\0

★ 串的个数不超过行大小，串长+1不超过列大小

★ 若初始化时不给出行，则缺省为串的个数

```
char a[3][5] = {"hello", "Hi", "Hello"};
```

有错，为什么？

§5. 数组

5.5. 字符数组

5.5.4. 字符串

5.5.4.5. 用字符串常量的方式初始化字符数组

★ 字符数组定义后，无论是单字符方式，还是字符串方式，均不允许整体进行赋值操作，只能单个元素依次赋值

```
int a[10];
a = {1,2,3,4,5,6,7,8,9,10};  ×
a[3] = 12;                   √
char ch[10];
ch = {'c','h','i','n','a'};  ×
ch = "china";                ×
ch[3] = 'A';                 √
```

★ 字符串可用专用函数进行整体操作，具体见5.6

§5. 数组

5.5. 字符数组

5.5.5. 字符数组的输入与输出

5.5.5.1. 字符的输出

★ 第2章补充 – 字符与字符串的输出

§2. 基础知识

补充：字符的输出

★ (前序)将默认输出窗口从Windows终端改为Windows控制台主机

★ 有效的输出

- 纯ASCII图形字符的输出(基本ASCII码 33-126之间)
- 汉字的输出(成对的扩展ASCII码)
- 汉字+纯ASCII图形字符的混合输出

★ 无效的输出

- 单独的扩展ASCII码的输出(中文系统下非法)
- 非法的情况(不成对的扩展ASCII码、含控制字符)
- 同一程序设置不同显示字体的情况下

★ 尾零的输出

- 不同控制台形式、不同字体下显示不同，最终结论：\0不是合法的图形字符，看到的内容不可信!!!

★ 总结

- 含汉字/图形字符/空格的字符串可以正常显示，否则输出不可信
- 想研究长度/占内存空间的做法：strlen、sizeof
- 想得到某个字节确定值的做法：转int打印

§5. 数组

5.5. 字符数组

5.5.5. 字符数组的输入与输出

5.5.5.1. 字符的输出

★ 第2章补充 – 字符与字符串的输出

5.5.5.2. 字符数组的输出

★ 字符数组中每个元素的单独输出

★ 以字符串方式输出含\0的一维字符数组

★ 以字符串方式输出不含\0的一维字符数组

§5. 数组

5.5. 字符数组

5.5.5. 字符数组的输入与输出

5.5.5.1. 字符的输出

5.5.5.2. 字符数组的输出

★ 字符数组中每个元素的单独输出（注意事项同字符的输出）

- 不要以字符形式输出\0及其它非图形字符(看到的内容不可信)
- 如想准确得知字符的值，转int输出即可

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    char a[10] = {'c','h','i','n','a'};
    for (i=0; i<10; i++)
        cout << a[i] << ' ';//原始
    cout << '*' << endl;

    for (i=0; i<10; i++)
        cout << int(a[i]) << ' ';//整型
    cout << '*' << endl;
    return 0;
}
```

例：初始化数组的前5个元素，再循环输出全部元素

情况1：
新版控制台/新宋体28

情况2：
旧版控制台/新宋体28

情况3：
旧版控制台/新宋体16

§5. 数组

5.5. 字符数组

5.5.5. 字符数组的输入与输出

5.5.5.1. 字符的输出

5.5.5.2. 字符数组的输出

★ 以字符串方式输出含\0的一维字符数组

```
#include <iostream>
using namespace std;
int main()
{
    char a[10] = {'c','h','i','n','a'}; //5个\0
    char b[]   = "hello";               //大小为6
    char c[10] = "同济\0大学";          //含显式\0
    int  d[10] = {'c','h','i','n','a'}; //int数组
    cout << a << '*' << endl;
    cout << b << '*' << endl;
    cout << c << '*' << endl;
    cout << d << '*' << endl;
}
```

(仅可显示字符和\0)

Microsoft Visual Studio 调试控制台

```
china*
hello*
同济*
00F5F894*
```

```
#include <iostream>
using namespace std;

int main()
{
    char s[] = "\bvt\\tnc\4921\x3fr\2a\'\r\\v\a\f";

    cout << strlen(s) << endl;
    cout << sizeof(s) << endl;
    cout << s << '*' << endl;

    return 0;
}
```

第2章 PPT作业
(含其它不可显示字符)

Microsoft Visual Studio 调试控制台

```
21
22
\v*nc
```

★ 以字符串方式输出不含\0的一维字符数组

```
#include <iostream>
using namespace std;
char c[5]; //全部变量不初始化，初值为0（\0）
int main()
{
    char a[5]; //自动变量不初始化，初值随机
    char b[5] = {'c','h','i','n','a'}; //无\0

    cout << a << '*' << endl;
    cout << b << '*' << endl;
    cout << c << '*' << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
烫烫烫烫烫烫 H          ?
china烫烫烫烫烫烫烫烫烫烫抬?↑          ?
*
```

结论：

- 输出跟数组名，其他类型(int/double等)是输出数组的起始地址，而字符数组(char/unsigned char)是表示从数组的起始地址开始依次输出各字符的值，到\0为止（简称：字符串方式输出）
- 字符串方式输出下\0表示结束，\0自身不输出
- 字符串方式输出不含\0的数组，则会持续越界输出直到碰见\0为止（错误的具体表现为正常/乱码等多种形式）
- 如果字符数组中包含多个显式/隐式\0，则输出到第一个\0即停止
- 对于含不可显示字符的字符串，直接输出方式不可信，仍建议用strlen/sizeof等方式进行研究

§5. 数组

5.5. 字符数组

5.5.5. 字符数组的输入与输出

5.5.5.1. 字符的输出

★ 第2章补充视频 – 字符与字符串的输出（自行回顾）

5.5.5.2. 字符数组的输出

★ 字符数组中每个元素的单独输出

★ 以字符串方式输出含\0的一维字符数组

★ 以字符串方式输出不含\0的一维字符数组

5.5.5.3. 字符数组的输入

★ 以单个字符方式输入到字符数组的某个元素中

★ 以字符串方式输入到整个一维字符数组中

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 操作对象

一维字符数组表示的字符串

★ 对应头文件

- 使用前加 #include <string.h> //C方式
  #include <cstring> //C++方式
  - ◆ VS下可以不加这个头文件
- VS认为部分函数不安全，使用前需要加 #define _CRT_SECURE_NO_WARNINGS

★ 常用字符串处理函数

① strlen (const char s[]);
② strcat (char dst[], const char src[]);
③ strncat(char dst[], const char src[], const unsigned int len);
④ strcpy (char dst[], const char src[]);
⑤ strncpy(char dst[], const char src[], const unsigned int len);
⑥ strcmp (const char s1[], const char s2[]);
⑦ strncmp(const char s1[], const char s2[], const unsigned int len);

课上只讲了其中4个，
通过作业进一步理解

- 更多的字符串处理函数通过作业完成并理解
- 教材/参考资料/其它老师的课件中，很多形式是 const char *s，暂时忽略，待学习指针后再进一步理解
- 先不要考虑这些函数的返回值，待学习指针后再进一步理解

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

① strlen(const char s[])

功　　能：求字符串的长度

输入参数：存放字符串的字符数组

返 回 值：整型值表示的长度

注意事项：返回第一个'\0'前的字符数量,不含'\0'

```
//例：字符数组与字符串长度
//读操作，不需要加_CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[]="Hello";
    cout << sizeof(str1) << endl;
    cout << strlen(str1) << endl;

    char str2[]="china\0Hello\0\0";
    cout << sizeof(str2) << endl;
    cout << strlen(str2) << endl;

    return 0;
}
```

6　数组长度
5　字符串长度

14　数组长度
5　字符串长度

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

② strcat(char dst[], const char src[])

功　　能：将字符串src连接到字符串dst的尾部

输入参数：存放字符串dst的字符数组dst

　　存放字符串src的字符数组src(只读)

返 回 值：改变后的字符数组dst

注意事项：字符数组dst要有足够的空间(两串总长+1)

```
//例：字符串连接
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
   char str1[30]="Tongji ";  //不能缺省，至少18字节!!!
   char str2[]="University"; //缺省长度为11
   cout << strcat(str1, str2) << endl;
   return 0;
}
```

cout输出为strcat函数的返回值，也证明了返回值是第1个字符数组

输出为：
Tongji University

```
//例：字符串连接
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
   char str1[]="Tongji ";    //缺省长度为8
   char str2[]="University"; //缺省长度为11
   cout << strcat(str1, str2) << endl;

   return 0;
}
```

观察输出是否有错

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

③ strncat(char dst[], const char src[], const unsigned int n)

功　　能：将字符串src的前n个字符连接到字符串dst的尾部

输入参数：存放字符串dst的字符数组dst

存放字符串src的字符数组src(只读)

要复制的长度n(只读，如果n超过src长度，则只连接src个)

返 回 值：改变后的字符数组dst

注意事项：字符数组dst要有足够的空间(原dst长度+n+1)

```
//例：字符串连接前n个字符
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char str1[30]="Tongji ";
    char str2[30]="Tongji ";
    char str3[]="University"; //缺省长度为11

    cout << strncat(str1, str3, 3) << '*' << endl;
    cout << strncat(str2, str3, 300) << '*' << endl;
    return 0;
}
```

输出为：
Tongji Uni*
Tongji University*

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

④ strcpy(char dst[], const char src[])

功　　能：将字符串src复制到字符串dst中，覆盖原dst串

输入参数：存放字符串dst的字符数组dst

　　存放字符串src的字符数组src(只读)

返 回 值：改变后的字符数组dst

注意事项：字符数组dst要有足够的空间(串src长+1)

● 字符串复制时包含'\0'，到'\0'为止

```
//例：字符串拷贝
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
   int i;
   char a[]="student", b[]="hello";
   strcpy(a,b);
   cout << a << endl;
   for(i=0;i<8;i++)
      cout << (int)a[i] << ' ';
   cout << endl;
   return 0;
}
```

复制前a	s	t	u	d	e	n	t	\0
复制后a	h	e	l	l	o	\0	t	\0

复制到\0为止
a[6]以后保持原值

a/b数组的缺省长度8/6

输出为：
hello
104 101 108 108 111 0 116 0

假设2：
char a[]="student", b[]="hellochina";
1、为什么错？
2、仅修改a的定义使正确，如何做？

假设1：
char a[]="student",b[]="hello\0china";
则：a/b数组的缺省长度8/12

但结果与本例完全相同，
　复制的字符个数也相同

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑤ strncpy(char dst[], const char src[], unsigned int n)

功　　能：将字符串src的前n个复制到字符串dst中，覆盖原dst串

输入参数：存放字符串dst的字符数组dst

存放字符串src的字符数组src（只读）　　继续复制导致出错

要复制的长度n（只读，如果n超过src长度，则~~只复制src个~~）

返 回 值：改变后的字符数组dst　　n+1

注意事项：字符数组dst要有足够的空间（~~min(串src长, n)+1~~）

● strncpy复制时不包含'\0'，且长度超过时出错，要人为保证n正确

```
//例：字符串拷贝前n个字符
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   int i;
    char a[]="student", b[]="hello";
    strncpy(a,b,2);
    cout << a << endl;
    for(i=0;i<8;i++)
       cout << (int)a[i] << ' ';
    cout << endl;
    return 0;
}
```

证明了未加\0

输出为：
heudent
104 101 117 100 101 110 116 0

```
//例：字符串拷贝前n个字符
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{  int i;
   char a[]="student", b[]="hello";
   strncpy(a, _______ ,2);
   cout << a << endl;
   for(i=0;i<8;i++)
      cout << (int)a[i] << ' ';
   cout << endl;
   return 0;
}
```

提示：数组名代表数组的首地址
即　：b ⇔ &b[0]

如果想从b[2]开始复制2个字符到a中，如何做？
即期望输出：lludent

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑤ strncpy(char dst[], const char src[], unsigned int n)

功　　能：将字符串src的前n个复制到字符串dst中，覆盖原dst串

输入参数：存放字符串dst的字符数组dst

存放字符串src的字符数组src（只读）　　继续复制导致出错

要复制的长度n（只读，如果n超过src长度，则~~只复制src个~~）

返 回 值：改变后的字符数组dst　　n+1

注意事项：字符数组dst要有足够的空间（~~min(串src长, n)+1~~）

深度讨论

● strncpy复制时不包含'\0'，且长度超过时出错，要人为保证n正确

```
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   int i;
    char a[] = "student", b[] = "hello";
    for (i = 0; i < 12; i++) //12已越界，目的？
        cout << (int)a[i] << ' ';
    cout << endl;
    strncpy(a, b, 200);
    cout << a << endl;
    for (i = 0; i < 12; i++) //12已越界，目的？
        cout << (int)a[i] << ' ';
    cout << endl;
    return 0;
}
```

D:\WorkSpace\VS2019-Demo\Debug\cpp-demo.exe

```
115 116 117 100 101 110 116 0 -52 -52 -52 -52
hello
104 101 108 108 111 0 0 0 0 0 0 0
```

VS输出为：上图+弹窗

```
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   int i;
    char a[] = "student", b[] = "hello";
    for (i = 0; i < 12; i++) //12已越界
        cout << (int)a[i] << ' ';
    cout << endl;
    strncpy(a, b, 200);
    cout << a << endl;
    for (i = 0; i < 12; i++) //12已越界，目的？
        cout << (int)a[i] << ' ';
    cout << endl;
    return 0;
}
```

D:\WorkSpace\VS2019-Demo\cpp-demo\cpp-demo.exe

```
115 116 117 100 101 110 116 0 8 0 0 0
hello
104 101 108 108 111 0 0 0 8 0 0 0

--------------------------------
Process exited after 0.3279 seconds with return value 3221225477
请按任意键继续. . .
```

D:\WorkSpace\VS2019-Demo\cpp-demo\cpp-demo.exe

```
115 116 117 100 101 110 116 0 8 0 0 0 32 -68 64 0 -32 -2 120 0
hello
104 101 108 108 111 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0

--------------------------------
Process exited after 0.3316 seconds with return value 3221225477
请按任意键继续. . .
```

Dev输出为：上图

正确吗？怎么体现错？

12换20或更大数试试

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑥ strcmp(const char s1[], const char s2[])

功　　能：比较字符串s1和字符串s2的大小

输入参数：存放字符串s1的字符数组s1(只读)

　　存放字符串s2的字符数组s2(只读)

返 回 值：整型值(0:相等　>0:串1大　<0:串1小)

另一种输出形式：
串1 < 串2

```
#include <iostream>
using namespace std;
int main()
{
    char str1[]="abcd", str2[]="abcde";
    int k = strcmp(str1, str2);
    if (k==0)
        cout << "串1 = 串2" << endl;
    else if (k<0)
        cout << "串1 < 串2" << endl;
    else
        cout << "串1 > 串2" << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    char str1[] = "house", str2[]  = "horse";
    char str3[] = "abcd",  str4[]  = "abcde";
    char str5[] = "abcd",  str6[]  = "abc";
    char str7[] = "abcd",  str8[]  = "abcd";
    char str9[] = "abcd",  str10[] = "abcd\0efgh";

    cout << strcmp(str1, str2)  << endl;
    cout << strcmp(str3, str4)  << endl;
    cout << strcmp(str5, str6)  << endl;
    cout << strcmp(str7, str8)  << endl;
    cout << strcmp(str9, str10) << endl;

    return 0;
} //仅比较无赋值，因此不需加_CRT_SECURE_NO_WARNINGS
```

1	串1>串2
-1	<
1	>
0	==
0	==

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑥ strcmp(const char s1[], const char s2[])

- 两串相等的条件是长度相等且对应位置字符的ASCII码值相等

- 字符串的比较流程如下：两串首字符对应比较，若不等则返回非0，若相等则继续比较下一个字符，重复到两串对应字符均为'\0'则结束比较，返回0(相等)

```
strcmp("house", "horse");   到第3个字符结束
strcmp("abcd",  "abcde");   到第5个字符结束
strcmp("abcd",  "abc");     到第4个字符结束
strcmp("abcd",  "abcd");    到第5个字符结束
```

- 不相等返回时，有些系统返回-1/1，有些系统返回第一个不相等字符的ASCII差值，因此一般不比较具体值，而只是判断 >0 / <0 / ==0

```
strcmp("house", "horse");   VS/Dev编译器返回1
                            某些编译器可能返回3
```

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑥ strcmp(const char s1[], const char s2[])

- 不能直接用比较运算符比较字符串的大小(但直接用比较运算符语法不错，只是含义不同)

输出：？

```
#include <iostream>
using namespace std;

int main()
{   char str1[]="house", str2[]="horse";
    int k;
    k = str1 < str2;
    cout << k << endl;
    return 0;
}
```

输出：？

```
#include <iostream>
using namespace std;

int main()
{   char str1[]="horse", str2[]="house";//互换
    int k;
    k = str1 < str2;
    cout << k << endl;
    return 0;
}
```

输出：？

```
#include <iostream>
using namespace std;

char str1[]="house", str2[]="horse";
int main()
{   int k;
    k = str1 < str2;
    cout << k << endl;
    return 0;
}
```

输出：？

```
#include <iostream>
using namespace std;

char str1[]="horse", str2[]="house";//互换
int main()
{   int k;
    k = str1 < str2;
    cout << k << endl;
    return 0;
}
```

问题：

1、四个例子输出分别是什么？为什么语法不错？
< 实际比较的是什么？（基本知识，需要弄懂）

2、局部和全局的定义，结果相反，说明局部变量和全局变量的分配分别是从高地址开始还是低地址开始？（深度讨论，不懂就算了）

§5. 数组

5.6. 常用的字符串处理函数

5.6.1. C方式的字符串处理函数

★ 常用字符串处理函数

⑦ strncmp(const char s1[], const char s2[], const unsigned int n)

功　　能：比较字符串s1和字符串s2的前n个字符的大小

输入参数：存放字符串s1的字符数组s1(只读)

存放字符串s2的字符数组s2(只读)

要比较的长度n(只读，n/s1长/s2长三者关系有影响吗?)

返 回 值：整型值(0:相等　>0:串1大　<0:串1小)

● n大于短串长度(短串长度+1)时，一定会结束

```
//例：字符串比较前n个字符，不需加_CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char str1[] = "abcd",  str2[]  = "abcde";
    cout << strncmp(str1, str2, 3)   << endl; //1~4均可
    cout << strncmp(str1, str2, 5)   << endl;
    cout << strncmp(str1, str2, 100) << endl;
    return 0;
}
```

```
0
-1
-1
```

§5. 数组

5.6. 常用的字符串处理函数

5.6.2. C方式的字符处理函数

① isdigit(char ch) //判断是否数字(0-9)

② isalpha(char ch) //判断是否字母(A~Z，a-z)

③ isalnum(char ch) //判断是否字母或数字(A~Z，a-z，0-9)

④ isxdigit(char ch) //判断是否16进制数字(0-9，A-F，a-f)

⑤ isspace(char ch) //判断是否空格(含回车/换行/换页/tab/竖向tab)

⑥ islower(char ch) //判断是否小写字母(a-z)

⑦ isupper(char ch) //判断是否大写字母(A-Z)

这些判断函数满足条件返回非零，不满足返回0
具体的返回值涉及到位运算概念，暂时忽略(荣誉)

⑧ tolower(char ch) //大写转小写，其余原值返回

⑨ toupper(char ch) //小写转大写，其余原值返回

★ 需要包含头文件<ctype.h>或<cctype>

```
//实现也相对比较简单，不再深入讨论
int isdigit(int ch)
{
    return (ch>='0' && ch<='9') ? 1 : 0;
}

int tolower(int ch)
{
    return (ch>='A'&&ch<='Z') ? ch+32 : ch;
}
```

§5. 数组

5.6. 常用的字符串处理函数

5.6.3. C方式的字符串与格式化输入输出

5.6.3.1. 将格式化数据输出到字符串中

★ 标准输出函数

int printf("格式串", 输出表列);

★ 向字符串输出函数

int sprintf(字符数组, "格式串", 输出表列);

- 返回值是输出字符的个数(同printf)
- 字符数组要有足够空间容纳输出的数据(否则越界错)
- 格式串同printf
- VS下需加 #define _CRT_SECURE_NO_WARNINGS

```
//例：sprintf的基本使用
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

int main()
{
    char str[80];
    int k=123, ret;
    double pi=3.1415925;

    ret = sprintf(str, "k=%-4d*pi=%.2f#", k, pi);
    printf("ret : %d\n", ret);
    printf("str : %s\n", str);

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
ret : 15
str : k=123 *pi=3.14#
```

§5. 数组

5.6. 常用的字符串处理函数

5.6.3. C方式的字符串与格式化输入输出

5.6.3.2. 从字符串中输入格式化数据

★ 标准输入函数

int scanf("格式串", 输入地址表列);

★ 从字符串中输入函数

int sscanf(字符数组, "格式串", 输入地址表列);

- 返回值是正确读入的输入数据的个数(同scanf)
- 格式串同scanf
- VS下需加 #define _CRT_SECURE_NO_WARNINGS

```
//例：sscanf的基本使用
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;

int main()
{
    char str[80] = "Hello 123 11.2", s[10];
    int  i, ret;
    double d;

    ret = sscanf(str, "%s %d %lf", s, &i, &d);
    printf("ret : %d\n", ret);
    printf("s=%s i=%d d=%f\n", s, i, d);

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
ret : 3
s=Hello i=123 d=11.200000
```

§5. 数组

5.6. 常用的字符串处理函数

5.6.3. C方式的字符串与格式化输入输出

Microsoft Visual Studio 调试控制台

```
123  10
01234567890123456789
       123
```

Microsoft Visual Studio 调试控制台

```
54321  8
01234567890123456789
   54321
```

例：输入整型x∈[1..99999]及宽度w∈[6..10]，以右对齐方式输出x

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int x, w;
    cin >> x >> w; //不考虑输入错误
    cout << "01234567890123456789" << endl; //标尺
    cout <<setw(w) << x << endl;

    return 0;
}
```

C++方式

```
//次差方法：%后的宽度要求常量，因此用case做变量转常量
switch(w) {
    case 6:
        printf("%6d", x);
        break;
    case 7:
        printf("%7d", x);
        break;
    ...
}
```

C方式方法2：次差

```
//最差方法：不讲任何技巧的无脑分类讨论
if (w==6) {
    if (x < 10) //1位数
        printf("     %d", x); //5空格
    else if (x < 100) //2位数
        printf("    %d", x);  //4空格
    else if //3位数
        ...
    }
else if (w==7) {
   if (x < 10) //1位数
        printf("      %d", x); //6空格
    else if (x < 100) //2位数
        printf("     %d", x);  //5空格
    else if //3位数
        ...
    }
... //8、9、10
```

C方式方法1：最差

```
//最佳方法：借助sprintf做变量转常量串
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    int x, w;
    scanf("%d %d", &x, &w); //不考虑输入错误
    printf("01234567890123456789\n"); //标尺

    char fmt[16];
    sprintf(fmt, "%%%dd\n", w);//变量w放入fmt
    printf(fmt, x);//fmt中的宽度为常量
    return 0;
}
```

C方式方法3

"%%%dd\n", w

%% => 字符%

%d => 8/10（w）

d => 字符d

=> "%8d"

"%10d"

§5.数组

5.6.常用的字符串处理函数

5.6.4.用C++方式的cin/cout成员函数处理字符串

★ cin.get();　　//多种形式

★ cin.getline();

★ cin.eof()

★ cin.peek()

★ cin.putback()

★ cin.ignore()

★ cout.put();

§5.数组

5.7.C++处理字符串的方法：字符串类与字符串变量

5.7.1.用一维字符数组来表示字符串变量的不足

★ 字符串的长度受限于数组定义时的大小

=> 数组定义过大导致浪费

=> 数组定义过小导致不够

=> 数组定义的大小不能动态变化

★ 赋值、复制、连接等必须用专用函数(strcpy/strcat/...)

5.7.2.字符串类(string类)的引入

string类是C++引入的字符串处理的新方法，通过定义类及运算符重载的方式实现

§5. 数组

5.7. C++处理字符串的方法：字符串类与字符串变量

5.7.3. string类简单变量的定义和使用（与C方式对比）

项目	字符串变量	字符数组
适用	C++	C、C++
头文件	#include <string>	#include <string.h> #include <cstring>
定义	string 变量名 string s1;	char 变量名 char s1[10];
定义时赋初值	string s1="hello"; 长度无限制	char s1[10]="hello"; 长度不超过9

赋值	string s1;
	s1="hello"; 长度无限制
	string s1="hello",s2;
	s2=s1;
		char s1[10];
		strcpy(s1, "hello"); 长度不超过9
		char s1[10]="hello",s2[10];
		strcpy(s2,s1);

单字符操作	string s1="hello"; s1[2]='p';	char s1[10]="hello"; s1[2]='p';
输入	cin	cin、scanf
输出	cout	cout、printf
字符串复制	string s1,s2; ... s1=s2; 不必考虑溢出	char s1[10],s2[10]; ... strcpy(s1, s2); 防止溢出
字符串连接	string s1,s2; ... s1=s1+s2;不必考虑溢出	char s1[20],s2[10]; ... strcat(s1,s2); 防止溢出
	注：s1+s2 ≠ strcat ；s1=s1+s2/s1+=s2 ⇔ strcat	

字符串比较	用比较运算符
	s1==s2;
	s1>s2;
	s1>=s2;
		用函数
		if (!strcmp(s1,s2))
		if (strcmp(s1,s2)>0)
		if (strcmp(s1,s2)>=0)

§5. 数组

5.7. C++处理字符串的方法：字符串类与字符串变量

5.7.3. string类简单变量的定义和使用（与C方式对比）

```
//观察string的所占空间及存放的字符串长度的关系
#include <iostream>
using namespace std;
int main()
{   string s1 = "abc";
    string s2 = "abcdefghijklmnopqrstuvwxyz0123456789";
    cout << sizeof(s1) << ' ' << s1.length() << endl;
    cout << sizeof(s2) << ' ' << s2.length() << endl;
    for (int i = 0; i < 10000; i++)
        s1 += s2;
    cout << sizeof(s1) << ' ' << s1.length() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
28 3
28 36
28 360003
```

现象：大小28字节的变量存储了超过28字节的内容

★ string变量不是原生的基本数据类型，是C++封装的一种复杂数据类型

★ string变量存放的字符串的长度会自动调整，不是固定大小

★ string变量的sizeof大小与存放串的长度无关，涉及到一种新的内存存储和分配方式(动态内存的申请与释放-后续课程)

★ string变量能用+、=、>等运算符来实现连接、赋值、比较等运算，具体的实现原理及实现方法待后续内容(运算符的重载-后续课程)学习完成后再进一步了解

★ 在本节中仅需要了解与一维字符数组表示字符串在表示及使用方法的差异即可

★ 除特定题目外，string类本学期都不准使用

§5.数组

5.7.C++处理字符串的方法：字符串类与字符串变量

5.7.4.string类数组的定义和使用（与C方式对比）

★ 形式

```
string 数组名[正整型常量表达式]; //一维
string 数组名[正整型常量表达式1][正整型常量表达式2]; //二维
string name[5];
string course[3][4];
```

★ 含义

数组有若干元素，每个元素是一个任意长度的string类变量

★ 定义时赋初值

```
string name[5]={"Zhang", "Li", "Wang", "Lin", "Zhao"};
string course[3][4]={ {"C语言", "C++", "Pyhton", "Java"},
                      {"高数", "数分, "高代", "线代"},
                      {"足球", "篮球", "排球", "游泳"}
                    };
```

- 对一维数组，字符串的总数不超过数组大小
- 对二维数组，内括号内字符串个数不超过列，内括号总数不超过行
- 字符串的长度不受限制

§5. 数组

5.7. C++处理字符串的方法：字符串类与字符串变量

5.7.4. string类数组的定义和使用（与C方式对比）

★ 与二维字符数组的内存表示比较(定义时赋初值)

```
char str[3][30]={"Zhang", "Li", "Wang"};
```

Z	h	a	n	g	\0	.....	\0
L	i	\0	\0	\0	\0	.....	\0
W	a	n	g	\0	\0	.....	\0

str占用连续的90个字节

```
string name[3]={"Zhang", "Li", "Wang"};
```

Z	h	a	n	g

L	i

W	a	n	g

name[0]、name[1]、name[2]分别占用连续的5、2、4个字节，但整体上不保证连续

★ string类一维数组与二维字符数组的比较(输入/输出)

```
char str[3][30];
string name[3];
int i;
for (i=0; i<3; i++) {
    cin >> str[i]; //二维数组带单下标，输入长度不超过29
    cin >> name[i];//一维数组带下标，输入长度不限
    cout << str[i] << '-' << name[i] << endl;
    }
```

Microsoft Visual Studio 调试控制台

```
Zhang zhang Li li Wang wang
Zhang-zhang
Li-li
Wang-wang
```

★ string类一维数组与二维字符数组的比较(通过strcpy/=进行赋值)

```
char str[3][30];
string name[3];
strcpy(str[0], "Zhang");
strcpy(str[1], "Li");
strcpy(str[2], "Wang");
name[0] = "Zhang";
name[1] = "Li";
name[2] = "Wang";
```

专用函数strcpy且串长不超过29

赋值运算符长度不限

§6. 指针基础

6.1. 基本概念

★ 数据在内存中的存放

- 根据不同的类型存放在动态/静态数据区
- 数据所占内存大小由变量类型决定 sizeof(类型)

★ 内存地址

- 给内存中每一个字节的编号
- 内存地址的表示根据内存地址的大小一般分为16位、32位和64位

（一般称为地址总线的宽度，是CPU的理论最大寻址范围,具体还受限于其它软、硬件）

16位：0 ~ $2^{16}-1$ (64KB)

32位：0 ~ $2^{32}-1$ (4GB)

64位：0 ~ $2^{64}-1$ (16EB)

例如：32位地址总线
4G内存
则：内存地址表示为
0x00000000
|
0xFFFFFFFF

说明：到目前为止，John von Neumann型计算机的地址都表示为一维线性结构

二进制大数的表示单位：

2^{10} = 1024 =1 KB (KiloByte)
2^{20} = 1024 KB =1 MB (MegaByte)
2^{30} = 1024 MB =1 GB (GigaByte)
2^{40} = 1024 GB =1 TB (TeraByte)
2^{50} = 1024 TB =1 PB (PeraByte)
2^{60} = 1024 PB =1 EB (ExaByte)
2^{70} = 1024 EB =1 ZB (ZettaByte)
2^{80} = 1024 ZB =1 YB (YottaByte)
2^{90} = 1024 YB =1 BB (BrontoByte)
2^{100} = 1024 BB =1 NB (NonaByte)
2^{110} = 1024 NB =1 DB (DoggaByte)
2^{120} = 1024 DB =1 CB (CorydonByte)

★ 内存中的内容

以字节为单位，用一个或几个字节来表示某个数据的值

（基本数据类型一般都是2的n次方）

§6. 指针基础

6.1. 基本概念

★ 内存地址的打印

cout << 地址 / printf("%p", 地址) : 打印连续空间的首字节的地址

```
#include <iostream>
using namespace std;

int main()
{
    int    a = 10;
    double b = 1.2;
    short  c[10] = {0};

    cout << &a << endl;
    printf("%p\n", &b);
    cout << c  << endl; //数组名代表数组的首地址，不加&

    return 0;
}
```

Debug x86

Microsoft Visual Studio 调试控制台

```
00AFF90C
00AFF8FC
00AFF8E0
```

a：占00AFF90C ~ 00AFF90F 四个字节
b：占00AFF8FC ~ 00AFF903 八个字节
c：占00AFF8E0 ~ 00AFF8F3 二十个字节

Debug x64

Microsoft Visual Studio 调试控制台

```
0000002690B9FCC4
0000002690B9FCE8
0000002690B9FD08
```

a：占 **FCC4 ~ **FCC7 四个字节
b：占 **FCE8 ~ **FCEF 八个字节
c：占 **FD08 ~ **FD1B 二十个字节

★ 内存中内容的访问

直接访问：按变量的地址取变量值

间接访问：通过某个变量取另一个变量的地址，再取另一变量的值

存放地址　　存放值

直接访问：拿出钥匙开抽屉，拿到想要的书
间接访问：拿出钥匙开抽屉，拿到另一个抽屉的钥匙，再打开另一个抽屉，拿到想要的书

★ 指针变量

存放地址的变量，称为指针变量

★ 指针

某一变量的地址，称为指向该变量的指针　（地址 ⇔ 指针）

§6. 指针基础

6.2. 变量与指针

6.2.1. 定义指针变量

数据类型 *变量名：表示该变量为指针变量，指向某一数据类型

★ 数据类型称为该指针变量的基类型

★ 变量中存放的是指向该数据类型的地址

int *p：p是指针变量(注意，不是*p)
存放一个int型数据的地址
p的基类型是int型

★ 指针变量所占的空间与基类型无关，与系统的地址总线的宽度有关

16位地址：一个指针变量占16位（2字节）
32位地址：一个指针变量占32位（4字节）
64位地址：一个指针变量占64位（8字节）

特别说明：
1、VS/x86和VS/x64下地址打印的区别参考上一页的例子
2、后续可能以VS/x86模式为基准，直接说"指针变量占4字节"

```
#include <iostream>
using namespace std;

int main()
{
    cout << sizeof(char)   << endl;   1
    cout << sizeof(short)  << endl;   2
    cout << sizeof(int)    << endl;   4
    cout << sizeof(long)   << endl;   4
    cout << sizeof(float)  << endl;   4
    cout << sizeof(double) << endl;   8
    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    cout << sizeof(char *)   << endl;   4
    cout << sizeof(short *)  << endl;   4
    cout << sizeof(int *)    << endl;   4
    cout << sizeof(long *)   << endl;   4
    cout << sizeof(float *)  << endl;   4
    cout << sizeof(double *) << endl;   4
    return 0;
}
```

Debug x86

```
#include <iostream>
using namespace std;

int main()
{
    cout << sizeof(char *)   << endl;   8
    cout << sizeof(short *)  << endl;   8
    cout << sizeof(int *)    << endl;   8
    cout << sizeof(long *)   << endl;   8
    cout << sizeof(float *)  << endl;   8
    cout << sizeof(double *) << endl;   8
    return 0;
}
```

Debug x64

★ 基类型的作用是指定通过该指针变量间接访问的变量的类型及占用的内存大小

§6. 指针基础

6.2. 变量与指针

6.2.2. 使用

变量名 = 地址

*变量名 = 值

```
short i, *p;    long t, *q;
p=&i;           q=&t;
*p=10 ⇔ i=10    *q=10 ⇔ t=10
```

每步分析

short i, *p; long t, *q;

p	???	3000 3003	i	???	2000 2001
q	???	4000 4003	t	???	2100 2103

p=&i; q=&t;

p	2000	3000 3003	i	???	2000 2001
q	2100	4000 4003	t	???	2100 2103

*p=10 ⇔ i=10 *q=10 ⇔ t=10

间接访问：

不是将p/q自身空间赋值为10，

而是将p/q中存放的值(&i/&t)

所对应的空间(i/t)赋值为10

注：i=10 / t=10 称为直接访问

p	2000	3000 3003	i	10	2000 2001
q	2100	4000 4003	t	10	2100 2103

★ 假设32位地址系统，则：

i占2字节是因为_short型_。

t占2字节是因为_long型___。

p/q占4字节是因为_指针类型_。

★ 假设p，q中存放的地址为2000/2100，则

*p=10：表示将2000-2001的2个字节赋值为10

*q=10：表示将2100-2103的4个字节赋值为10

问题1：下面的输出分别是什么？

```
cout << i << ' '  << j  << endl; 10 10
cout << p << ' '  << q  << endl; 2000 2100
cout << *p << ' ' << *q << endl; 10 10
cout << &p << ' ' << &q << endl; 3000 4000
```

问题2：p/q中只存放了变量的首地址，如何知道变量所占字节的长度？

★ 基类型的作用是指定通过该指针变量间接访问的变量的类型及占用的内存大小

```
#include <iostream>
using namespace std;
int main()
{
    char   c, *p;
    double d, *q;
    p = &c;
    q = &d;
    cout << sizeof(p)   << endl;
    cout << sizeof(*p)  << endl;
    cout << sizeof(q)   << endl;
    cout << sizeof(*q)  << endl;
    return 0;
}
```

4：指针变量的大小

1：指针变量的基类型大小

4：指针变量的大小

8：指针变量的基类型大小

4
1
4
8

§6. 指针基础

6.2. 变量与指针

6.2.3. &与*的使用

★ &表示取变量的地址，*表示取指针变量的值

★ 两者优先级相同，右结合

```
int i=5, *p=&i;
```

&*p ⇔ &i ⇔ p

*&i ⇔ i

变量定义时赋初值

```
int i=5, *p=&i;
```

用赋值语句赋值

```
int i=5, *p;
p=&i;
```

p	2000	3000 3003

i	5	2000 2003

§6. 指针基础

6.2. 变量与指针

6.2.4. 指针变量的++/--

★ 指针变量的++/--单位是该指针变量的基类型【指针变量++ ⇔ 所指地址+=sizeof(基类型)】

定义	赋值为10	++运算后地址(假设初始地址均为2000)
char *p1;	*p1=10：2000赋值为10	p1++：p1为2001
short *p2;	*p2=10：2000-2001赋值为10	p2++：p2为2002
long *p3;	*p3=10：2000-2003赋值为10	p3++：p3为2004
float *p4;	*p4=10：2000-2003赋值为10	p4++：p4为2004
double *p5;	*p5=10：2000-2007赋值为10	p5++：p5为2008

```
#include <iostream>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    double d, *p5 = &d;

    cout << p2    << endl;   假设地址A
    cout << ++p2  << endl;   =地址A+2    2字节
    cout << p5    << endl;   假设地址B
    cout << ++p5  << endl;   =地址B+8    8字节

    return 0;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    char   d, *p5 = &d;
    cout << p2    << endl;                    假设地址A
    cout << ++p2  << endl;                    =地址A+2
    cout << hex << (int *)(p5)    << endl;    假设地址B
    cout << hex << (int *)(++p5) << endl;     =地址B+1

    return 0;
}
```

问题1：直接用 cout << p5 << endl;
cout << ++p5 << endl;
为什么会输出一串乱字符？
问题2：为什么输出char型的地址要转为非char型？转int *好还是void *好？

Microsoft Visual Studio 调试控制台
00B3FA98
00B3FA9A
烫烫烫烫虢
烫烫烫烫汦

1、char a[10];
cout << a << endl; //未初始化，无\0则乱码（第5章概念）
2、数组名代表数组首地址 ⇔ 给char型地址表示以字符串方式输出
3、p5 ⇔ &d ⇔ 给出了char型地址 ⇔ 未初始化，无\0则乱码

§6. 指针基础

6.2. 变量与指针

6.2.4. 指针变量的++/--

★ 指针变量的++/--单位是该指针变量的基类型

★ void可以声明指针类型，但不能++/--

(void不能声明变量，但可以是函数的形参及返回值)

void k; × 不允许

void *p; ✓

p++; × 因为不知道基类型的大小

p--; ×

★ *与++/--的优先级关系

*比后缀++/--优先级低 *:3 后缀:2

*与前缀++/--优先级相同，右结合 *:3 前缀:3

int i, *p=&i;

*p++ ⇔ *(p++)：保留p的旧值到临时变量中，p再++(不指向i)，最后取旧值所指的值(i的值)

*++p ⇔ *(++p)：p先++(不指向i)，再取p的值(非i)

(*p)++ ⇔ i++ ：取p所指的值(i)，i值再后缀++

++*p ⇔ ++i ：取p所指的值(i)，i值再前缀++

很多教材和网上资料对后缀++的解释（不够准确）：

先取p所指的值(i),p再++(不指向i)

★ 虽然能解释最终结果的正确性，但无法解释后缀++优先级高于*，为什么不先做++

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;   （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

p1	???	3000 3003
p2	???	3100 3103
p	???	3200 3203

a	???	2000 2003
b	???	2100 2103

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

p1	???
p2	???
p	???

a	???
b	???

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

（假设键盘输入是45 78）

p1	???	3000 3003
p2	???	3100 3103
p	???	3200 3203

a	45	2000 2003
b	78	2100 2103

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

p1	???
p2	???
p	???

a	45
b	78

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

（假设键盘输入是45 78）

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;  （假设键盘输入是45 78）
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

（假设键盘输入是45 78）

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;
    p1=&a;
    p2=&b;

    if (a<b) {
       p=p1;
       p1=p2;
       p2=p;
       }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

（假设键盘输入是45 78）

带地址的图示法
给一个虚假的10进制地址
（一般地址是16进制）

不带具体地址的图示法
（教科书、后续课程）
不具体写明地址，用带
箭头的线段指向来表示

例：键盘输入两数，按从大到小的顺序依次输出（指针法处理）

```
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2, *p, a, b;

    cin >> a >> b;  (假设键盘输入是45 78)
    p1=&a;
    p2=&b;

    if (a<b) {
        p=p1;
        p1=p2;
        p2=p;
        }

    cout << *p1 << ' ' << *p2 << endl;
    return 0;
}
```

赋值语句不能表示为：*p1=&a;
*p2=&b;

若表示为定义时赋初值，则
int a, b, *p1=&a, *p2=&b, *p;

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

实参：整型简单变量
形参：整型简单变量
匹配

为什么无法交换？

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

为什么无法交换？

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

为什么无法交换？

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

为什么无法交换？

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

为什么无法交换？

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int x, int y)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(i,j);
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
}
```

为什么无法交换？

错误原因：C/C++中函数参数是单向传值，形参的改变不能影响实参

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

实参：整型变量地址
形参：整型指针变量
匹配

正确的方法

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

正确的方法

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

正确的方法

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

正确的方法

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

正确的方法

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

例：编写一个函数，将两个整数进行交换后输出

```
void swap(int *x, int *y)
{   int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=10, j=15, *p1=&i, *p2=&j;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(p1,p2);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

实参：整型指针变量
形参：整型指针变量
匹配

与swap(&i,&j)等价

正确的方法

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159

void SL(double R, double *S, double *L)
{
    *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{
    double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;    s=28.2743
    cout << "l=" << l << endl;    l=18.8495
    return 0;
}
```

函数执行后同时得到周长及面积
(都是指针变量做函数形参)

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159

void SL(double R, double *S, double *L)
{
    *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{
    double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;
    cout << "l=" << l << endl;
    return 0;
}
```

实参与形参

s=28.2743
l=18.8495

函数执行后同时得到周长及面积
(都是指针变量做函数形参)

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159

void SL(double R, double *S, double *L)
{
    *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{
    double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;    s=28.2743
    cout << "l=" << l << endl;    l=18.8495
    return 0;
}
```

函数执行后同时得到周长及面积
(都是指针变量做函数形参)

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159

void SL(double R, double *S, double *L)
{
    *S = PI*R*R;
    *L = 2*PI*R;
}
int main()
{
    double s,l, r=3;
    SL(r,&s,&l);
    cout << "s=" << s << endl;    s=28.2743
    cout << "l=" << l << endl;    l=18.8495
    return 0;
}
```

函数执行后同时得到周长及面积
(都是指针变量做函数形参)

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

```
#include <iostream>
using namespace std;
#define PI 3.14159
double SL(double R, double *L)
{
    double S;
    S = PI*R*R;
    *L = 2*PI*R;
    return S;
}
int main()
{
    double s,l, r=3;
    s=SL(r,&l);
    cout << "s=" << s << endl;
    cout << "l=" << l << endl;
    return 0;
}
```

初始内存分配如图所示
请自行画出SL中三句话
执行时内存的变化
理解最后的输出结果

s=28.2743
l=18.8495

函数执行后同时得到周长及面积
周长：指针变量做形参方式
面积：函数返回值方式
注:函数的return只能带一个返回值!!

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

★ 必须通过改变形参指针变量所指变量(即实参)值的方法来达到改变实参值的目的，仅通过改变形参指针变量的值的方法是无效的

```
void swap(int *x, int *y)
{     int *t;
      t = x;
      x = y;
      y = t;
}

int main()
{
      int i=10, j=15;
      cout << "i=" << i << " j=" << j << endl;    i=10  j=15
      swap(&i,&j);
      cout << "i=" << i << " j=" << j << endl;    i=10  j=15
}
```

初始内存分配如图所示
请自行画出swap中三句话
执行时内存的变化
理解为什么无法交换

§6. 指针基础

6.2. 变量与指针

6.2.5. 指针变量作函数的参数

★ 指针变量做函数参数，虽然可以通过形参(实参地址)来间接访问实参，从而达到改变实参值的目的，但本质上仍然是单向传值，而不是形参值回传实参

★ 指针变量做参数，可以同时得到多个改变的实参值，从而达到一个函数返回多个值的目的

★ 必须通过改变形参指针变量所指变量(即实参)值的方法来达到改变实参值的目的，仅通过改变形参指针变量的值的方法是无效的

★ 指针变量的使用，一定要有确定的值，否则会出现错误

● 若定义指针变量未赋初值，则随机指向，无法正常使用，称为野指针(Wild Pointer)/悬挂指针(Dangling Pointer)

```
void swap(int *x, int *y)
{   int *t;
    *t = *x;
    *x = *y;
    *y = *t;
}

int main()
{
    int i=10, j=15;
    cout << "i=" << i << " j=" << j << endl;    i=10 j=15
    swap(&i,&j);
    cout << "i=" << i << " j=" << j << endl;    i=15 j=10
}
```

或 死机或其它非正常现象

VS编译报错
-使用了未初始化的局部变量t
其它编译器可能可以运行
初始内存分配如图所示，请自行画出swap中三句话执行时内存的变化，理解为什么出现严重错误
另1：哪句是错误的关键？
另2：int *t 改为 int tt, *t;
t = &tt;
为什么就正确了？

提示：5000-5003系统是否分配给了程序？

§6. 指针基础

6.3. 一维数组与指针

6.3.1. 基本概念

数组的指针：数组的起始地址(⇔&a[0])

数组元素的指针：数组中某个元素的地址

6.3.2. 指向数组元素的指针变量

short a[10], *p;

p = &a[5];

表示p指向a数组的第5个(从0开始)元素

6.3.3. 指向数组的指针变量

short a[10], *p;

p = &a[0]：数组的第0个元素的地址就是数组的起始地址

p = a　　：数组名代表首地址

§6. 指针基础

6.3. 一维数组与指针

6.3.1. 基本概念

数组的指针：数组的起始地址(⇔&a[0])

数组元素的指针：数组中某个元素的地址

6.3.2. 指向数组元素的指针变量

6.3.3. 指向数组的指针变量

short a[10], *p;

p = &a[5]：表示p指向a数组的第5个(从0开始)元素

p = &a[0]：数组的第0个元素的地址就是数组的起始地址

p = a　　：数组名代表首地址

★ 对一维数组而言，数组的指针和数组元素的指针，其实都是指向数组元素的指针变量(特指0/任意i)，因此本质相同(基类型相同)

★ 数组名代表数组首地址，指针是地址，但本质不同(sizeof(数组名)/sizeof(指针)大小不同)

换为short，结果？

```
#include <iostream>
using namespace std;
int main()
{   int a[10] = {0,1,2,3,4,5,6,7,8,9};
    cout << a                  << endl;   数组a的地址                          地址a
    cout << &a[0]              << endl;   a[0]元素的地址                       地址a
    cout << sizeof(*a)         << endl;   地址a的基类型       ⇔ a[0]的类型    4
    cout << sizeof(*(&a[0]))   << endl;   地址&a[0]的基类型 ⇔ a[0]的类型    4
    cout << sizeof(a)          << endl;   数组a的大小                          40
    cout << sizeof(&a[0])      << endl;   地址&a[0]的大小                      4
}
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.1. 形式

```
int a[10], *p;
p=&a[5];
*p=10 ⇔ a[5]=10
```

指针法　　下标法

6.3.4.2. 下标法与指针法的区别

若int a[10], *p=a

★ p[i] ⇔ *(p+i)

★ a[i] ⇔ *(a+i)

都表示访问数组的第i个元素
等价关系，非常重要!!!

若：int a[10], *p=&a[3]
则：*(p+2)/p[2] ⇔ *(a+5)/a[5]
　　a[0] - a[9] 为合理范围
　　p[-3] - p[6] 为合理范围

```
#include <iostream>
using namespace std;

int main()
{
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    int *p = &a[3], i;

    for (i = 0; i < 10; i++)
        cout << a[i] << ' ';
    cout << endl;

    for (i = -3; i < 7; i++)
        cout << p[i] << ' ';
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

p	2012	3000 3003

0		2000 2003
1		2004 2007
2		2008 2011
3		2012 2015
4		2016 2019
5		2020 2023
6		2024 2027
7		2028 2031
8		2032 2035
9		2036 2039

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.1. 形式

6.3.4.2. 下标法与指针法的区别

★ 数组的首地址不可变，指针的值可以改变

a++ ✗ p++ ✓

★ C/C++语言对指针/数组下标的越界不做检查，因此必须保证引用有效的数组元素，否则可能产生错误

int a[10], *p=a;

p[100]/*(p+100)/a[100]/*(a+100)

编译正确，使用出错

- C++源程序文件中的下标形式在可执行文件中都按指针形式处理，即 a[i] 按 *(a+i) 的方式处理，因此可以理解为可执行文件中已经无下标的概念，也就不会对下标越界进行检查
- VS会有IntelliSense(智能提示)，但不够准确，经常误判

```
//思考：为什么编译不错？如何解释6[a]？
#include <iostream>
using namespace std;

int main()
{
    int a[10] = { 0,1,2,3,4,5,6,7,8,9 };
    cout << a[5] << endl;
    cout << 6[a] << endl;
    return 0;
}
```

★ p[i]/*(p+i)/a[i]/*(a+i)的求值过程

取p/a的地址为基地址，则p[i]/*(p+i)/a[i]/*(a+i)的地址为 基地址+i*sizeof(基类型)

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

(1)-数组名用下标法

```
int main()
{   int a[10], i;
    for(i=0; i<10; i++)
       cin >> a[i];
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

(2)-数组名用指针法

```
int main()
{   int a[10], i;
    for(i=0; i<10; i++)
       cin >> *(a+i);
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << *(a+i) << " ";
    cout << endl;
    return 0;
}
```

4种方法都正确，效率相同

(1)变化-指针用下标法

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> p[i];
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << p[i] << " ";
    cout << endl;
    return 0;
}
```

(2)变化-指针用指针法

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(i=0; i<10; i++)
       cout << *(p+i) << " ";
    cout << endl;
    return 0;
}
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

(3) 用指针运算循环数组

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

p<a+10 表示p和a是否相差10个int型元素

(3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(i=0; i<10; i++)
       cin >> *(p+i);
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

去掉阴影中语句正确

(3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p<a+10; p++)
       cin >> *p;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

去掉后2处阴影中语句错误
为什么？哪个不能去？

执行效率高于前4种实现方式
(前4种的效率相同)
前几种：每次计算p+i*sizeof(int)
本程序：只要 p+sizeof(int)

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.2. 下标法与指针法的区别

★ 常见用法与错误

例：键盘读入10个数并输出

(3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p<a+10;)
       cin >> *p++;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p<a+10;)
       cout << *p++ << " ";
    cout << endl;
    return 0;
}
```

正确

(3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p-a<10;)
       cin >> *p++;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p-a<10;)
       cout << *p++ << " ";
    cout << endl;
    return 0;
}
```

正确

(3)-变化

```
int main()
{   int a[10], i, *p=a;
    for(p=a; p-a<10; p++)
       cin >> *p;
    cout << endl; //先输出一个换行，和输入分开
    for(p=a; p-a<10; p++)
       cout << *p << " ";
    cout << endl;
    return 0;
}
```

正确

p-a<10 ⇔ p<a+10
表示p和a是否相差10个int型的元素

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

A. 指针变量 ± 整数 (包括++/--)

指针变量++ ⇔ 所指地址 += sizeof(基类型)

指针变量-- ⇔ 所指地址 -= sizeof(基类型)

指针变量+n ⇔ 所指地址 + n*sizeof(基类型)

指针变量-n ⇔ 所指地址 - n*sizeof(基类型)

```
#include <iostream>
using namespace std;

int main()
{
    int a[10], *p=a;
    cout << a << "--" << ++p << endl;
    p = &a[5];
    cout << &a[5] << "--" << --p << endl;
    p = &a[3];
    cout << p << "--" << (p+3) << endl;
    p = &a[7];
    cout << p << "--" << (p-3) << endl;
    return 0;
}
```

实际运行一次，观察打印出来的地址间的关系

地址a--地址a+4

地址a+20--地址a+16

地址a+12--地址a+24

地址a+28--地址a+16

Microsoft Visual Studio 调试控制台

```
0079FCF4--0079FCF8
0079FD08--0079FD04
0079FD00--0079FD0C
0079FD10--0079FD04
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

A. 指针变量 ± 整数（包括++/--）

指针变量++ ⇔ 所指地址 += sizeof(基类型)

指针变量-- ⇔ 所指地址 -= sizeof(基类型)

指针变量+n ⇔ 所指地址 + n*sizeof(基类型)

指针变量-n ⇔ 所指地址 - n*sizeof(基类型)

★ 若指针变量指向数组，则±n表示前/后的n个元素

注意不要超出数组的范围，否则无意义

假设：int a[10], *p=&a[3];

p++ : p指向a[4]

p-- : p指向a[2]

p+5 : a[8]的地址(p未变)

p-3 : a[0]的地址(p未变)

p+=3 : p指向a[6]

p-=2 : p指向a[1]

p+9 : a[12]的地址(已越界)

★ 以上每个操作均为独立操作

★ 若指针变量指向简单变量，则语法正确，但无实际意义

假设：int a, b, *p=&a, *q=&b;

p++ :若a的地址为2000，则p指向2004(不再指向a)

q-=3:若b的地址为2100，则q指向2088(不再指向b)

1. 可以运算并得到结果，但结果无实际意义

2. 即使p++/q--指向其它简单变量也没有实际意义

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

B. 两个基类型相同的指针变量相减

指针变量1-指针变量2 ⇔ 地址差/sizeof(基类型)

★ 若两个指针变量都指向同一个数组，则差值表示两者所指元素之间相对位置

```
#include <iostream>
using namespace std;
int main()
{   int a[10], *p=&a[3], *q=&a[5];
    cout << a << endl;
    cout << p << endl;
    cout << q << endl;
    cout << (p-q) << endl;
    cout << (q-p) << endl;
}
```

地址a
地址a+12
地址a+20
-2
2

Microsoft Visual Studio 调试控制台

```
009EF9D4
009EF9E0
009EF9E8
-2
2
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

B. 两个基类型相同的指针变量相减

指针变量1-指针变量2 ⇔ 地址差/sizeof(基类型)

★ 若两个指针变量都指向同一个数组，则差值表示两者所指元素之间相对位置

★ 若两个指针变量分别指向不同的数组，则语法正确，但无实际意义

```
#include <iostream>
using namespace std;
int main()
{   int a[10], *p=&a[3];
    int b[20], *q=&b[12];

    cout << (p-q) << endl;
    cout << (q-p) << endl;
}
```

不同编译器结果不相同

Dev: 11, -11

VS: 16, -16

假设：int a[10], *p=&a[3];
int b[20], *q=&b[12];

则：

cout<<(p-q); 某值x(正负不确定)

cout<<(q-p); -x

可以运算并得到确定结果，但结果无实际意义

★ 若两个指针变量分别指向不同的简单变量，则语法正确，但无实际意义

```
#include <iostream>
using namespace std;
int main()
{   int i, *p=&i;
    int j, *q=&j;

    cout<< (p-q) <<endl;
    cout<< (q-p) <<endl;
}
```

不同编译器结果不相同

Dev: 1, -1

VS: 6, -6

假设：int i, *p=&i;
int j, *q=&j;

则：

cout<<(p-q); 某值x(正负不确定)

cout<<(q-p); -x

可以运算并得到确定结果，但结果无实际意义

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

C. 两个基类型相同的指针变量相减后与整数做比较运算

指针变量1-指针变量2 比较运算符 n

⇔

指针变量1-指针变量2 比较运算符 n*sizeof(基类型)

指针变量1-指针变量2 比较运算符 n　　(p-q < 2)

等价变换

指针变量1 比较运算符 指针变量2 + n　(p < q+2)

```
#include <iostream>
using namespace std;
int main()
{
    int a[10], *p=&a[3], *q=&a[5];

    cout << (q-p == 2) << endl;    1
    cout << (q == p+2) << endl;    1

    cout << (q-p <= 2) << endl;    1
    cout << (q <= p+2) << endl;    1

    cout << (p-q < 0)  << endl;    1
    cout << (p < q)    << endl;    1
}
```

```
int a[10]={...}, *p=a;
for(p=a; p-a<10;)
    cout << *p++ << " ";
```

10表示p和a之间差10个int型元素
(实际地址差40)

```
int a[10]={...}, *p=a;
for(p=a; p<a+10;)
    cout << *p++ << " ";
```

10表示p和a之间差10个int型元素
(实际地址差40)

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

C. 两个基类型相同的指针变量相减后与整数做比较运算

★ 只有当两个指针变量都指向同一个数组时才有意义，若两个指针变量分别指向不同的数组或不同的简单量，则语法正确，但无实际意义（与B相似，不再举例）

★ 指针变量与整数不能进行乘除运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    cout << (p*2) << endl;
    cout << (p/2) << endl;
}
```

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p;
6      cout << (p * 2) << endl;
7      cout << (p / 2) << endl;
8  }
```

```
(6,19): error C2296: "*": 非法，左操作数包含"int *"类型
(7,19): error C2296: "/": 非法，左操作数包含"int *"类型
```

★ 两个基类型相同的指针变量之间不能进行加/乘/除运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int *p, *q;
    cout << (p+q) << endl;
    cout << (p*q) << endl;
    cout << (p/q) << endl;
}
```

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p, * q;
6      cout << (p + q) << endl;
7      cout << (p * q) << endl;
8      cout << (p / q) << endl;
9  }
```

```
(6,19): error C2110: "+": 不能添加两个指针
(7,19): error C2296: "*": 非法，左操作数包含"int *"类型
(7,19): error C2297: "*": 非法，右操作数包含"int *"类型
(8,19): error C2296: "/": 非法，左操作数包含"int *"类型
(8,19): error C2297: "/": 非法，右操作数包含"int *"类型
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.3. 指向数组的指针变量的运算

C. 两个基类型相同的指针变量相减后与整数做比较运算

★ 两个不同基类型的指针变量不能进行包括减及比较在内的任何运算(编译报错)

```
#include <iostream>
using namespace std;
int main()
{   int   *p;
    short *q;
    cout << (p-q)   << endl;
    cout << (p-q<2) << endl;
    cout << (p+q)   << endl;
    cout << (p*q)   << endl;
    cout << (p/q)   << endl;
}
```

```
cpp-demo.cpp
cpp-demo                    (全局范围)
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int* p;
6      unsigned int* q;
7      cout << (p - q) << endl;
8      cout << (p - q < 2) << endl;
9      cout << (p + q) << endl;
10     cout << (p * q) << endl;
11     cout << (p / q) << endl;
12 }
```

```
(7,19): error C2440: "-": 无法从"short *"转换为"int *"
(8,20): error C2440: "-": 无法从"short *"转换为"int *"
(9,19): error C2110: "+": 不能添加两个指针
(10,19): error C2296: "*": 非法，左操作数包含"int *"类型
(10,19): error C2297: "*": 非法，右操作数包含"short *"类型
(11,19): error C2296: "/": 非法，左操作数包含"int *"类型
(11,19): error C2297: "/": 非法，右操作数包含"short *"类型
```

★ void型的指针变量不能进行相互运算(不知道基类型)

```
#include <iostream>
using namespace std;
int main()
{   void *p, *q;
    cout << (p+2)   << endl;   //编译错(void无大小)
    cout << (q--)   << endl;   //编译错(void无大小)
    cout << (p-q)   << endl;   //编译错(void无大小)
    cout << (p<q+1) << endl;   //编译错(void无大小)
    cout << (p<q)   << endl;   //编译错(pq未初始化)
}
```

```
cpp-demo.cpp
cpp-demo                    (全局范围)
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      void* p, *q;
6      cout << (p + 2)      << endl;//编译错(void无大小)
7      cout << (q--)        << endl;//编译错(void无大小)
8      cout << (p - q)      << endl;//编译错(void无大小)
9      cout << (p < q + 1)  << endl;//编译错(void无大小)
10     cout << (p < q)      << endl;//编译错(pq未初始化)
11 }
```

```
(6,19): error C2036: "void *": 未知的大小
(7,15): error C2036: "void *": 未知的大小
(8,19): error C2036: "void *": 未知的大小
(9,23): error C2036: "void *": 未知的大小
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.4.4. 指针变量的各种表示

int a[10], *p=&a[i]; //假设i∈[0..9]

p+1　：取p所指元素(a[i])的下一个数组元素(a[i+1])的地址 p+sizeof(数组类型)

*(p+1)：取p所指元素的下一个数组元素(a[i+1])的值(p不变)

*p+1　：取p所指元素(a[i])的值，值再+1

p++　：p指向下一个数组元素(a[i+1])的地址(p改变)

*(p++)：⇔ *p++，保留p旧值(a[i])，p指向下一个数组元素(a[i]+1)的地址，再取p旧值所指元素(a[i])的值(p改变)

> 很多教材和网上资料对后缀++的解释（不够准确）：
>
> 先取p所指的值(a[i]),p再++(指向a[i+1])
>
> ★ 虽然能解释最终结果的正确性，但无法解释后缀++优先级高于*，为什么不先做++

*++p　：表示p指向下一个数组元素(a[i+1])的地址，再取该元素(a[i+1])的值

(*p)++：取p所指数组元素(a[i])的值，值再++

【注】：如果因为i=9导致a[i+1]越界，则会产生越界错，需要人为保证正确性(VS有IntelliSense)

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.5. 用指针变量作函数参数接收数组地址

★ C/C++语言将形参数组作为一个指针变量来处理

★ 开始等于实参数组的首地址，执行过程中可以改变

本质都是指针变量

```
int main()
{ int a[10];
  ...
  fun(a);
  ...
}
```

实参是数组名，传入数组的首地址

```
fun(int x[10])
{    int x[123]
  ...
  ...
}
```

形参是数组名(带大小)

```
fun(int x[])
{
  ...
  ...
}
```

形参是数组名(不带大小)

```
fun(int *x)
{
  ...
  ...
}
```

形参是指针变量

```
//第05模块例子
#include <iostream>
using namespace std;
void f1(int x1[])      //形参数组不指定大小
{   cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[10])    //形参数组大小与实参相同
{   cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[1234]) //形参数组大小与实参不同
{   cout << "x3_size=" << sizeof(x3) << endl;
}
int main()
{   int a[10];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);
    f2(a);
    f3(a);
}
```

```
a_size=40
x1_size=4   因为int*
x2_size=4   因为int*
x3_size=4   因为int*
(为什么是4指针部分会解释)
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.5. 用指针变量作函数参数接收数组地址

例：选择排序（下标法和指针法对比）

```
void select_sort(int array[], int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (array[j] < array[k])
             k=j;
       t=array[k];
       array[k]=array[i];
       array[i]=t;
       }
}
```

数组法

```
void select_sort(int *array, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (array[j] < array[k])
             k=j;
       t=array[k];
       array[k]=array[i];
       array[i]=t;
       }
}
```

语句理解为数组法访问指针变量

形参是指针变量其余同左

```
void select_sort(int *p, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (p[j] < p[k])
             k=j;
       t= p[k];
       p[k] = p[i];
       p[i] =t;
       }
}
```

array换名为p，其余同上

```
void select_sort(int *p, int n)
{   int i,j,k,t;
    for(i=0; i<n-1; i++) {
       k=i;
       for(j=i+1; j<n; j++)
          if (*(p+j) < *(p+k))
             k=j;
       t= *(p+k);
       *(p+k) = *(p+i);
       *(p+i) =t;
       }
}
```

数组法访问改成指针法访问

指针法

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.5. 用指针变量作函数参数接收数组地址

★ 可以通过改变该指针变量所指的变量的值来达到改变实参数组值的目的

```
//第05模块，数组名及简单变量做函数参数，验证形参修改是否影响实参
#include <iostream>
using namespace std;
void fun(int x[], int y)
{
    x[2]=37; //修改形参数组某元素的值
    y=45;    //修改简单变量形参的值
}
int main()
{   int a[10] = {7, -2, 18, 25}, w=19;
    cout << a[2] << ' ' << w << endl;
    fun(a, w);
    cout << a[2] << ' ' << w << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
18 19
37 19
```

```
//第05模块，数组名及简单变量做函数参数，验证形参修改是否影响实参
#include <iostream>
using namespace std;
void fun(int *x, int y)
{
    *(x+2)=37; //x[2]=37;
    y=45;
}
int main()
{   int a[10] = {7, -2, 18, 25}, w=19;
    cout << a[2] << ' ' << w << endl;
    fun(a, w);
    cout << a[2] << ' ' << w << endl;
    return 0;
}
```

形参本质是指针，用指针法表示x[2]

传参时，函数传地址，简单变量传值

```
//第05模块，数组名及简单变量做函数参数，验证形参修改是否影响实参
#include <iostream>
using namespace std;
void fun(int *x, int y)
{
    *(x+2)=37; //x[2]=37;
    y=45;
}
int main()
{   int a[10] = {7, -2, 18, 25}, w=19;
    cout << a[2] << ' ' << w << endl;
    fun(a, w);
    cout << a[2] << ' ' << w << endl;
    return 0;
}
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.5. 用指针变量作函数参数接收数组地址

★ 可以通过改变该指针变量所指的变量的值来达到改变实参数组值的目的

```
void fun(int *x)
{  *(x+5)=15;
}
int main()
{ int a[10];
  ...
  a[5]=10;
  cout << "a[5]=" << a[5]; a[5]=10
  fun(a);
  cout << "a[5]=" << a[5]; a[5]=15
  ...
}
```

★ 实参数组也可以用指向它的指针变量来代替

```
fun(int *x)
{
    ...
}
int main()
{ int a[10], *p;
  p=a;
  ...
  fun(p);
  ...
}
```

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用数组元素

6.3.5. 用指针变量作函数参数接收数组地址

★ 形参无论表示为数组名形式还是指针变量形式，本质都是一个指针变量

实参/形参的四种组合

```
//形参是一维数组名
void fun(int p[])
{
    ...
}
```

带不带大小都可以

p++正确
本质是指针变量

```
//形参是指针变量
void fun(int *p)
{
    ...
}
```

p++正确

```
//实参是一维数组名
int main()
{   int a[10]={...};

    fun(a);
}
```

a++错误

```
//实参是指针变量
int main()
{   int a[10]={...};
    int *p=a;
    fun(p);
}
```

p++正确

§5. 数组

5.4. 用数组名作函数参数

5.4.1. 用数组元素做函数实参

5.4.2. 用一维数组名做函数实参

这些内容与第04模块中简单变量做实形参的概念不同，第06模块指针中再详细解释

★ 形参为相应类型的一维数组

形参本质是指针变量，只是可以用数组法表示，当然没有大小

★ 实参传递时，将实参数组的首地址(数组名表示数组的首地址)传给形参，因此实、形参数组的内存地址重合(实参占用空间，形参不占用空间)

形参只是指向实参的指针变量，因此可通过访问形参所指变量值的方式来访问实参

★ 形参数组值的改变会影响到实参(与简单参数不同)

★ 因为形参数组不分配空间，因此数组大小可不指定

★ 因为形参数组不分配空间，因此实形参类型必须完全相同，否则编译错

形参指针变量p的基类型必须与实参数组的类型一致，这样p++/p--/*(p+i)/p[i]等操作才等价于访问实参数组的元素

§6. 指针基础

6.3. 一维数组与指针

6.3.4. 指针法引用一维数组元素

6.3.5. 用指针变量作函数参数接收数组地址

★ C/C++语言将形参数组作为一个指针变量来处理

★ 开始等于实参数组的首地址，执行过程中可以改变

★ 可以通过改变该指针变量所指的变量的值来达到改变实参数组值的目的

★ 实参数组也可以用指向它的指针变量来代替

★ 形参无论表示为数组名形式还是指针变量形式，本质都是一个指针变量

§6. 指针基础

6.4. 字符串与指针

6.4.1. 字符指针的定义及使用(重复之前的内容，基类型为char/unsigned char)

char *指针变量名

char *p;

指向简单变量的指针：

char ch='A', *p;

p=&ch; 地址

*p='B'; 值，相当于ch='B'

赋值语句

char ch, *p=&ch; 定义时赋初值

指向一维数组的指针：

char ch[10], *p;

p=&ch[3]; 指向数组的第3个元素

p=ch / p=&ch[0]; 指向数组的开始

§6. 指针基础

6.4. 字符串与指针

6.4.2. 用字符指针指向字符串

数组模块中：

```
char s[]="china"; //一维字符数组,缺省为6
string s="china"; //C++类变量表示，暂不讨论
```

本模块：

```
char *p="china";  //字符指针指向字符串
```

区别：s是一个一维字符数组名，占用一定的内存空间，编译时确定地址不变(s++错)

p是基类型为char的指针变量，表示一个字符(串首字符)的地址,在程序的运行中可以改变

初始状态　　char s[]="china";　　char *p="china";

§6. 指针基础

6.4. 字符串与指针

6.4.2. 用字符指针指向字符串

定义时赋初值

char *p=(char *)"china"; ✓

用赋值语句赋值

char *p;

p="china"; ✓ p表示取地址，将字符串常量的首地址赋给p

*p="china"; ✗ *p表示取值，基类型是char，因此不能是字符串

*p='c'; ? 编译正确，能否正确执行视情况而定，具体例子后面会给出

是否正确？
后续解决

定义时赋初值

char s[]="china"; ✓

用赋值语句赋值

char s[10];

s="china"; ✗

数组不能整体赋值

要用strcpy(s, "China");

*s="china"; ✗

*s⇔s[0],char型值

*s='c'; ✓

cpp-demo.cpp

cpp-demo (全局范围)

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char s[10];
6      s = "china";
7
8      *s = "china";
9
10     *s = 'c';
11 }
```

(6,4): error C3863: 不可指定数组类型“char [10]”
(8,14): error C2440: “=”: 无法从“const char [6]”转换为“char”
(8,7): message : 没有使该转换得以执行的上下文

§6. 指针基础

6.4. 字符串与指针

6.4.3. 字符指针的打印

```
#include <iostream>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    double d, *p5 = &d;

    cout << p2    << endl;
    cout << ++p2  << endl;
    cout << p5    << endl;
    cout << ++p5  << endl;

    return 0;
}
```

假设地址A
=地址A+2　2字节
假设地址B
=地址B+8　8字节

★ 本页重复6.2.4的内容

```
#include <iostream>
using namespace std;
int main()
{
    short  s, *p2 = &s;
    char   d, *p5 = &d;

    cout << p2    << endl;
    cout << ++p2  << endl;
    cout << hex << (void *)(p5)    << endl;
    cout << hex << (void *)(++p5)  << endl;

    return 0;
}
```

问题1：直接用 cout << p5 << endl;
cout << ++p5 << endl;
为什么会输出一串乱字符？

问题2：为什么输出char型的地址要转为非char型？
转int *好还是void *好？

假设地址A
=地址A+2　2字节
假设地址B
=地址B+1　1字节

1、char a[10];
cout << a << endl; //未初始化，无\0则乱码（第5章概念）
2、数组名代表数组首地址 ⇔ 给char型地址表示以字符串方式输出
3、p5 ⇔ &d ⇔ 给出了char型地址 ⇔ 未初始化，无\0则乱码

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

```
char a[]="china", *p;
p=a; / p=&a[0];
```

a[i]	p[i]
*(a+i)	*(p+i)
*a++	*p++
(*a)++	(*p)++

8种表示方式中有一种是错误的？哪种？

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout << "str1:" << p1 << endl
                << "str2:" << p2 << endl;
    return 0;
}
```

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    int i;
    for(i=0;str1[i]!='\0';i++)
        str2[i] = str1[i];
    str2[i]='\0';
    cout << str1 << endl;
    cout << str2 << endl;
    return 0;
}
```

数组法实现

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    int i;
    for(i=0;*(str1+i)!='\0';i++)
        *(str2+i) = *(str1+i);
    *(str2+i)='\0';
    cout << str1 << endl;
    cout << str2 << endl;
    return 0;
}
```

等价指针法

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{  char str1[]="I Love CHINA!",str2[20];
   char *p1,*p2;
   p1=str1;
   p2=str2;
   for(;*p1!='\0';p1++,p2++)
      *p2=*p1;
   *p2='\0';
   p1=str1;
   p2=str2;
   cout ...
   return 0;
}
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

```
例：将str1的内容复制给str2
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

```
例：将str1的内容复制给str2
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

！赋值完成后，循环结束
注意：\0未赋

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

！赋值完成后，循环结束
注意：\0未赋

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

重新指向
并打印

Microsoft Visual Studio 调试控制台

```
str1=I Love CHINA!
str2=I Love CHINA!
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    for(;*p1!='\0';p1++,p2++)
        *p2=*p1;
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

1、判断
2、赋值
3、++到下一元素
循环结束,\0未赋

```
for(;*p1!='\0';)
    *p2++ = *p1++;
```

```
for(;*p1;)
    *p2++ = *p1++;
```

```
while(*p1)
    *p2++ = *p1++;
```

几种等价表示

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    while((*p2=*p1)!='\0') {
        p1++;   p2++;
        }
    ~~*p2='\0';~~
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

1、赋值
2、判断
3、++到下一元素
循环结束，\0已赋

```
for(;*p1!='\0';p1++,p2++)
    *p2=*p1;
```

1、判断
2、赋值
3、++到下一元素
循环结束，\0未赋

虽不完全等价
但都是正确的

```
while(*p2=*p1) {
    p1++; p2++;
    }
```

```
while(*p2++ = *p1++);
```

1、赋值
2、++到下一元素
3、判断旧值
循环结束，\0已赋

```
while(*p2++ = *p1++)
    ;
```

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

例：将str1的内容复制给str2

```
int main()
{   char str1[]="I Love CHINA!",str2[20];
    char *p1,*p2;
    p1=str1;
    p2=str2;
    do {
        *p2++ = *p1++;
    } while(*p1);
    *p2='\0';
    p1=str1;
    p2=str2;
    cout ...
    return 0;
}
```

1、赋值
2、++到下一元素
3、判断新值
循环结束,\0未赋

问：改为do-while循环后是否正确?
答：就本例而言，是正确的
若 char str1[]="" ，即空串的情况下有可能错误

1、str2[0] = str1[0]
'\0' <= '\0'
2、++到 str1[1]
3、判断str1[1]是否\0

至此，str1[1]已越界，但程序会继续执行
若str1[1]!='\0'，则
str2[1] = str1[1]
依次类推到str1[x]是'\0'为止

若x<=19，则读非法，但写仍在str2的合理范围内,错误可能不体现
若x>19，则读非法，且对str2的赋值会超过数组长度20，会导致非法写，出错概率大于x<=19

课后自行尝试：
现有编译器排列组合，观察x的值
VS/Debug/x86
VS/Debug/x64
VS/Release/x86
VS/Release/x64

Dev/32bit/Debug
Dev/32bit/Release
Dev/32bit/Profiling
Dev/64bit/Debug
Dev/64bit/Release
Dev/64bit/Profiling

各种可能的表现：
1、x<20，返回0，不弹窗
2、x<20，返回非0，不弹窗
3、x<20，返回非0，弹窗
4、x>=20，返回0，不弹窗
5、x>=20，返回非0，不弹窗
6、x>=20，返回非0，弹窗
7、其它错误表现
注：任何表现均可接受，具体表现无讨论价值

§6. 指针基础

6.4. 字符串与指针

6.4.4. 下标法与指针法处理字符串

```
#include <iostream>
using namespace std;
int main()
{  char str1[]="I love CHINA!", str2[20];
   char *p1 = str1, *p2 = str2;
   cout << hex << (int *)(str2) << endl;
   for(;*p1!='\0';p1++,p2++)
       *p2=*p1;
   *p2='\0';
   cout << hex << (int *)(p2) << endl;
   cout << dec << p2-str2 << endl;
   cout << str2 << endl;
}
```

完全正确
地址str2

地址str2+13
13
I love CHINA!

```
#include <iostream>
using namespace std;
int main()
{  char str1[]="", str2[20];
   char *p1 = str1, *p2 = str2;
   cout << hex << (int *)(str2) << endl;
   for(;*p1!='\0';p1++,p2++)
       *p2=*p1;
   *p2='\0';
   cout << hex << (int *)(p2) << endl;
   cout << dec << p2-str2 << endl;
   cout << str2 << endl;
}
```

完全正确
地址str2

地址str2
0
空行

```
#include <iostream>
using namespace std;
int main()
{  char str1[]="I Love CHINA!", str2[20];
   char *p1 = str1, *p2 = str2;
   cout << hex << (int *)(str2) << endl;
   do {
      *p2++ = *p1++;
   } while(*p1);
   *p2='\0';
   cout << hex << (int *)(p2) << endl;
   cout << dec << p2-str2 << endl;
   cout << str2 << endl;
}
```

目前正确
地址str2

地址str2+13
13
I love CHINA!

```
#include <iostream>
using namespace std;
int main()
{  char str1[]="", str2[20];
   char *p1 = str1, *p2 = str2;
   cout << hex << (int *)(str2) << endl;
   do {
      *p2++ = *p1++;
   } while(*p1);
   *p2='\0';
   cout << hex << (int *)(p2) << endl;
   cout << dec << p2-str2 << endl;
   cout << str2 << endl;
}
```

运气好正确
本质不正确!!!
地址str2

地址str2+x
x（不确定）
空行

§6. 指针基础

6.4. 字符串与指针

6.4.5. 指向字符数组的指针作函数参数(四种组合,同前)

```
//形参是指针变量
void fun(char *s)
{
    ...
}
```

```
//形参是一维数组名
void fun(char s[])
{
    ...
}
```

三种形式均可
[]　：空
[6]　：与实参大小一致
[123]：与实参大小不一致

```
//实参是一维数组名
int main()
{
    char str[]="Hello";
    ...
    fun(str);
}
```

```
//实参指针变量
int main()
{
    char str[]="Hello";
    char *p=str;
    fun(p);
}
```

§6. 指针基础

6.4. 字符串与指针

6.4.6. 字符指针与字符数组的区别

★ 字符数组占用一定的连续存储空间，而指针仅有一个有效地址的存储空间

```
char s[80];
cin >> s;
```

当输入小于80个字符时，正确

s	2000	预留空间
	2079	

```
char *s;
cin >> s;
```

VS编译报error

```
#include <iostream>
using namespace std;
int main()
{
    char *s;
    cin >> s;
    return 0;
}
```

(6): error C4700: 使用了未初始化的局部变量“s”

其他编译器能运行，但本质是错的

s	3000 3003	?(假设5000)

设键盘输入10个字符，则会覆盖5000-5010的11字节的空间(非法占用)

```
#include <iostream>
using namespace std;
int main()
{
    char *s;
    cin >> s;
    return 0;
}
```

```
Hello

Process exited after 6.662 seconds with return value 0
请按任意键继续. . .
```

编译及运行结果暂时看不出问题

如何使正确？

```
char ss[80], *s;
s=ss;   //使s指向确定空间
cin >> s;
```

§6. 指针基础

6.4. 字符串与指针

6.4.6. 字符指针与字符数组的区别

★ 字符数组占用一定的连续存储空间，
而指针仅有一个有效地址的存储空间

★ 赋初值的方式相同，但含义不同

```
char s[]="china";
char *p="china";
```

s	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0

字符串常量
"china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

p	4000 4003	3000

★ 字符数组在执行语句中赋值时只能逐个进行，而字符指针仅能整体赋首址

```
char s[80];
s="china";×
```

数组不允许整体赋值

```
s[0]='c'; ✓
s[1]='h'; ✓
s[2]='i'; ✓
s[3]='n'; ✓
s[4]='a'; ✓
s[5]='\0'; ✓
```

s	2000
	预留空间
	2079

```
char *s;
s="china"; ✓
```

```
s[0]='c'; ×
s[1]='h'; ×
s[2]='i'; ×
s[3]='n'; ×
s[4]='a'; ×
s[5]='\0'; ×
```

修改常量值

s	2100 2103	3000

字符串常量
"china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

★ 数组首地址的值不可变，指针的值可变

§6. 指针基础

6.4. 字符串与指针

6.4.6. 字符指针与字符数组的区别

6.4.2 中的遗留问题：

定义时赋初值

```
char *p="china"; ✓
```

用赋值语句赋值

```
char *p;
p="china";   ✓   p表示取地址，将字符串常量的首地址赋给p
*p="china";  ×   *p表示取值，基类型是char，因此不能是字符串
*p='c';      ?   编译正确，能否正确执行视情况而定，具体例子后面会给出
```

是否正确？
后续解决

正确/错误的各种情况

```
int main()
{   char *p;
    *p='c';
    return 0;
} //编译有警告，运行错
```

error C4700: 使用了未初始化的局部变量“p”

```
int main()
{   char ch, *p=&ch;
    *p='c';
    return 0;
} //正确,ch被赋值为'c'
```

```
int main()
{   char *p=(char *)"Hello";
    *p='c';
    return 0;
} //编译不错运行错
```

已退出，代码为 -1073741819。

```
int main()
{   char ch[]="Hello";
    char *p=ch;
    *p='c';
    return 0;
} //正确,ch变为"cello"
```

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

p	3000 3003	?

x	4000 4003	?

a	2000	0
	2004	1
	2008	2
	2012	3
	2016	4
	2020	5
	2024	6
	2028	7
	2032	8
	2036	9

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl;  *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl;  *p=6
}
```

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

```
#include <iostream>
using namespace std;

int *fun(int *x)
{
    x++;
    return x;
}
int main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9}, *p;
    p=fun(a);
    cout << "*p=" << *p << endl; *p=1
    p=fun(a+5);
    cout << "*p=" << *p << endl; *p=6
}
```

请自行画出打印 *p=6 的示例图

§6. 指针基础

6.5. 返回指针值的函数

6.5.1. 定义

返回基类型 *函数名（形参表）

int *fun(int x)

float *function(char ch)

6.5.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

```
int *fun()
{
    int k=10;
    return &k;//不允许
}
```

```
int *fun(int k)
{
    return &k;//不允许
}
```

warning C4172: 返回局部变量或临时变量的地址: k

§6. 指针基础

6.5. 返回指针值的函数

6.5.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;

int *fun()
{
    int k=10;
    return &k;//warning
}

int main()
{
    int *p;
    p=fun();
    cout << *p << endl;
    return 0;
}
```

warning C4172: 返回局部变量或临时变量的地址: k

Microsoft Visual Studio 调试控制台

10

cout时，k所占空间已释放

未被再次分配并赋值：*p=10

已被再次分配并赋值：*p=其他

=> *p不可信

1、VS有编译警告

2、读非法空间而未写，不会死机

问：如果必须返回局部变量的值，如何做？

答:设为静态局部即可(k不释放)

§6. 指针基础

6.5. 返回指针值的函数

6.5.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;
int *fun()
{   int k=10;
    cout << &k << endl;
    return &k; //warning
}
void fun2()
{   int m=20;
    cout << &m << endl;
}
int main()
{   int *p;
    p=fun();
    fun2();
    cout << *p << endl;
}
```

地址a
地址a
20

Microsoft Visual Studio 调试控制台

```
0136FD78
0136FD78
20
```

调用fun2时，k所占空间已释放，因此m申请的空间会重复利用原k的空间(证明：k/m的地址一样)

打印*p时，k/m所占空间已经释放

=> 程序错误，*p不可信

§6. 指针基础

6.5. 返回指针值的函数

6.5.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

```
#include <iostream>
using namespace std;

void fun()
{
    int k=10;
    cout << &k << endl;
}
int main()
{
    int i;
    for(i=0; i<10; i++)
        fun();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
009DFDC8
```

上例证明了自动变量k每次函数调用完成后会释放空间

本例证明了再次调用时，k分配的空间大概率是上次分配的空间

（04模块中：两次分配不保证同一空间）

§6. 指针基础

6.5. 返回指针值的函数

6.5.2. 使用

★ return中的返回值必须是指针（地址）

★ 不能返回一个自动变量/形参的地址，否则可能出错

★ 可以通过返回数组首地址（指针）的方式来返回整个数组

```
#include <iostream>
using namespace std;

char *my_strcpy(char *dst, const char *src)
{
    int i;
    for (i=0; src[i]; i++)
        dst[i] = src[i];
    dst[i] = 0;
    return dst;
}
int main()
{
    char s1[]="student", s2[]="hello";
    cout << s1 << endl;
    cout << my_strcpy(s1, s2) << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
student
hello
```

§6. 指针基础

6.6. 空指针NULL

★ 指针允许有空值 NULL(系统宏定义#define NULL 0)，表示不指向任何变量

- 若定义指针变量未赋初值，则随机指向，无法正常使用，称为野指针(Wild Pointer)/悬挂指针(Dangling Pointer)

NULL与空字符串的区别：

```
char *s1 = NULL; //s1是指针，存放地址0，地址0中的内容不一定是'\0'，
                 //即strlen(s1)不一定为0
char *s2 = "";   //s2是指针，存放一个长度为0的无名字符串常量的首地址(非0)，
                 //strlen(s2)为0
```

```
char s3[]="";    //正确
char s4[]=NULL;  //错，不能用无{}的一个数字初始化
```

```
#include <iostream>
using namespace std;
int main()
{   char s3[]="";
    char s4[]=NULL;//编译报错
}
```

error C2440: "初始化": 无法从"int"转换为"char []"
message : 没有转换为数组类型，但有转换为数组的引用或指针

§6. 指针基础

6.6. 空指针NULL

★ 指针允许有空值 NULL(系统宏定义#define NULL 0)，表示不指向任何变量

- 若定义指针变量未赋初值，则随机指向，无法正常使用，称为野指针(Wild Pointer)/悬挂指针(Dangling Pointer)

★ 系统的字符串操作函数若传入参数为NULL则会出错

(包括 strcpy/strcat/strcmp/strlen/strncpy/strncmp等，以及未出现过的同类函数)

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    int len;
    len=strlen(s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcpy(s2, s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcat(s2, s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char *s2 = NULL;
    int k=strcmp(s1,s2);
}
```

错

```
已退出，代码为 -1073741819。
```

=> 自行实现类似功能的字符串处理函数时，可以对NULL进行特殊处理(具体见作业，这不是标准，只是为了强行与系统函数不同)

- 求长度时为0
- 复制、连接、拷贝时当做空串进行处理

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

含义：变量的别名

声明：int a, &b=a; //a和b表示同一个变量

★ 引用不分配单独的空间(指针变量有单独的空间)

变量的定义：分配空间

变量的声明：不分配空间

★ 引用需在声明时进行初始化，指向同类型的变量，在整个生存期内不能再指向其它变量

```
int a, &c=a; //正确
int b, &c=b; //错
       &c=b; //错
```

c已是a的别名,不能再b

无论定义/赋值均不行

```
int a, &c=a, b;
  c=b/b=c ⇔ a=b/b=a
int a, &c=a, b[10];
  c=b[3]  ⇔ a=b[3]
```

都正确

★ 不能声明引用数组和指向引用的指针，但可声明数组的引用、数组元素的引用和指向指针的引用

```
int &b[3];            //错误，不能声明引用数组
int &*p;              //错误，不能定义指向引用的指针
int a[5],(&b)[5]=a;   //正确，引用指向整个数组
int a[5],&b=a[3];     //正确，引用指向数组元素
int *a, *&b=a;        //正确，指向指针的引用
```

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

含义：变量的别名

声明：int a, &b=a;　　//a和b表示同一个变量

★ 引用不分配单独的空间(指针变量有单独的空间)

★ 引用需在声明时进行初始化，指向同类型的简单变量，在整个生存期内不能再指向其它变量

★ 不能声明指向数组的引用、引用数组和指向引用的指针，但可声明数组元素的引用和指向指针的引用

★ &的理解

定义语句新变量名前：引用声明符

```
int a, &b=a;
```

其它(定义语句已定义变量名，执行语句)：　取地址运算符

```
int a, *p=&a;
p=&a;
```

引用的简单使用：出现在普通变量可出现的任何位置

```
//例：简单变量的引用
#include <iostream>
using namespace std;
int main()
{   int a=10, &b=a;
    a=a*a;
    cout << a << " " << b << endl;    100   100
    b=b/5;
    cout << a << " " << b << endl;    20    20
    return 0;
}
```

本例无任何实用价值
1、多定义一个名称
2、两者容易混淆

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

例：两数交换

```
void swap(int x, int x)
{   int t;
    t = x;
    x = y;
    y = t;
}
int main()
{   int i=3, j=5;
    swap(i, j);
    cout << i << " " << j << endl;
    return 0;
}
```

直接传值
错误

```
void swap(int *x, int *y)
{   int t;
    t  = *x;
    *x = *y;
    *y = t;
}
int main()
{   int i=3, j=5;
    swap(&i, &j);
    cout << i << " " << j << endl;
    return 0;
}
```

传地址
正确

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

例：两数交换

```
void swap(int &x, int &y)
{   int t;
    t = x;
    x = y;
    y = t;
}

int main()
{   int i=3, j=5;
    swap(i, j);
    cout << i << " " << j << endl;
    return 0;
}
```

引用做形参
正确

★ 形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问

★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合(地址传递方式)

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合(地址传递方式)

- C++函数参数传递的两种方式
  - ◆ 传值：单向传值，实形参分占不同空间

```
void fun(int x)
{ …
}

int main()
{  int k = 10;
   fun(k);
}
```

形参的变化
不影响实参

```
void fun(int *x)
{ …
}

int main()
{  int k=10;
   fun(&k);
}
```

可通过形参间接
访问实参，但本质
仍是单向传值

  - ◆ 传址：实形参重合，对形参的访问就是对实参的访问

```
void fun(int *x)
{ …
}

int main()
{  int k[10] = {…};
   fun(k);
}
```

对形参数组
的修改影响
实参数组

```
void fun(int &x)
{ …
}

int main()
{  int k=10;
   fun(k);
}
```

对形参的访问
就是对实参的访问

§6. 指针基础

6.7. 引用(C++新增)

6.7.1. 引用的基本概念

6.7.2. 引用作函数参数

★ 形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问

★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合(地址传递方式)

★ 引用允许传递

```
void swap1(int &a, int &b)
{   int t;
    t = a;
    a = b;
    b = t;
}
void swap(int &x, int &y)
{
    swap1(x, y);
}
int main()
{
    int i=3, j=5;
    swap(i, j);
}
```

§6. 指针基础

6.7. 引用(C++新增)

6.7.2. 引用作函数参数

★ 形参是引用时，不需要声明时初始化，调用时，形参不分配空间，只是当作实参的别名，因此对形参的访问就是对实参的访问

★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合(地址传递方式)

★ 引用允许传递

★ 当引用做函数形参时，实参不允许是常量/表达式，否则编译错误(形参为const引用时实参可为常量/表达式)

```
#include <iostream>
using namespace std;

void fun(int x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);    //正确
    fun(15);   //正确
    return 0;
}
```

```
#include <iostream>
using namespace std;

void fun(int &x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);    //正确
    fun(15);   //编译错
    return 0;
}
```

error C2664: “void fun(int &)”: 无法将参数 1 从“int”转换为“int &”

```
#include <iostream>
using namespace std;

void fun(const int &x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);    //正确
    fun(15);   //正确
    return 0;
}
```

问：系统认为错误的原因是什么？是为了防止什么隐患出现？
答：引用做函数形参，表示形参是可读可写的，而实参是常量/表达式，不可写
=> 形参是实参别名，但得到的权限(读/写)大于原始权限(只读)

§6. 指针基础

6.7. 引用(C++新增)

6.7.3. 关于引用的特别说明

★ 引用在需要改变实参值的函数调用时比指针方式更容易理解，形式也更简洁，不容易出错

★ 引用不能完全替代指针(可以将指针理解为if-else，引用理解为switch-case)

★ 引用是C++新增的，纯C的编译器不支持，后续工作学习中接触的大量底层代码仍是由C编写的，此时无法使用引用
(VS/Dev都是C++编译器，兼容编译纯C，以后缀名.c/.cpp来区分如何编译)

★ 对于计算机底层而言，仍需要透彻理解指针!!!

§6. 指针基础

6.8. 不同基类型指针的相互赋值

★ 不同类型的指针变量不能直接相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;

int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;

    p1 = p;   //编译错
    p2 = p;   //编译错

    cout << *p  << endl;
    cout << *p1 << endl;
    cout << *p2 << endl;

    return 0;
}
```

```
(9,11): error C2440: "=" : 无法从"long *"转换为"short *"
(9,10): message : 与指向的类型无关；强制转换要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
(10,11): error C2440: "=" : 无法从"long *"转换为"char *"
(10,10): message : 与指向的类型无关；强制转换要求 reinterpret_cast、C 样式强制转换或函数样式强制转换
```

```
#include <iostream>
using namespace std;

int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;

    p1 = (short *)p;
    p2 = (char *)p;

    cout << *p  << endl;   70000
    cout << *p1 << endl;   4464
    cout << *p2 << endl;   p

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
70000
4464
p
```

70000 = 00000000 00000001 00010001 01110000

p	2000
p1	2000
p2	2000

a:低位在前存放	
2000	01110000
2001	00010001
2002	00000001
2003	00000000

§6. 指针基础

6.8. 不同基类型指针的相互赋值

★ 不同类型的指针变量不能直接相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;
int main()
{
    char s[]="abcd";
    int *p=(int *)s;

    cout << dec << *p << endl; //取基类型为int的指针变量p的值
    cout << hex << *p << endl; //取基类型为int的指针变量p的值

    return 0;
}
```

1684234849 64636261	Microsoft Visual Studio 调试控制台 1684234849 64636261

s[5]	
2000	a
2001	b
2002	c
2003	d
2004	\0

s[5]	
2000	0110 0001
2001	0110 0010
2002	0110 0011
2003	0110 0100
2004	0000 0000

p	2000

int型(低位在前)	
2000	0110 0001
2001	0110 0010
2002	0110 0011
2003	0110 0100

1684234849 = 01100100 01100011 01100010 01100001

§6. 指针基础

6.8. 不同基类型指针的相互赋值

★ 不同类型的指针变量不能直接相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;

int main()
{
    int a=4276803; //0x414243
    char *s=(char *)&a;
    cout << s << '#' << endl; //串方式输出

    return 0;
}
```

CBA

Microsoft Visual Studio 调试控制台

CBA#

6276803 = 00000000 01000001 01000010 01000011

a:低位在前存放

2000	0100 0011
2001	0100 0010
2002	0100 0001
2003	0000 0000

s	2000

字符型

2000	C
2001	B
2002	A
2003	\0

```
int main()
{
    int a= 0x41424344;
    char *s=(char *)&a;
    cout << s << '#' << endl; //串方式输出

    return 0;
}
```

Microsoft Visual Studio 调试控制台

DCBA烫烫萌\?默#

为什么?

§6. 指针基础

科学

1.50146484375 × 2 ^ 13 =

12,300

6.8. 不同基类型指针的相互赋值

★ 不同类型的指针变量不能直接相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;
int main()
{   double d=1.23e4;
    unsigned char *p=(unsigned char *)&d;
    for (int i=0; i<8; i++)
        cout << hex << int(p[i]) << ' ';
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0 0 0 0 0 6 c8 40
```

1.23e4 =	0100 0000	1100 1000	0000 0110	40bit0

尾数符号位：0
阶码：100 0000 1100 = 1036 - 1023 = 13
尾数：1000 0000 0110 … = 0.50146484375
加1 = 1.50146484375
1.50146484375 x 2^{13} = 12300 = 1.23e4

d:低位在前存放	
2000	0x00
2001	0x00
2002	0x00
2003	0x00
2004	0x00
2005	0x06
2006	0xc8
2007	0x40

```
#include <iostream>
using namespace std;
int main()
{   float d=1.23e4; //无warning
    unsigned char *p=(unsigned char *)&d;
    for (int i=0; i<4; i++)
        cout << hex << int(p[i]) << ' ';
    cout << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0 30 40 46
```

1.23e4 =	0100 0110	0100 0000	0011 0000	0000 0000

尾数符号位：0
阶码：100 0110 0 = 140 - 127 = 13
尾数：100 0000 0011 0000 0000 0000 = 0.50146484375
加1 = 1.50146484375
1.50146484375 x 2^{13} = 12300 = 1.23e4

d:低位在前存放	
2000	0x00
2001	0x30
2002	0x40
2003	0x46

§6. 指针基础

6.8. 不同基类型指针的相互赋值

★ 不同类型的指针变量不能直接相互赋值，若要赋值，则需要进行强制类型转换

例：验证浮点数表示是否存在误差的样例程序

```
#include <iostream>
using namespace std;

int main()
{
    float d1 = 1.23e4; //无warning
    cout << (d1==1.23e4) << endl;

    float d2 = 1.2; //有warning
    cout << (d2==1.2) << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1
0
```

§7. 结构体、类和对象

7.1. 用户自定义类型的引入

7.1.1. 用户自定义类型(派生类型)的含义

用基本数据类型以及已存在的自定义数据类型组合而成的新数据类型

7.1.2. 自定义数据类型的分类

元素同类型的自定义数据类型：数组

元素不同类型的自定义数据类型：结构体、共用体、类

§2. 基础知识

2.5. C++的数据类型

2.5.1. 数据类型分类

- 数据类型
  - 基本类型
    - 整型
      - 短整型 (short int)
      - 整型 (int)
      - 长整型 (long int)
      - 长长整型(long long int)
      - 每种int都有signed与unsigned(包括char型)
    - 字符型(char)
    - 浮点型
      - 单精度型 (float)
      - 双精度型 (double)
      - 长双精度型(long double)
    - 布尔型(bool)
  - 派生类型
    - 指针类型 (*)
    - 枚举类型 (enum)
    - 数组类型 ([])
    - 结构体类型(struct)
    - 共用体类型(union)
    - 类类型 (class)
    - 可以用基本数据类型来构造派生类型
  - 空类型(void)

§7. 结构体、类和对象

7.2. 结构体类型

7.2.1. 引入

将不同性质类型但是互相有关联的的数据放在一起，组合成一种新的复合型数据类型，称为结构体类型（简称结构体）

★ 将描述一个事物的各方面特征的数据组合成一个有机的整体，说明数据之间的内在关系

例1：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出

```
int main()
{   int num, age;
    char sex, name[20], addr[30];
    float score;
    cin >> num ... ;
    ...
    cout << sex ...;
    return 0;
}
```

1个学生的6方面信息：
用6个彼此完全独立的不同类型的变量来表达

缺点：访问时无整体性

例2：键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出

```
const int N=100;
int main()
{   int num[N], age[N], i;
    char sex[N], name[N][20], addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
        cin >> num[i] ... ;
        ...
        cout << sex[i] ...;
        }
}
```

100个学生的6方面信息：
用6个彼此完全独立的不同类型的数组变量来表达

缺点：1. 访问时无整体性
2. 访问同一个人时，不同数组的下标必须对应

说明：
这3个例子都是非结构体方式，主要看缺点

例3：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出，要求以指针方式操作

```
int main()
{   int num, age, *p_num=&num, *p_age=&age;
    char sex, name[20], addr[30];
    char *p_sex=&sex, *p_name=name, *p_addr=addr;
    float score, *p_score=&score;
    cin >> *p_num ... ;
    ...
    cout << *p_sex ...;
    return 0;
}
```

学号	姓名	性别	年龄	课程成绩	家庭住址
1001	张三	男	18	80.5	上海市杨浦区***
1002	李四	女	18	76	黑龙江省齐齐哈尔市***
1003	王五	女	19	90.5	四川省宜宾市***
1004	赵六	男	17	88	陕西省汉中市***
...	...	...	...	...	...

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.1. 结构体类型声明的形式

```
struct 结构体名 {
    结构体成员1（类型名   成员名）
    ...
    结构体成员n（类型名   成员名）
};  （带分号）
```

```
struct student {
  int   num;
  char  name[20];
  char  sex;
  int   age;
  float score;
  char  addr[30];
};
```

★ 结构体成员也称为结构体的数据成员

★ 结构体名,成员名命名规则同变量

★ 同一结构体的成员名不能同名,但可与其它名称(其它结构体的成员名,其它变量名等)相同

```
struct x1 {     struct x2 {     struct x3 {     int main()      int fun()       void num()
    int num;        int num;        float num;  {               {               {
    ...             ...             ...             long num;       int num[5];     ...
};              };              };              }               }               }
```

★ 每个成员的类型可以相同，也可以不同

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.1. 结构体类型声明的形式

★ 每个成员的类型既可以是基本数据类型，也可以是已存在的自定义数据类型

```
struct date {
    int year;
    int month;
    int day;
};
struct student {
    int  num;
    char name[20];
    char sex;
    struct date birthday;
    float score;
    char addr[30];
};
```

struct date必须在struct student的前面定义,否则无法知道birthday占多少字节(编译器思维)

```
struct student {
    int  num;
    char name[20];
    char sex;
    struct student monitor;
    float score;
    char addr[30];
};
```

★ 每个成员的类型不允许是自身的结构体类型
原因：套娃式定义导致无法确定 monitor 占多少个字节

★ 每个成员的类型不允许是自身的结构体类型

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

★ 结构体类型的大小为所有成员的大小的总和，可用sizeof(struct 结构体名) 计算，但不占用具体的内存空间
(结构体类型不占空间，结构体变量占用一段连续的空间)

★ C的结构体只能包含数据成员，C++还可以包含函数(后续模块)

int i; sizeof(int)得4
但int型不占空间，i占4字节

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：为保证CPU的运算稳定和效率，要求基本数据类型在内存中的存储地址必须对齐，即基本数据类型的变量不能简单的存储于内存中的任意地址处，该变量的起始地址必须是该类型大小的整数倍

例：1、32位编译系统下，int型数据的起始地址是4的倍数，short型数据的起始地址是2的倍数，double型数据的起始地址是8的倍数，指针变量的起始地址是4的倍数

2、64位编译系统下，指针变量的起始地址是8的倍数

结构体的成员对齐：

★ 结构体类型的起始地址，必须是所有数据成员中占最大字节的基本数据类型的整数倍

★ 结构体类型的所有数据成员的大小总和，必须是所有数据成员中占最大字节的基本数据类型的整数倍，因此结构体类型最后可能会有填充字节

★ 结构体类型中各数据成员的起始地址，必须是该类型大小的整数倍，因此结构体成员之间可能会有填充字节

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

```
//例1：结构体声明与字节对齐
#include <iostream>
using namespace std;
struct s1 {
    int   a;
    short b;
    char  c;
};
struct s2 {
    short b;
    int   a;
    short c;
};
struct s3 {
    int a;
    short b;
    short c;
};
int main()
{
   cout << sizeof(s1) << endl;
   cout << sizeof(s2) << endl;
   cout << sizeof(s3) << endl;
}
```

理论：7字节
实际：8字节

理论：8字节
实际：12字节

理论：8字节
实际：8字节

8字节

假设无填充字节，
则读a需要两个CPU
时钟周期

=>节约4字节，
速度慢一半

1、起始地址必须是4的倍数
2、总大小必须是4的倍数
3、每个成员的起始地址必须是1/2/4的倍数

Microsoft Visual Studio 调试控制台

```
8
12
8
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

```
//例2：结构体声明与字节对齐
#include <iostream>
using namespace std;
struct s1 {
    char   a;
    int    b;
    double c;
};
struct s2 {
    char   a;
    double c;
    int    b;
};
int main()
{
   cout << sizeof(s1) << endl;
   cout << sizeof(s2) << endl;
}
```

理论：13字节
实际：16字节

理论：13字节
实际：24字节

1、起始地址必须是8的倍数
2、总大小必须是8的倍数
3、每个成员的起始地址必须是1/4/8的倍数

Microsoft Visual Studio 调试控制台

```
16
24
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.2. 结构体类型的声明

7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：

结构体的成员对齐：

```
//例3：结构体声明与字节对齐
#include <iostream>
using namespace std;
struct s1 {
    char a[5];
    char b[3];
    int  c;
};
struct s2 {
    char a[5];
    int  c;
    char b[3];
};
int main()
{
   cout << sizeof(s1) << endl;
   cout << sizeof(s2) << endl;
}
```

理论：12字节
实际：12字节

理论：12字节
实际：16字节

推论：
1、s2中，a数组越界3字节/b数组越界1字节，不会导致系统错误
2、s2中，a数组越界4 ~ 11字节，会影响c/b的取值，但不会导致系统错误
=> 越界错误的后果不可预料
(包括不体现出错误)

1、起始地址必须是4的倍数
2、总大小必须是4的倍数
3、每个成员的起始地址必须是1/4的倍数

Microsoft Visual Studio 调试控制台

```
12
16
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.3. 结构体变量的定义及初始化

7.2.3.1. 先定义结构体类型，再定义变量

```
struct student {
    ...
};
struct student s1;
struct student s2[10];
struct student *s3;
```

★ 关键字struct(阴影部分)在C中不能省，在C++中可省略

★ 结构体变量占用实际的内存空间，根据变量的不同类型(静态/动态/全局/局部)在不同区域进行分配，遵守各自的初始化规则

7.2.3.2. 在定义结构体类型的同时定义变量

```
struct student {
    ...
} s1, s2[10], *s3;
struct student s4;
```

★ 可以再次用7.2.3.1的方法定义新的变量

§7. 结构体、类和对象

7.2. 结构体类型

7.2.3. 结构体变量的定义及初始化

7.2.3.3. 直接定义结构体类型的变量(结构体无名)

```
struct {
    ...
} s1, s2[10], *s3;
```

★ 因为结构体无名，因此无法再用7.2.3.1的方法进行新的变量定义
(适用于仅需要一次性定义的地方)

7.2.3.4. 结构体变量定义时初始化

```
student s1={1,"张三", 'M', 20, 78.5, "上海"};
```

```
struct student {
  int   num;
  char  name[20];
  char  sex;
  int   age;
  float score;
  char  addr[30];
};
```

★ 按各成员依次列出

★ 若嵌套使用，要列出最低级成员

```
student s1={1,"张三", 'M',{1982,5,9},78.5};
```

内{}可省
但不建议

```
struct date {
   int year;
   int month;
   int day;
};
```

```
struct student {
   int  num;
   char name[20];
   char sex;
   struct date birthday;
   float score;
};
```

★ 可用一个同类型变量初始化另一个变量

```
student s1={1,"张三", 'M',{1982,5,9},78.5};
student s2=s1;
```

内{}可省
但不建议

§7. 结构体、类和对象

7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.1. 形式

变量名.成员名

★ .称为成员运算符（附录D 优先级第2组，左结合）

★ C中最高，C++中次高

```
struct student {
    int   num;
    char  name[20];
    char  sex;
    int   age;
    float score;
    char  addr[30];
} s1;
```

```
s1.num = 1;
strcpy(s1.name,"张三");
s1.sex = 'M';
s1.age = 20;
s1.score = 76.5;
strcpy(s1.addr,"上海");
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.1. 形式

7.2.4.2. 使用

★ 结构体变量允许进行整体赋值操作

student s1={...}, s2;　　用一个同类型变量初始化另一个变量：

s2=s1;　//赋值语句　　student s1={...}, s2=s1; //定义时初始化

★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员

int i, *p;	student s1;　int *p;	
i++;	s1.num++;	自增/减
...+ i*10 +...;	... + s1.num*10 +...;	各种表达式
if (i>=10)	if (s1.num>=10)	
p = &i;	p = &s1.num;	取地址
scanf("%d", &i);	scanf("%d",&s1.num);	输入
cout << i;	cout << s1.num;	输出
fun(i);	fun(s1.num);	函数实参
return i;	return s1.num;	返回值

§7. 结构体、类和对象

7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.2. 使用

★ 结构体变量允许进行整体赋值操作

★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员

★ 若嵌套使用，只能对最低级成员操作

```
struct date {
    int year;
    int month;
    int day;
    };
struct student {
    int  num;
    char name[9];
    char sex;
    struct date birthday ;
    float score;
    };
```

```
s1.birthday.year=1980;
cin >> s1.birthday.month;
cout << s1.birthday.day;
```

★ 结构体变量不能进行整体的输入和输出操作

```
student s1={...};
cin >> s1;    ×
cout << s1;   ×
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.2. 使用

例1：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出(前面例子的对比)

6个独立变量

```
int main()
{
    int num;
    int age;
    char sex;
    char name[20]
    char addr[30];
    float score;

    cin >> num ... ;
    ...
    cout << sex ...;
    return 0;
}
```

1个变量的6个成员

```
struct student {
    ...;
};

int main()
{
    struct student s1;

    cin >> s1.num ... ;
    ...
    cout << s1.sex ...;
    return 0;
}
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.1. 含义

一个数组，数组中的元素是结构体类型

7.2.5.2. 定义

```
struct 结构体名 数组名[正整型常量表达式]
struct 结构体名 数组名[正整型常量表达式1][正整型常量表达式2]
```

★ 包括整型常量、整型符号常量和整型只读变量

```
struct student s2[10];
struct student s4[10][20];
```

7.2.5.3. 定义时初始化

内{}可省
但不建议

```
struct student s2[10] = { {1, "张三", 'M', 20, 78.5, "上海"},
                          {2, "李四", 'F', 19, 82, "北京"},
                          {..},{..},{..},{..},{..},{..},{..},{..} };
```

★ 其它同基本数据类型数组的初始化

（占用空间、存放、下标范围、初始化时省略大小）

§7. 结构体、类和对象

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.4. 使用

数组名[下标].成员名

```
s2[0].num=1;
cin  >> s2[0].age >> s2[0].name;
cout << s2[1].age << s2[1].name;
s2[2].name[0] = 'A'; //注意两个[]的位置
```

例2：键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出(前面例子对比)

```
const int N=100;

int main()
{   int num[N], age[N], i;
    char sex[N];
    char name[N][20]
    char addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
       cin >> num[i] ... ;
       ...
       cout << sex[i] ...;
       }
}
```

6个独立的大小为100的数组变量

```
const int N=100;
struct student {
    ...;
};
int main()
{   int i;
    struct student s2[N];
    for(i=0; i<N; i++) {
      cin >> s2[i].num ... ;
      ...
      cout << s2[i].sex ...;
      }
    return 0;
}
```

1个大小为100的数组，每个元素有6个成员

§7. 结构体、类和对象

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.4. 使用

例：现有Li/Zhang/Sun三个候选人，键盘输入10个候选人的名字，统计每个人的得票

```
#include <iostream>
using namespace std;

struct Person {
    char name[20];
    int  count;
};
int main()
{   struct Person leader[3]={"Li", 0, "Zhang", 0, "Sun", 0};
    int i, j;
    char leader_name[20];
    for(i=0; i<10; i++) {
        cin >> leader_name;//一维数组不带下标，表示串方式输入(≤19)
        for(j=0; j<3; j++)
            if (!strcmp(leader_name, leader[j].name)) //严格大小写
                leader[j].count++;
       } //end of for(i)
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" << leader[i].count << endl;
    return 0;
}
```

可改进的地方：
1、3/10/20应该用宏定义或常变量
2、下面的语句可优化效率

```
if(!strcmp(leader_name, leader[j].name)){
    leader[j].count++;
    break;
    }
```

运行效率高，避免比较成功后再做不必要的比较

Microsoft

```
Zhang
Li
Li
Sun
Zhang
Sun
Sun
Li
Sun
Sun

Li:3
Zhang:2
Sun:5
```

```
#include <iostream>
using namespace std;

struct Person {
    string name; //变化，用string替代一维字符数组
    int    count;
};
int main()
{   Person leader[3]={"Li", 0, "Zhang", 0, "Sun", 0};
    int i, j;
    string leader_name;
    for(i=0; i<10; i++) {
        cin >> leader_name; //可输入任意长度字符串
        for(j=0; j<3; j++)
            if (leader_name == leader[j].name) //直接用==进行比较
                leader[j].count++;
        } //end of for(i)
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" << leader[i].count << endl;
    return 0;
}
```

换string后：
长度不受限、比较运算较简单，但不影响整个程序的处理逻辑

§7. 结构体、类和对象

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

含义：存放结构体变量的地址

7.2.6.1. 结构体变量的地址与结构体变量中成员地址

student s1;

&s1　　：结构体变量的地址

（基类型是结构体变量，+1表示一个结构体长度）

&s1.age ：结构体变量中某个成员的地址

（基类型是该成员的类型,+1表示一个成员长度）

```
struct student {
    int    num;
    char   name[20];
    char   sex;
    int    age;
    float  score;
    char   addr[30];
};
```

&s1.age　　=> 2028

(&s1.age)+1 => 2032

有填充(实际)

&s1　　=> 2000

&s1+1 => 2068

```
#include <iostream>
using namespace std;

struct student {
    int    num;
    char   name[20];
    char   sex;
    int    age;
    float  score;
    char   addr[30];
};

int main()
{
    student s1;

    cout << sizeof(s1)          << endl;
    cout << &s1                 << endl;
    cout << &s1.num             << endl;
    cout << (void *)s1.name     << endl;
    cout << (void *)(&s1.sex)   << endl;
    cout << &s1.age             << endl;
    cout << &s1.score           << endl;
    cout << (void *)s1.addr     << endl;

    return 0;
}
```

在多编译器下运行本程序，观察：
1、s1的大小是否是4的倍数
2、s1的起始地址是否4的倍数
3、s1中每个数据成员的地址是否其自身类型的整数倍

Microsoft

```
68
0053FD24
0053FD24
0053FD28
0053FD3C
0053FD40
0053FD44
0053FD48
```

1、为什么部分转void
2、为什么部分无&

§7. 结构体、类和对象

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

含义：存放结构体变量的地址

7.2.6.1. 结构体变量的地址与结构体变量中成员地址

student s1;

&s1　　：结构体变量的地址

（基类型是结构体变量，+1表示一个结构体长度）

&s1.age ：结构体变量中某个成员的地址

（基类型是该成员的类型,+1表示一个成员长度）

7.2.6.2. 结构体指针变量的定义

struct 结构体名 *指针变量名

struct student s1, *s3;

int *p;

s3=&s1;　　结构体变量的指针

s3的值为2000，++s3后值为2068

p=&s1.age;　结构体变量成员的指针

p的值为2028，++p后值为2032

（注:不要说指向score，应该说不再指向age）

```
#include <iostream>
using namespace std;

struct student {
    int   num;
    char  name[20];
    char  sex;
    int   age;
    float score;
    char  addr[30];
};

int main()
{
    struct student s1;
    cout << &s1 << endl;                    地址X
    cout << &s1+1 << endl;                  地址X + 68

    cout << &s1.num << endl;                地址X
    cout << &s1.num+1 << endl;              地址X + 4

    cout << (void *)(&s1.sex) << endl;      地址Y(X+24)
    cout << (void *)(&s1.sex+1)<<endl;      地址Y + 1

    cout << &s1.age << endl;                地址Z(Y+1+3)
    cout << &s1.age+1 << endl;              地址Z + 4
    return 0;
}
```

Microsoft

```
010FFC50
010FFC94
010FFC50
010FFC54
010FFC68
010FFC69
010FFC6C
010FFC70
```

§7.结构体、类和对象

7.2.结构体类型

7.2.6.指向结构体变量的指针

7.2.6.1.结构体变量的地址与结构体变量中成员地址

7.2.6.2.结构体指针变量的定义

7.2.6.3.使用

(*指针变量名).成员名

指针变量名->成员名 ⇔ (*指针变量名).成员名

★ -> 称为间接成员运算符（附录D 优先级第2组，左结合）

```
struct student s1, *s3=&s1;
cout << s1.num    << s1.name    << s1.sex;
cout << (*s3).num << (*s3).name << (*s3).sex;
cout << s3->num   << s3->name   << s3->sex;
```

s3->age++;　值后缀++

++s3->age;　值前缀++

§7. 结构体、类和对象

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

7.2.6.3. 使用

例3：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出，要求以指针方式操作(前例)

```
int main()
{
    int num, age;
    char sex,name[20],addr[30];
    float score;
    int *p_num=&num;
    int *p_age=&age;
    char *p_sex=&sex;
    char *p_name=name;
    char *p_addr=addr;
    float *p_score=&score;
    cin >> *p_num ... ;
    ...
    cout << *p_sex ...;
    return 0;
}
```

6个值变量
6个指针变量
分别指向

```
struct student {
    ...;
};

int main()
{
    struct student  s1;
    struct student *s3;
    s3 = &s1;
    cin >> s3->num ... ;
    ...
    cout << s3->sex ...;
    return 0;
}
```

1个值变量
1个指针变量
指向6个成员

§7. 结构体、类和对象

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

7.2.6.4. 指向结构体数组的指针

```
struct student s2[10], *p;
p = s2;            ✓
p = &s2[0];        ✓
p = &s2[0].num;    × 指针的基类型不匹配
p = (struct student *)&s2[0].num;  ✓ 强制类型转换
```

各种表示形式：

(*p).num	：取p所指元素中成员num的值
p->num	：...
p[0].num	：...
p+1	：取p指元素的下一个元素的地址
(*(p+1)).num	：取p指向的元素的下一个元素的num值
(p+1)->num	：...
p[1].num	：...
(p++)->num	：保留p的旧值到临时变量中，p++后指向下一元素，再取p旧值所指元素的成员num的值
(++p)->num	：p先指向下一个元素，再取p所指元素的成员num的值
p->num++	：取p所指元素中成员num的值，值++

§7. 结构体、类和对象

7.2. 结构体类型

7.2.7. 结构体数据类型作为函数参数

7.2.7.1. 形参为结构体简单变量

★ 对应实参为结构体简单变量/数组元素

```
void fun(struct student s)
{
    ...;
}
int main()
{
    struct student s1, s2[10];
    struct student s3[3][4];
    ...
    fun(s1);
    fun(s2[4]);
    fun(s3[1][2]);
    return 0;
}
```

```
void fun(int s)
{
    ...;
}
int main()
{
    int s1, s2[10];
    int s3[3][4];
    ...
    fun(s1);
    fun(s2[4]);
    fun(s3[1][2]);
    return 0;
}
```

7.2.7.2. 形参为结构体变量的指针

★ 对应实参为结构体简单变量的地址/一维数组名

```
void fun(struct student *s)
{        struct student s[]
    ...;
}
int main()
{
    struct student s1, s2[10];
    ...
    fun(&s1);
    ...
    fun(s2);
    return 0;
}
```

```
void fun(int *s)
{        int s[]
    ...;
}
int main()
{
    int s1, s2[10];
    ...
    fun(&s1);
    ...
    fun(s2);
    return 0;
}
```

7.2.7.3. 形参为结构体的引用声明

★ 对应实参为结构体简单变量

```
void fun(struct student &s)
{
    ...;
}

int main()
{
    struct student s1;
    ...
    fun(s1);
    ...
    return 0;
}
```

```
void fun(int &s)
{
  ...;
}

int main()
{
    int s1;
    ...
    fun(s1);
    ...
    return 0;
}
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

- 函数内部:
  - 可定义结构体类型的各种变量/成员级访问
- 函数外部:
  - 从定义点到本源程序文件的结束前:
    - 可定义结构体类型的各种变量/成员级访问
  - 其它位置(本源程序定义点前/其它源程序):
    - 有该结构体的提前声明:
      - 仅可定义指针及引用/整体访问
    - 有该结构体的重复定义:
      - 可定义结构体类型的各种变量/成员级访问

类似外部全局变量概念,但不完全相同

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况一：定义在函数内部

```
#include <iostream>
using namespace std;

void fun(void)
{  struct student {
       int  num;
       char name[20];
       char sex;
       int age;
       float score;
       };
  struct student s1, s2[10], *s3;
  s1.num = 10;
  s2[4].age = 15;
  s3 = &s1;
  s3->score = 75;
  s3 = s2;
  (s3+3)->age = 15;
}
```

正确

```
int main()
{
    struct student s;
    s.age = 15;

    return 0;
}
```

不正确

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况二：定义在函数外部,从定义点到本源程序结束前

```
#include <iostream>
using namespace std;
struct student {
    int  num;
    char name[20];
    char sex;
    int age;
    float score;
};
void f1(void)
{
  struct student s1, s2[10], *s3;
  s1.num = 10;
  s2[4].age = 15;
  s3 = &s1;
  s3->score = 75;
  s3 = s2;
  (s3+3)->age = 15;
}
```

都正确

```
void f2(struct student *s)
{
    s->age = 15;
}
struct student f3(void)
{
    struct student s;
    ...
    return s;
}
int main()
{
    struct student s1, s2;
    f1();
    f2(&s1);
    s2 = f3();
    return 0;
}
```

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况三：ex1.cpp和ex2.cpp构成一个程序，无提前声明

/* ex1.cpp */
#include <iostream>
using namespace std;
 	
void f1()
{
　　不可定义/使用student型各种变量　　×
}
struct student {
　　...;
};
int fun()
{
　可定义student型各种变量,访问成员　　√
}
int main()
{
　可定义student型各种变量,访问成员　　√
}
	/* ex2.cpp */
	#include <iostream>
	using namespace std;
	
	int f2()
	{
	　不可定义/使用student型各种变量　　×
	}

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况四：ex1.cpp和ex2.cpp构成一个程序，有提前声明

```
/* ex1.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
 可定义student型各种变量,访问成员
}
```

允许

不允许

```
/* ex2.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明

void f2()
{
   struct student *s1;

   struct student s3, &s2=s3;

   s1.age = 15;
}
```

允许

不允许

虽可定义指针/引用,但不能进行成员级访问，无意义

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况四(变化1)：ex1.cpp和ex2.cpp构成一个程序，有提前声明

```
/* ex1.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
 可定义student型各种变量,访问成员
}
```

允许

不允许

```
/* ex2.cpp */
#include <iostream>
using namespace std;

void f2()
{
   struct student *s1;
}
void f3()
{
   struct student; //结构体声明

   struct student *s1;
   s1->age = 15;
}
```

不允许

允许

不允许

虽可定义指针/引用,但不能进行成员级访问，无意义

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体见后)

情况五：ex1.cpp和ex2.cpp构成一个程序，有重复定义

```
/* ex1.cpp */
#include <iostream>
using namespace std;

struct student {  //结构体定义
    ...;
};

int fun()
{
    可定义/使用student型各种变量   √
}

int main()
{
    可定义/使用student型各种变量   √
}
```

```
/* ex2.cpp */
#include <iostream>
using namespace std;

struct student {  //结构体定义
    ...;
};

int f2()
{
   可定义/使用student型各种变量   √
}
```

本质上是两个不同的结构体struct student，因此即使不完全相同也能正确，这样会带来理解上的偏差

§7. 结构体、类和对象

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用(续7.2.2)

★ 结构体类型的定义既可以放在函数外部，也可以放在函数内部(具体见后)

问题：如何在其它位置访问定义和使用结构体？

```
/* ex.h */
struct student {  //结构体定义
    ...;
};
```

```
/* ex1.cpp */
#include <iostream>
#include "ex.h"
using namespace std;

int fun()
{
    可定义/使用student型各种变量   √
}
int main()
{
    可定义/使用student型各种变量   √
}
```

```
/* ex2.cpp */
#include <iostream>
#include "ex.h"
using namespace std;

int f2()
{
    可定义/使用student型各种变量   √
}
```

解决方法：在头文件中定义

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.1. 类的引入

★ 使用结构体带来的好处

★ 能否使结构体的表达更清晰易懂

例2：键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出

```
const int N=100;

int main()
{   int num[N], age[N], i;
    char sex[N];
    char name[N][20]
    char addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
       cin >> num[i] ... ;
       ...
       cout << sex[i] ...;
       }
}
```

6个独立的大小为100的数组变量

```
const int N=100;
struct student {
    ...;
};
int main()
{   int i;
    struct student s2[N];
    for(i=0; i<N; i++) {
      cin >> s2[i].num ... ;
      ...
      cout << s2[i].sex ...;
      }
    return 0;
}
```

1个大小为100的数组，每个元素有6个成员

```
const int N=100;
struct student {
    ...;
};

//形参类型为引用*2/指针*1
//无特殊含义，仅表示均可用
void input(student &stu)
{   //输入某学生的6个成员
}
void output(student &stu)
{   //输出某学生的6个成员
}
void grade(student *stu)
{   //根据成绩打印等级
}
int main()
{   struct student s1, s2[N];
    input (s2[17]);
    output(s1);
    grade (&s2[23]);
    grade (&s1)
}
```

公共函数，通过传入不同元素的值/地址来访问各元素

```
const int N=100;
struct student {
    ...;
};

//同cin.good()/cout.put()形式
//称这种形式的函数为成员函数
void ...input(...)
{   //输入某学生的6个成员
}
void ...output(...)
{   //输出某学生的6个成员
}
void ...grade(...)
{   //根据成绩打印等级
}
int main()
{   struct student s1, s2[N];
    s2[17].input(...);
    s1.output(...);
    s2[23].grade(...);
    s1.grade(...);
}
```

改为这种形式，是否可读性更好？更容易理解？

第07模块例，从左到右依次是：

1、6个独立变量

2、1个结构体变量有6个成员，在同一个函数中实现输入/输出/计算等不同功能（用相同下标控制对同一变量的访问）

3、1个结构体变量有6个成员，用不同公共函数传不同变量指针/引用方式实现（用相同下标控制对同一变量的访问）

4、(期望)1个结构体变量有6个成员，用成员.成员函数()方式实现

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.1. 类的引入

在结构体只包含数据成员的基础上，引入成员函数的概念，使结构体同时拥有数据成员和成员函数

7.3.2. 声明类类型

```
class student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

```
struct student {
    int num;
    char name[20];
    char sex;
};
```

★ 类类型的使用与结构体的使用方法基本相同

§7. 结构体、类和对象

7.2.2. 结构体类型的声明

7.2.2.1. 结构体类型声明的形式

所有“结构体”替换为“类”，均有效

★ 结构体成员也称为结构体的数据成员

★ 结构体名，成员名命名规则同变量

★ 同一结构体的成员名不能同名，但可与其它名称(其它结构体的成员名，其它变量名等)相同

★ 每个成员的类型可以相同，也可以不同

★ 每个成员的类型既可以是基本数据类型，也可以是已存在的自定义数据类型

★ 每个成员的类型不允许是自身的结构体类型

★ 结构体类型的定义既可以放在函数外部，也可以放在函数内部(具体见后)

不含成员函数

★ 结构体类型的大小为所有数据成员的大小的总和，可以用sizeof(struct 结构体名)计算，但不占用具体的内存空间(结构体类型不占空间，结构体变量占用一段连续的空间)

★ C的结构体只能包含数据成员，C++还可以包含函数(后续模块)

7.2.2.2. 结构体类型声明与字节对齐

内存对齐的基本概念：为保证CPU的运算稳定和效率，要求基本数据类型在内存中的存储地址必须对齐，即基本数据类型的变量不能简单的存储于内存中的任意地址处，该变量的起始地址必须是该类型大小的整数倍

结构体的成员对齐：

★ 结构体类型的起始地址，必须是所有数据成员中占最大字节的基本数据类型的整数倍

★ 结构体类型的所有数据成员的大小总和，必须是所有数据成员中占最大字节的基本数据类型的整数倍，因此结构体类型最后可能会有填充字节

★ 结构体类型中各数据成员的起始地址，必须是该类型大小的整数倍，因此结构体成员之间可能会有填充字节

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.2. 声明类类型

★ 用sizeof(类名)计算类的大小时，成员函数不占用空间

★ 通过类的成员访问限定符(private/public)，可以指定成员的属性是私有(private)或公有(public)，私有成员不能被外部函数访问，公有成员可被外部函数所访问，具体可由实际应用需求决定

- 内部函数：该class的成员函数
- 外部函数：其它函数

注意：和函数部分的静态函数/外部函数的概念有差别!!!

- 类的成员访问限定符是限制"外部函数"的访问，类的"内部函数"不受限定符的限制
- 建议数据成员private，成员函数public

★ 在类的定义中，private/public出现的顺序，次数无限制

```
class student {
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
  private:
    int num;
    char name[20];
    char sex;
};
```

class做为一个整体，不必考虑对数据成员的访问在数据成员的定义之前

```
class student {
  private:
    int num;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
  private:
    char name[20];
    char sex;
};
```

```
class student {
  private:
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

本例：无论三个数据成员的限定符是什么；无论display函数的限定符是什么；都不影响display函数对三个数据成员的访问

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.3. 对象的定义和访问

结构体类型 --实例化--> 变量

类 --实例化--> 对象

★ 含义相同，称呼不同

7.3.3.1. 先定义类，再定义对象

```
class student {
    ...
    };
student s1;
student s2[10];
student *s3;
```

```
struct student {
    ...
    };
struct student s1;
struct student s2[10];
struct student *s3;
```

★ 结构体变量/类对象占用实际的内存空间，根据不同类型(静态/动态/全局/局部)在不同区域进行分配

7.3.3.2. 在定义类的同时定义对象

```
class student {
    ...
    } s1, s2[10], *s3;
student s4;
```

```
struct student {
    ...
    } s1, s2[10], *s3;
struct student s4;
```

★ 可以再次用7.3.3.1的方法定义新的变量/类对象

7.3.3.3. 直接定义对象(类无名)

```
class {
    ...
    } s1, s2[10], *s3;
```

```
struct {
    ...
    } s1, s2[10], *s3;
```

★ 因为结构体/类无名，因此无法再用7.3.3.1的方法进行新的变量/对象定义

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.4. 类与结构体的比较

★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能

```
class student {
  private:
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

在.cpp中，替换为struct，功能完全相同
在.c中则报语法错误

demo.c

demo-c (全局范围)

```
1   #include <stdio.h>
2
3   struct student {
4     private:
5         int num;
6         char* name;
7     };
8
9   int main()
10  {
11      return 0;
12  }
```

```
demo.c(4,1): error C2061: 语法错误: 标识符“private”
demo.c(7,1): error C2059: 语法错误: “}”
```

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.4. 类与结构体的比较

★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能

★ 若不指定成员访问限定符，则struct缺省为public，class缺省为private

```
#include <iostream>
using namespace std;

struct student {
    int num;
    char name[20];
    char sex;
    void display() {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
        }
};

int main()
{
    student s1;
    s1.num = 1001;
    return 0;
}
```

全部public，
外界(main)可访问，
与C相比，多成员函数

```
#include <iostream>
using namespace std;

class student {
    int num;
    char name[20];
    char sex;
    void display() {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
        }
};

int main()
{
    student s1;
    s1.num = 1001;
    return 0;
}
```

全部private，
外界(main)不可访问，编译错

```
(17,7): error C2248: “student::num”: 无法访问 private 成员(在“student”类中声明)
(5): message : 参见“student::num”的声明
(4): message : 参见“student”的声明
```

```
class student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
    }
};
```

全部是private

```
struct student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
    }
};
```

全部是public

```
class student {
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        ...
    }
};
```

私有

公有

```
struct student {
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        ...
    }
};
```

公有

公有

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.5. 对象成员的访问

7.3.5.1. 通过对象名访问对象中的成员

7.3.5.2. 通过指向对象的指针访问对象中成员

7.3.5.3. 通过对象的引用来访问对象中的成员

```
class student {
  private:
    int name[20];
    char sex;
  public:
    int num;
    void display()
    {
        ...
    }
};
```

类定义

★ 通过对象名访问对象中的成员

```
int main()
{
    student s1, s2[10];
    s1.sex = 'm';        ×
    s1.num=10001;        ✓
    s1.display();        ✓
    s2[0].sex = 'f';     ×
    s2[0].num=10002;     ✓
    s2[3].display();     ✓
}
```

①

★ 通过指向对象的指针来访问对象中的成员

```
int main()
{
    student s1, *s3=&s1;
    s1.num     = 10001; ✓
    (*s3).num  = 10001; ✓
    s3->num    = 10001; ✓
    s1.display();       ✓
    (*s3).display();    ✓
    s3->display();      ✓
}
```

②

★ 通过对象的引用来访问对象中的成员

```
int main()
{
    student s1, &s3=s1;
    s1.num = 10001; ✓
    s3.num = 10001; ✓
    s1.display();   ✓
    s3.display();   ✓
}
```

③

★ 注意访问权限，只能是public

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.5. 对象成员的访问

7.3.5.4. 访问规则

★ 只能访问公有的数据成员和成员函数

★ 数据成员可出现在其基本类型允许出现的任何地方（外部需公有）

int i, *p;	student s1; int *p;	
i++;	s1.num++;	自增/减
...+ i*10 +...;	... + s1.num*10 +...;	各种表达式
if (i>=10)	if (s1.num>=10)	
p = &i;	p = &s1.num;	取地址
scanf("%d", &i);	scanf("%d",&s1.num);	输入
cout << i;	cout << s1.num;	输出
fun(i);	fun(s1.num);	函数实参
return i;	return s1.num;	返回值

★ 成员函数的参数传递规则仍为实参单向传值到形参（引用仍为别名）

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.1. 成员函数的实现

体内实现：class中给出成员函数的定义及实现过程

```
class student {
  ...
  public:
    void display()
    {
        cout<<"num:" <<num <<endl;
        cout<<"name:"<<name<<endl;
        cout<<"sex:" <<sex <<endl;
    }
};
```

体外实现：class中给出成员函数的定义，class外部(class后)给出成员函数的实现

★ 函数实现时需要加类的作用域限定符

★ 即使类成员函数是体外实现方式，仍然算"内部函数"，不受private/public访问限定符限制!!!

```
class student {
  public:
    void display();
};

void student::display()
{
    cout << "num:"  <<num  <<endl;
    cout << "name:" <<name <<endl;
    cout << "sex:"  <<sex  <<endl;
}
```

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

★ 对应类的成员函数(类函数)，一般的普通函数称为全局函数

★ 成员函数的定义、实现及调用时参数传递的语法规则与全局函数相同

★ 成员函数也受类的成员访问限定符的约束，只有公有的成员函数可以被"外部函数"调用

★ 私有和公有的成员函数均可以访问/调用本类的所有数据成员/成员函数，不受private/public的限制

(再次强调：private/public是用来限制外部函数对类数据成员/成员函数的访问)

```
class test {
  private:
    int a;
    int f1();
  public:
    int b;
    int f2();
    int f3();
};
```

```
int test::f1()
{   a=10;   ✓
    b=15;   ✓
    f2();   ✓
}
int test::f2()
{   ...
}
int test::f3()
{   a=20;   ✓
    b=25;   ✓
    f1();   ✓
}
```

```
int main()
{
    test t1;

    t1.a=10;   ×
    t1.f1();   ×
    t1.b=15;   ✓
    t1.f2();   ✓
    t1.f3();   ✓
}
```

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

★ 全局函数与成员函数可以同名，按照低层屏蔽高层的原则进行，也可以通过全局作用域符(::级别最高)强制访问高层

```
class test {
  ...
  public:
    int fun();
    int f1();
};

int fun()  全局函数
{
    ...
}
```

```
int test::fun()  类函数
{
    ...
}

int test::f1()
{
    fun();     类函数
    ::fun();   全局函数
}
```

成员函数内部

```
int main()
{
    test t1;

    t1.fun();  类函数
    fun();     全局函数
}
```

全局函数中表示不会冲突

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.2. 成员函数的性质

★ 全局函数与成员函数可以同名，按照低层屏蔽高层的原则进行，也可以通过全局作用域符(::级别最高)强制访问高层（类的数据成员与全局变量也遵循此强制访问规则）

```
int a;       //全局变量
void fun()
{
   int a;    //局部变量
   a=10;     //访问局部变量
   ::a=15;   //访问全局变量(第04模块时说不行)
}            //这种方式仅适合C++，纯C不支持
```

```
int a; //全局变量

class test {
   ...
   public:
      int a;  //类数据成员
      int f1();
};

int test::f1()
{
   a=10;     //类数据成员
   ::a=15;   //全局变量
}
```

```
int a; //全局变量
class test {
   ...
   public:
      int a;    //类数据成员
      int f1();
   };

int test::f1()
{
  int a;       //成员函数内的自动变量
  a=5;         //自动变量
  test::a=10;  //类数据成员
  ::a=15;      //全局变量
}

int main()
{
   test t1;
   t1.f1();
                                //外部无法访问f1自动变量
   cout << t1.a << endl;        //类数据成员(需public)
   cout << a << endl;           //全局变量
}
```

注：极端情况可能出现部分成员无法访问的情况，不建议深究

```
int a;  //全局变量

class test {
   public:
      int a;    //类数据成员
};

int test::f1()
{
   int a;        //成员函数的自动变量
   if (***) {
      long a;    //if语句内的自动变量
      for (***) {
         short a;  //for循环内的自动变量

         全局a/test类a/函数内a/if中a/for中a，
         共5个，如何在此处访问不同a?
         } //end of for
      } //end of if
}
```

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.6. 类的成员函数

7.3.6.3. 成员函数的存储方式

★ 每个类的实例对象仅包含数据成员(sizeof(类)=所有数据成员之和)，根据不同的定义位置占用不同的数据空间(静态数据区或动态数据区)

★ 类的成员函数占用函数(代码)区，每个类的每个成员函数(包括体内实现和体外实现)只占用一段空间，所有该类的对象共用成员函数的代码空间

★ 当通过对象调用成员函数时，系统会缺省设置一个隐含的this指针，指向被调用的对象，并以此来区分成员函数对数据成员的访问

```
class student {
  private:
    int num;
  public:
    void set(int n) {
      num = n;
      }
    void display() {
      cout << num << endl;
      }
};
```

```
int main()
{  student s1, s2;
   s1.set(10);
   s2.set(15);
   s1.display();
   s2.display();
   return 0;
}
```

10

15

s1,s2占用不同的4字节
为什么 s1.set / s2.display 时会指向不同的4字节

```
class student {
  private:
    int num;
  public:
    void set(student *this, int n) {
      this->num = n;
      }
    void display(student *this) {
      cout << this->num << endl;
      }
};
```

类成员函数中隐含了一个this指针
调用时会隐含传入调用对象的地址
s1.set(10) ⇔ s1.set(&s1, 10);
s2.display()⇔s2.display(&s2);

注：this不能显式写在函数声明中，但可显式访问

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

★ 公有函数可被外界调用，称为类的公共/对外接口通过对象.公用函数(实参表)的方法进行调用，将函数称为方法，将调用过程称为消息传递

★ 如果允许外界直接改变某个数据成员的值，可直接设置属性为public(不提倡)

★ 其它不愿公开的数据成员和成员函数可设置为私有,对外部隐蔽，但仍可通过公有函数进行访问及修改

```
class student {
  private:
    int num;
  public:
    void set(int n)
    {
        num = n;
    }
    void display()
    {
        cout << num << endl;
    }
};
```

```
int main()
{
  student s1, s2;
   s1.set(10);
   s2.set(15);
   s1.display(); 10
   s2.display(); 15

   return 0;
}
```

set/display函数均间接访问了私有成员num

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

★ 公有函数的形参称为提供给外部的访问接口，在形参的数量、类型、顺序不变的情况下，私有成员的变化及公有函数实现部分的修改不影响外部的调用

①

```
class student {
  private:
    int num;
  public:
    void set(int n)
    {   num = n;
    }
    void display()
    {   cout << num << endl;
    }
};
```

②

```
class student {
  private:
    int xh;
  public:
    void set(int n)
    {   xh = n;
    }
    void display()
    {   printf("%d\n", xh);
    }
};
```

③

```
class student {
  private:
    int xh;
  public:
    void set(int n)
    {   xh = (n>=0 ? n:0);
    }
    void display()
    {   printf("%d\n", xh);
    }
};
```

```
int main()
{
   student s1, s2;
   s1.set(10);
   s2.set(15);
   s1.display(); 10
   s2.display(); 15
   return 0;
}
```

假设class student由乙编写
main函数由甲编写
则：乙用三种方法
甲的程序均不需要变化

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

应用实例1：

谷歌的Android 4.x内核(假装C++)

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见

```
  void show(char *图片名)
```

函数实现，不可见

```
};
```

***公司的游戏软件

```
int main()
{
    ....
    picture p1;
    p1.show(文件名);
    ....
}
```

谷歌公司的Android 12内核

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见
可能已进行过很大调整

```
  void show(char *图片名)
```

函数实现，不可见
实现过程可能与2.3完全不同

```
};
```

谷歌称12.0的显示速度
经优化后比4.x快**%
用户程序不需要变化

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

应用实例2：

A公司的乙团队：V1.0版本

```
class translation {
```

类的私有数据成员
及成员函数
外界不可见

```
  void trans(char *英文)
```

函数功能为输出中文
具体实现过程不可见

```
};
```

A公司的甲团队

```
int main()
{
    ....
    translation t1;
    t1.trans("****");
    ....
}
```

A公司的乙团队：V1.1版本

```
class translation {
```

类的私有数据成员
及成员函数
外界不可见
可能已进行过很大调整

```
  void trans(char *英文)
```

函数实现，不可见
实现过程可能与1.0完全不同

```
};
```

V1.1比V1.0的翻译结果
更准确，更贴切
用户程序不需要变化
两个团队能同时工作

思考：

1、甲乙两个团队哪个更不可替代？

2、你的职业期望是哪个团队？

3、进入不同团队对不同知识的要求？

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明(*.h)与类成员函数的实现(*.cpp)分开

例1：假设程序由ex1.cpp、ex2.cpp和ex.h共同构成

```
/* ex.h */
class student {
  private:
    数据成员1;
    ...;
    数据成员n;
  public:
    成员函数1;
    ...;
    成员函数2;
}
```

```
/* ex2.cpp */
#include <iostream>
#include "ex.h"
using namespace std;
main及其它函数的实现
```

```
/* ex1.cpp */
#include <iostream>
#include "ex.h"
using name space std;

返回值 student::成员函数1()
{
    成员函数1的实现;
}

...

返回值 student::成员函数n()
{
    成员函数n的实现;
}
```

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明(*.h)与类成员函数的实现(*.cpp)分开

例2：数学函数sqrt

```
/* math.h */
double sqrt(double x);
```

```
/* 实现sqrt的源码 math.cpp */

sqrt的具体实现过程被隐藏
提供lib/dll（静态/动态库）
```

```
/* test.cpp */
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
    cout << sqrt(2) << endl;
    return 0;
}
```

h+lib/dll 即可实现相同功能

§7. 结构体、类和对象

7.3. 类和对象的基本概念

7.3.7. 类的封装性和信息隐蔽

7.3.7.1. 公有接口和私有实现的分离

7.3.7.2. 类声明和成员函数定义的分离

★ 将类的声明(*.h)与类成员函数的实现(*.cpp)分开

★ 在需要外部调用的地方，只要提供声明部分即可，类的实现可通过库文件(*.lib)或动态链接库(*.dll)的方式提供，而不必提供实现的源码

★一个程序包含多源程序文件的方法已掌握

★ 建立库文件/动态链接库的方法有兴趣可自学

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.1. 引入及作用

- 构造函数：Constructor Function，用于类对象生成时的初始化
- 析构函数：Destructor Function，用于类对象消失时的收尾工作

7.3.3中

结构体类型 --实例化--> 变量

类 --实例化--> 对象

★ 含义相同，称呼不同

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.2. 对象的初始化

对象的初值：在静态数据区分配的对象，数据成员初值为0；

在动态数据区分配的对象，数据成员的初值随机

（与普通变量相同）

对象的初始化方法：

(1) 若全部成员都是公有，可按结构体的方式进行初始化

```
#include <iostream>
using namespace std;

class Time {
   public:
     int f2();  //函数不占用对象空间
     int hour;
     int minute;
     int sec;
};

int main()
{
   Time t1={14,15,23};
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

Microsoft Visual Studio 调试控制台

```
14:15:23
```

以下两种形式均报错：

```
Time t1(14,15,23);
Time t1=(14,15,23);
```

```
#include <iostream>
using namespace std;

class Time {
   public:
     int f2();  //函数不占用对象空间
     int hour;
     int minute;
     int sec;
};

int main()
{
   Time t1=(14,15,23);
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

error C2440: “初始化”: 无法从“int”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确

以下两种形式均报错：

```
Time t1(14,15,23);
Time t1=(14,15,23);
```

```
#include <iostream>
using namespace std;

class Time {
   public:
     int f2();  //函数不占用对象空间
     int hour;
     int minute;
     int sec;
};

int main()
{
   Time t1(14,15,23);
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

error C2440: “初始化”: 无法从“initializer list”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.2. 对象的初始化

对象的初值：在静态数据区分配的对象，数据成员初值为0；

在动态数据区分配的对象，数据成员的初值随机

（与普通变量相同）

对象的初始化方法：

(1) 若全部成员都是公有，可按结构体的方式进行初始化（若有私有成员，不能用此方法）

```
#include <iostream>
using namespace std;

class Time {
   public:
     int f2();  //函数不占用对象空间
     int hour;
     int minute;
     int sec;
};

int main()
{
   Time t1={14,15,23};
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

以下两种形式均报错：
Time t1(14,15,23);
Time t1=(14,15,23);

Microsoft Visual Studio 调试控制台
14:15:23

```
#include <iostream>
using namespace std;

class Time {
   public:
     int hour;
     int minute;
   private:
     int sec;
     int f2();  //函数不占空间
  };
int main()
{
   Time t1={14,15,23};
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

error C2440: “初始化”: 无法从“initializer list”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确

```
#include <iostream>
using namespace std;

class Time {
     int f1(); //缺省私有
   public:
     int hour;
     int minute;
     int sec;
};

int main()
{
   Time t1={14,15,23};
   cout << t1.hour << ':' << t1.minute
                    << ':' << t1.sec << endl;
}
```

只要所有数据成员均公有即可

Microsoft Visual Studio 调试控制台
14:15:23

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.2. 对象的初始化

对象的初始化方法：

(1) 若全部成员都是公有，可按结构体的方式进行初始化(若有私有成员，不能用此方法)

(2) 写一个赋初值的公有成员函数，在其它成员被调用之前进行调用

```
class Time {
    private:
        int hour;
        int minute;
        int sec;
    public:
        void set(int h, int m, int s)
        {    hour=h;
             minute=m;
             sec=s;
         }
    };
int main()
{   Time t;

    t.set(14,15,23);
    t.其它
}
```

(3) 声明类时对数据成员进行初始化

（C++11标准支持，目前双编译器均可）

```
class Time {
    public:
        int hour=0;
        int minute=0;
        int sec=0;
    };
```

不同对象的值被统一初始化，无法个性化

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

引入：完成对象的初始化工作，对象建立时被自动调用

形式：与类同名，无返回类型(非void，也不是缺省int)

```
#include <iostream>
using namespace std;

class Time {
   private:
      int hour, minute, sec;  //相同类型可以写在一行上
   public:
      Time()
      {   hour=0;
          minute=0;
          sec=0;
      }
      void display()
      {   cout << hour << ':' <<minute << ':' << sec << endl;
      }
};

int main()
{
    Time t; //t的三个成员都是0
    t.display();
}
```

体内实现

Microsoft Visual Studio 调试控制台

```
0:0:0
```

```
#include <iostream>
using namespace std;

class Time {
   private:
      int hour, minute, sec;  //相同类型可以写在一行上
   public:
      Time();
      void display()
      {   cout << hour << ':' <<minute << ':' << sec << endl;
      }
};
Time::Time()
{   hour=0;
    minute=0;
    sec=0;
}
int main()
{
    Time t; //t的三个成员都是0
    t.display();
}
```

体外实现

Microsoft Visual Studio 调试控制台

```
0:0:0
```

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

引入：完成对象的初始化工作，对象建立时被自动调用

形式：与类同名，无返回类型(非void，也不是缺省int)

使用：

★ 对象建立时被自动调用

★ 构造函数必须公有

★ 若不指定构造函数，则系统缺省生成一个构造函数，形式为无参空体

★ 若用户定义了构造函数，则缺省构造函数不再存在

★ 构造函数既可以体内实现，也可以体外实现

★ 允许定义带参数的构造函数，以解决无参构造函数初始化各对象的值相同的情况(个性化初值)

```
#include <iostream>
using namespace std;

class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int h, int m, int s);
      void display()
      {  cout << hour << ':' << minute << ':' << sec << endl;
      }
   };
Time::Time(int h, int m, int s)
{
   hour   = h;
   minute = m;
   sec    = s;
}
```

也允许体内实现

```
int main()
{
    Time t1(14,15);
    Time t2;
}
```

```
cpp-demo.cpp(24,19): error C2661: "Time::Time": 没有重载函数接受 2 个参数
cpp-demo.cpp(25,10): error C2512: "Time": 没有合适的默认构造函数可用
cpp-demo.cpp(4,7): message : 参见"Time"的声明
```

```
int main()
{
   Time t1(14,15,23);
   Time t2{15,16,24};
   Time t3={16,17,25};
   t1.display();
   t2.display();
   t3.display();
}
```

三种形式均可

Microsoft Visual Studio 调试控制台

```
14:15:23
15:16:24
16:17:25
```

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用：

★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化

体内实现

```
class Time {
   private:
     int hour;
     int minute;
     int sec;
   public:
     Time(int h, int m, int s): hour(h),minute(m),sec(s)
     {
     }
};
```

成员名

参数名

允许写其它语句，若没有，则函数体为空

h初始化hour
m初始化minute
s初始化sec

体外实现

```
class Time {
   private:
     int hour;
     int minute;
     int sec;
   public:
     Time(int h, int m, int s);
};
Time::Time(int h, int m, int s):hour(h),minute(m),sec(s)
{
}
```

成员名

参数名

允许写其它语句，若没有，则函数体为空

h初始化hour
m初始化minute
s初始化sec

★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化(仅适用于简单的赋值)

```
Time::Time(int h, int m, int s)
{
    if (h>=0 && h<=23)
        hour = h;
    else
        hour = 0;
    ....
}
```

无法通过参数初始化表实现

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用：

★ 允许定义带参数的构造函数，以解决无参构造函数初始化各对象的值相同的情况(个性化初值)

引申问题：构造函数如何做到更自由的个性化？

例：

```
int main()
{
    Time t1(14);        //期望是14:00:00
    Time t2(14,15);     //期望是14:15:00
    Time t3(14,15,23);  //期望是14:15:23
}
```

```
//方案1：用三个构造函数实现不同功能
=> 构造函数的名称是类名
=> 三个不同函数名称相同
Time(int h)
{
    hour   = h;
    minute = 0;
    sec    = 0;
}

Time(int h,int m)
{
    hour   = h;
    minute = m;
    sec    = 0;
}

Time(int h,int m,int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

```
//方案2：用一个构造函数实现，
         通过形参有默认值的方式来实现

Time(int h, int m = 0, int s = 0)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

调用时，如果m/s有值，则用给定值，否则用默认值

PPT作业中已出现过
cin.get()
cin.getline()

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用：

★ 构造函数允许重载

```
class Time {
   ...
   public:
      Time();
      Time(int h, int m, int s);
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t(14,15,23); //正确
   Time t2;          //正确
   ...
}
```

也可以体内实现

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用：

★ 构造函数允许带默认参数，但要注意可能与重载产生二义性冲突

```
class Time {
   ...
   public:
      Time();
      Time(int h, int m, int s=0);
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t1(14,15,23); //正确
   Time t2(14,15);    //正确
   Time t3;           //正确
}
```

无参与带缺省参数的重载，
不冲突
适应带0/2/3个参数的情况

```
class Time {
   ...
   public:
      Time();
      Time(int h=0, int m=0, int s=0);
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t1(14,15,23); //正确
   Time t2(14,15);    //正确
   Time t3(14);       //正确
   Time t4;           //错误
}
```

无参与带缺省参数的重载，
冲突!!!

```
cpp-demo.cpp(25): error C2668: “Time::Time”: 对重载函数的调用不明确
cpp-demo.cpp(14,5): message : 可能是“Time::Time(int,int,int)”
cpp-demo.cpp(8,5): message : 或       “Time::Time(void)”
cpp-demo.cpp(25,12): message : 尝试匹配参数列表“()”时
```

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.3. 构造函数的引入及使用

使用：

★ 构造函数也可以显式调用，一般用于带参构造函数

```
class Test {
  private:
    int a;
  public:
    Test(int x) {
      a=x;
      }
};
Test fun()
{  ...
   return Test(10);  //显式
}

int main()
{
   Test t1(10);      //隐式
   Test t2=Test(10); //显式
   Test t3=Test{10}; //显式
}
```

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.4. 析构函数

引入：在对象被撤销时(生命期结束)时被自动调用，完成一些善后工作(主要是内存清理)，但不是撤销对象本身

形式：

~类名();

★ 无返回值(非void, 也不是int)，无参，不允许重载

```
class Time {
   ...
   public:
      Time()   //构造体内实现
      {  ...
      }

      ~Time() //析构体内实现
      {  ...
      }
};
```

```
class Time {
   ...
   public:
      Time();   //构造声明
      ~Time(); //析构声明
};

Time::Time()   //构造体外实现
{  ...
}

Time::~Time() //析构体外实现
{  ...
}
```

使用：

★ 对象撤销时被自动调用，用户不能显式调用

★ 析构函数必须公有

★ 若不指定析构函数，则系统缺省生成一个析构函数, 形式为无参空体

★ 若用户定义了析构函数，则缺省析构函数不再存在

★ 析构函数既可以体内实现，也可以体外实现

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数(动态内存申请为后续课程内容，此处不再展开)

§7. 结构体、类和对象

7.4. 构造函数与析构函数

7.4.5. 构造函数与析构函数的调用时机

构造函数：

★ 自动对象(形参)　　：函数中变量定义时

★ 静态局部对象　　：第一次调用时

★ 静态全局/外部全局对象：程序开始时

~~★ 动态申请的对象　　：~~后续课程课内容(略)

main开始前

析构函数：

★ 自动对象(形参)　　：函数结束时

★ 静态局部对象　　：程序结束时(在全局之前)

★ 静态全局/外部全局对象：程序结束时

~~★ 动态申请的对象　　：~~后续课程内容(略)

main结束后

§7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
void fun()
{   Time t1;
    cout << "addr:" << &t1 << endl;
    cout << "fun" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
main begin
Time Begin
addr:地址a
fun
Time End
continue
Time Begin
addr:地址a(同上)
fun
Time End
main end
```

1、函数调用时分配空间
结束时回收空间
2、函数多次调用则多次
分配/回收空间

验证了函数模块遗留的什么问题？

§7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
void fun()
{   static Time t1;
    cout << "fun" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
main begin
Time Begin
fun
continue
fun
main end
Time End
```

1、函数第1次调用时分配
2、后续函数调用不分配
3、全部程序结束后回收

验证了函数模块遗留的什么问题？

§7. 结构体、类和对象

7.4.5. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
Time t1;
void fun()
{   cout << "fun begin" <<endl;
    cout << "fun end" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
Time begin
main begin
fun begin
fun end
main end
Time End
```

§7. 结构体、类和对象

7.5. 对象数组

7.5.1. 形式

类型 对象名[整型常量表达式] ：一维数组

类型 对象名[整型常量表达式1][整型常量表达式2] ：二维数组

```
Time t[10];
Time s[3][4];
```

7.5.2. 定义对象时进行初始化

★ 若未定义构造函数或构造函数无参，则按简单对象使用无参构造函数的规则进行

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        void display()
        {  cout << hour << ':' <<minute << ':' << sec << endl;
        }
};

int main()
{   Time t[10];
    for (int i=0; i<10; i++)
        t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
-858993460:-858993460:-858993460
```

10个元素的三个成员的值都是随机的，因为调用缺省构造，什么也没做

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time() { hour=0;  minute=0;  sec=0;}
        void display()
        {  cout << hour << ':' <<minute << ':' << sec << endl;
        }
};

int main()
{
     Time t[10];
     for (int i=0; i<10; i++)
         t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
0:0:0
```

10个元素的三个成员的值都是0，因为调用无参构造

§7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 若带参构造函数只带一个参数，可用数组定义时初始化的方法进行

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h)
         {
             hour   = h;
             minute = 0;
             sec    = 0;
         }
        void display()
        {
            cout << hour << ':' <<minute << ':' << sec << endl;
        }
};

int main()
{
     Time t[10] ={1,2,3,4,5, 6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中hour=1-10,其它两个为0
调用一个参数的构造

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour, minute, sec;
    public:
        Time()
        {
            hour   = 0;
            minute = 0;
            sec    = 0;
        }
        Time(int h)
        {   hour   = h;
            minute = 0;
            sec    = 0;
        }
        void display()
        {
            cout << hour << ':' <<minute << ':' << sec << endl;
        }
};
int main()
{    Time t[10] ={1,2,3,4,5, 6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中hour=1-10,其它两个为0
两个构造用一个参数的

§7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 若带参构造函数有带一个参数和多个参数共存(可以是带默认参数的构造函数)，则可用数组定义时初始化的方法进行，每个数组元素只传一个参数

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time()
        {   hour=0;  minute=0;  sec=0;
        }
        Time(int h)
        {   hour=h;  minute=0;  sec=0;
        }
        Time(int h,int m)
        {   hour=h; minute=m; sec=0;
        }
        void display()
        {   cout << hour << ':' <<minute << ':' << sec << endl;
        }
};
int main()
{
    Time t[10] ={1,2,3,4,5, 6,7,8,9,10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

无参1、2重载

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中hour=1-10,其它两个为0
多个构造用一个参数的

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h=0,int m=0, int s=0)
        {
            hour   = h;
            minute = m;
            sec    = s;
        }
        void display()
        {   cout << hour << ':' <<minute << ':' << sec << endl;
        }
};

int main()
{
    Time t[10] ={1,2,3,4,5, 6,7,8,9,10};
    for (int i=0; i<10; i++)
        t[i].display();
}
```

带默认参数的构造函数
可带0/1/2/3个参数

Microsoft Visual Studio 调试控制台

```
1:0:0
2:0:0
3:0:0
4:0:0
5:0:0
6:0:0
7:0:0
8:0:0
9:0:0
10:0:0
```

10个元素的三个成员中hour=1-10,其它两个为0
调用带一个参数的构造

§7. 结构体、类和对象

7.5. 对象数组

7.5.2. 定义对象时进行初始化

★ 如果希望初始化时多于一个参数，则初始化时显式给出构造函数及实参表

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour;
        int minute;
        int sec;
    public:
        Time(int h=0,int m=0, int s=0)
        {   hour=h; minute=m; sec=s;
        }
        void display()
        {   cout << hour << ':' <<minute << ':' << sec << endl;
        }
};
int main()
{    Time t[10] ={Time(1,2,3),Time(4,5),6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

Microsoft Visual Studio 调试控制台

```
1:2:3     ... t[0] 三参构造
4:5:0     ... t[1] 两参构造
6:0:0     ... t[2] 一参构造
7:0:0
8:0:0
9:0:0
10:0:0    ... t[7] 零参构造
0:0:0
0:0:0
0:0:0
```

问：以下两种的差别在哪里？

```
Time t[10] ={ {1,2,3}, {4,5}, 6,7,8,9,10 };
Time t[10] ={ (1,2,3), (4,5), 6,7,8,9,10 };
```

★ 初始化的数量不能超过数组大小

★ 定义数组时可不定义大小，有初始化表决定

```
Time t[10]={Time(1,2,3),Time(4,5),6,7,8,9,10};
```

不能比7小

```
Time t[]={Time(1,2,3),Time(4,5),6,7,8,9,10};
```

自动为7

§7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

形式：

类名 *指针变量名

Time *t;

指针的赋值：

Time t1, *t;　　Time t1, *t=&t1;

t = &t1;

定义后赋值语句赋值　　定义时赋初值

Time t2[10], *t;　　Time t2[10], *t=t2;

t = t2;　　t指向t2[0]，t++则指向t2[1]

t++ ⇔ t+sizeof(time)

class Time换成 int
class Time换成 struct student，方法相同

§7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

使用：

class Time换成 struct student，方法相同，
但要注意成员访问限定，外部仅限public

Time类定义

```
class Time {
   private:
      int minute;
      int sec;
   public:
      int hour; //公有
      Time(int h=0, int m=0, int s=0);
      ~Time();
      void display();
   };
```

指向简单变量的指针

```
Time t1, *t=&t1;

 t  : t1对象的地址
 *t : t1对象
 (*t).hour       ⇔ t->hour       ⇔ t1.hour;
 (*t).display() ⇔ t->display() ⇔ t1.display()
```

指向数组变量的指针

```
Time t2[10], *t=t2;

 t  : t2数组的第[0]个对象的地址
 *t : t2数组的第[0]个对象

 (*t).hour       ⇔ t->hour       ⇔ t2[0].hour
 (*t).display() ⇔ t->display() ⇔ t2[0].display()

 t+3    : t2数组的第[3]个对象的地址
 *(t+3) : t2数组的第[3]个对象

 (*(t+3)).hour       ⇔ t[3].hour       ⇔ (t+3)->hour ⇔       t2[3].hour
 (*(t+3)).display() ⇔ t[3].display() ⇔ (t+3)->display() ⇔ t2[3].display()
```

§7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

7.6.2. 指向对象成员的指针

7.6.2.1. 指向对象的数据成员的指针

定义：数据成员的基类型 *指针变量名

赋值：指针变量名 = 数据成员的地址

```
Time t1;
int *p;
p=&t1.hour;
```

class Time换成 struct student，方法相同，
但要注意成员访问限定，外部仅限public

使用：

```
*p ⇔ t1.hour;
```

★ 对象的数据成员必须是public

7.6.2.2. 指向对象的成员函数的指针(后续课程内容，略)

§7. 结构体、类和对象

7.6. 对象指针

7.6.1. 指向对象的指针

7.6.2. 指向对象成员的指针(其中：指向成员函数的指针 略)

7.6.3. this指针

含义：指向当前被访问的成员函数所对应的对象的指针，名称固定为this，基类型为类名

```
void Time::display()
{
    cout << hour   << endl;
    cout << minute << endl;
    cout << sec    << endl;
}
```

相当于

```
void Time::display(const Time *this)
{
    cout << this->hour   << endl;
    cout << this->minute << endl;
    cout << this->sec    << endl;
} //编译会错，只是含义上相当于!!!
```

```
Time t1, t2;
t1.display() 时，this指向t1
            ⇔ t1.display(&t1);
t2.display() 时，this指向t2
            ⇔ t2.display(&t2);
```

```
void Time::set(int h, int m, int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

相当于

```
void Time::set(const Time *this, int h, int m, int s)
{
    this->hour   = h;
    this->minute = m;
    this->sec    = s;
} //编译会错，只是含义上相当于!!!
```

```
Time t1, t2;
  t1.set(14,15,23) 时，this指向t1
            ⇔ t1.set(&t1,14,15,23);
  t2.set(16,30,0) 时，this指向t2
            ⇔ t2.set(&t2,16,30,0);
```

§7. 结构体、类和对象

7.6. 对象指针

7.6.3. this指针

含义：指向当前被访问的成员函数所对应的对象的指针，名称固定为this，基类型为类名

使用：

★ 隐式使用，相当于通过对象调用成员函数时传入该对象的自身的地址

★ 也可以显式使用(但不能显式定义)

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h, int m, int s)
        {
            hour = h;
            minute = m;
            sec = s;
        }
        void display()
        {
            cout << this->hour << ':' << this->minute
                                << ':' << this->sec << endl;
        }
};
int main()
{   Time t1(12, 13, 24);
    t1.display();
}
```

隐式使用

显式使用

Microsoft Visual Studio 调试控制台

```
12:13:24
```

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h, int m, int s)
        {
            hour = h;
            minute = m;
            sec = s;
        }
        void display(const Time *this)
        {
            cout << this->hour << ':' << this->minute
                                << ':' << this->sec << endl;
        }
};
int main()
{
    Time t1(12, 13, 24);
    t1.display(&t1);
}
```

```
error C2143: 语法错误: 缺少“)”(在“this”的前面)
error C2143: 语法错误: 缺少“;”(在“this”的前面)
error C2059: 语法错误:“this”
error C2059: 语法错误:“)”
error C2334: “{”的前面有意外标记；跳过明显的函数体
```

不能显式定义

```
//特殊约定，this不能显式，其它名字可以
void display(const Time *that)
{
    cout << that->hour << ':' << this->minute
                        << ':' << this->sec << endl;
}
```

§7. 结构体、类和对象

7.7. 对象的赋值与复制

7.7.1. 对象的赋值

含义：将一个对象的所有数据成员的值对应赋值给另一个已存在对象的数据成员

形式：类名 对象名1，对象名2;

...

对象名1=对象名2;　　//执行语句的方式

```
Time t1(14,15,23), t2;

t2=t1;
```

★ 两个对象属于同一个类，且不能在定义时赋值

★ 系统默认的赋值操作是将右对象的全部数据成员的值对应赋给左对象的全部数据成员(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现(通过=运算符的重载实现，后续课程内容)

§7. 结构体、类和对象

7.7. 对象的赋值与复制

7.7.2. 对象的复制

含义：建立一个新对象，其值与某个已有对象完全相同

使用：

类 对象名(已有对象名)

类 对象名=已有对象名

两种形式
本质一样

```
Time t1(14,15,23), t2(t1), t3=t1;
```

★ 与对象赋值的区别：定义语句/执行语句中

```
Time t1(14,15,23), t2, t3=t1; //复制，t3为新对象
t2 = t1; //赋值，t2为已有对象
```

★ 系统默认的复制操作是将已有对象的全部数据成员的值对应赋给新对象的全部数据成员(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现(通过重定义复制/拷贝构造函数来实现，后续课程内容)

§7. 结构体、类和对象

7.8. 友元

7.8.1. 引入

当在外部访问对象时，private全部禁止，public全部允许，为使应用更灵活，引入友元(friend)的概念，允许友元访问private部分

★ 友元不是面向对象的概念，它破坏了数据的封装性，但方便使用，提高了运行效率

问题：在全局函数display(外部)中如何访问私有成员？

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      ...
};

void display(Time t)
{
    想访问 t.hour;
}
```

方法1：通过公有函数间接访问

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      int get_hour() { return hour; }
      void set_hour(int h) { hour = h; }
      ...
};

void display(Time t)
{
    通过 t.get_hour() 读
    通过 t.set_hour(12) 赋值
}
```

缺点：当频繁调用时，效率较低

方法2：成员直接公有

```
class Time {
   public:
      int hour;
      int minute;
      int sec;
   public:
      ...
};

void display(Time t)
{
    直接 t.hour;
}
```

缺点：所有外部函数都能访问
不仅局限于一个display()
失去了类的封装和隐蔽性

§7. 结构体、类和对象

7.8. 友元

7.8.1. 引入

可以成为类的友元的成分：

★ 全局函数

★ 其它类的成员函数

★ 其它类

友元的声明方式：

在类的声明中，相应要成为友元的函数/类前加friend关键字即可

§7. 结构体、类和对象

7.8. 友元

7.8.2. 声明全局函数为友元函数

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
      friend void display(Time &t);
   public:
      ...
   };
```

全局函数

```
void display(Time &t)
{
  cout << t.hour ; ✓
}
```

```
void fun(Time &t)
{
  cout << t.hour ; ×
}
```

```
void main()
{
    Time t1;
    display(t1);
}
```

★ 不能直接写成员名，要通过对象来调用
因为不是成员函数，没有this指针

★ 声明友元的位置不限private/public

§7. 结构体、类和对象

7.8. 友元

7.8.3. 声明其它类的成员函数为友元函数

`class Time;` ← 在test中引用Time时，Time尚未定义，因此要提前声明	~~`class Time;`~~ ← 如无提前声明，test中每个Time前加class也可以（数量多时不方便）

`class test {`
`  public:`
`  void display(Time &t);`
`  };`
	`class test {`
	`  public:`
	`  void display( class Time &t);`
	`  };`

`class Time {`
`  private:`
`  int hour;`
`  ...`
`  friend void test::display(Time &t);`
`  };`
	`class Time {`
	`  private:`
	`  int hour;`
	`  ...`
	`  friend void test::display(Time &t);`
	`  };`

`void test::display(Time &t)`
`{`
`  cout << t.hour << ... << endl;`
`}`
	`void test::display(Time &t)`
	`{`
	`  cout << t.hour << ... << endl;`
	`}`

§7. 结构体、类和对象

7.8. 友元

7.8.3. 声明其它类的成员函数为友元函数

```
class Time;
```

在test中引用Time时，Time尚未定义因此要提前声明

```
class test {
   public:
      void display(Time &t);
   };
```

```
class Time {
   private:
      int hour;
      ...
      friend void test::display(Time &t);
   };
```

声明友元不限定private/public
但友元函数所在类要符合限定规则

```
void test::display(Time &t)
{
   cout << t.hour << ... << endl;
}
```

//成员.对象方式访问

§7. 结构体、类和对象

7.8. 友元

7.8.4. 友元类

★ 提前声明遵循刚才的原则

```
class test;//有提前声明
class Time {
   private:
      ...
      friend test;
   };
class test {
   ...
   };
```

test的所有成员函数都可以访问Time的私有成员

```
class test; //无提前声明
class Time {
   private:
      ...
      friend class test;
   };
class test {
   ...
   };
```

★ 友元是单向而不是双向的

本例中：Time中不能访问test的私有

★ 友元不可传递

```
class A {
   friend class B;
};
class B {
   friend class C;
};
class C {
   C不能访问A的私有成员
};
```

★ C++规定同类的不同对象互为友元

```
class Student {
   private:
      int num;
   public:
      void display();
};
void Student::display()
{  Student s;
   ...
   if (this->num > s.num) {...}
}
```

互为友元

§7. 结构体、类和对象

7.9. 构造函数初始值列表

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

赋值：先初始化再赋值

```
ConstRef::ConstRef(int ii)
{
    i = ii;    //正确
    ci = ii;   //错误：不能给const赋值
    ri = i;    //错误：ri没被初始化
}
//错误：ci和ri必须被初始化
```

§7. 结构体、类和对象

7.9. 构造函数初始值列表

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

初始化：直接初始化数据成员

```
ConstRef::ConstRef(int ii): i(ii), ci(i), ri(ii) { }
//正确：显式的初始化引用和const成员
```

结论：如果成员是const、引用或者属于某种未提供默认构造函数的类类型，必须通过构造函数初始值列表为这些成员提供初值

§7. 结构体、类和对象

7.9. 构造函数初始值列表

成员初始化的顺序：与类定义中出现的顺序一致，跟初始化列表中的顺序无关

建议1：构造函数初始值的顺序与成员声明的顺序保持一致

建议2：尽量避免使用某些成员初始化其他成员

默认实参和构造函数：

不能为构造函数的全部形参都提供默认实参

```
//接受string的构造函数
class Sales_data{
    Sales_data (strings = ""): bookNo(s) { }
};
//接受istream&参数的构造函数
class Sales_data{
    Sales_data (istream &is = cin) { is >> *this; }
};
```

上例：错误。不提供任何实参的创建类对象时，产生二义性。

§7. 结构体、类和对象

7.10. 委托构造函数

委托构造函数使用它所属类的其他构造函数执行自己的初始化过程，或者说它把自己的一些(或全部)职责委托给了其他的构造函数。

委托构造函数也有一个成员初始值的列表和一个函数体。在委托构造函数内，成员初始值列表只有唯一的一个入口，就是类名本身。和其他成员初始值一样，类名后面紧跟圆括号括起来的参数列表，参数列表必须与类中另外一个构造函数匹配。

```
class Sales_data {
public:
    //非委托构造函数接受三个实参，使用这些实参初始化数据成员，然后结束工作
    Sales_data(string s, unsigned cnt, double price):
        bookNo(s), units_sold(cnt), revenue(cnt*price) { }
    //其余构造函数全部委托给另一个构造函数
    Sales_data():Sales_data(s,0,0) { }
        //定义默认构造函数令其使用三参数的构造函数完成初始化过程
    Sales_data(string s):Sales_data(s,0,0) { }
        //定义接收一个string的构造函数，同样委托给了三参数版本
    Sales_data(istream &is):Sales_data() { read(is, *this); }
        //定义接受istream&的构造函数，它委托给了默认构造函数，默认构造函数接着委托给三参数的构造函数
        //当接受委托的构造函数执行完后，接着执行istream&构造函数体的内容，即调用read函数读取给定的istream
private:
     string bookNo; unsigned units_sold; double revenue;
};
```

§7. 结构体、类和对象

7.11. 默认构造函数的作用

默认初始化发生的情况：

在块作用域内不使用任何初始值定义一个非静态变量或数组

类本身含有类类型的成员并且使用合成的默认构造函数

当类类型的成员没有在构造函数初始值列表中显示的初始化

值初始化发生的情况：

数组初始化的过程中如果提供的初始值少于数组的大小

不使用初始值定义一个局部静态变量

通过书写形如T()的表达式显式地请求值初始化时，T是类型名

类必须包含一个默认构造函数以便在上述情况下使用

合成构造函数：

如果用户定义的类中没有显式的定义任何构造函数，编译器才会自动为该类型生成默认构造函数，称为合成的构造函数

§7. 结构体、类和对象

7.12. 隐式的类类型转换

转换构造函数：

当一个构造函数只有一个参数，而且该参数又不是本类的const引用时，这种构造函数称为转换构造函数

转换构造函数的作用是将一个其它类型的数据转换成一个类的对象

注意转换构造函数只能有一个参数，如果有多个参数就不是转换构造函数

```
class Sales_data {
private:
    string book_no;
    unsigned units_sold = 1;
    double revenue = 1.0;
public:
    Sales_data() = default; //不接受任何实参，默认构造函数
    Sales_data(const string& s) : book_no(s) { } //类型转换构造函数
    Sales_data(const string& s, unsigned n, double p) : book_no(s), units_sold(n), revenue(p* n) { }
    Sales_data(istream&) { }
    Sales_data& combine(const Sales_data&);
    //其他成员函数...
};
```

```
string null_book = "9-999-999-9"; //构造一个临时的Sales_data对象item
item.conbine(null_book);
```

§7. 结构体、类和对象

7.12. 隐式的类类型转换

只允许一步类类型转换：

```
Sales_data(const string& s) : book_no(s) { } //类型转换构造函数
Sales_data& combine(const Sales_data&);
item.combine("9-999-999-9");
//错误：需要用户定义的两种转换："9-999-999-9"到string到Sales_data
item.combine(string("9-999-999-9"));
//正确：显式地转换成string，隐式地转换成Sales_data
item.combie(Sales_data("9-999-999-9"));
//正确：隐式地转换成string，显式地转换成Sales_data
```

类类型转换不是总有效：

item.combine(cin)隐式地将cin转换成Sales_data，这个转换执行接受一个istream的Sales_data构造函数，该构造函数通过读取标准输入创建了一个临时的Sales_data对象，随后将得到的对象传递给combine。该对象是一个临时量，一旦combine完成就不能再访问它了。

```
Sales_data(istream &) { }
Sales_data& combine(const Sales_data&);
```

§7. 结构体、类和对象

7.12. 隐式的类类型转换

抑制构造函数定义的隐式类型转换：

将构造函数声明为explicit可以阻止构造函数的隐式类型转换

```
explicit Sales_data(const std::string &s):bookNo(s) { }
explicit Sales_data(std::istream&);
```

此时：

```
item.combine(null_book); //错误
item.combine(cin);       //错误
```

关键字explicit只对一个实参的构造函数有效，需要多个实参的构造函数不能用于隐式转换，无须将构造函数指定为explicit

只能在类内声明构造函数时使用explicit，类外定义时不应重复

当使用explicit声明构造函数时，将只能以直接初始化的形式使用

```
Sales_data item1(null_book);  //正确，直接初始化
Sales_data item2 = null_book; //错误，不能将explicit构造函数用于拷贝形式的初始化过程
```

§7. 结构体、类和对象

7.12. 隐式的类类型转换

为转换显式地使用构造函数：

```
explicit Sales_data(const std::string &s):bookNo(s) { }
explicit Sales_data(std::istream&);
Sales_data& combine(const Sales_data&);
```

```
item.combine(Sales_data(null_book));
//直接使用Sales_data的构造函数，该调用通过接受string的构造函数创建一个临时的Sales_data对象
item.combine(static_cast<Sales_data>(cin));
//使用_cast执行了显式的转换，使用istream构造函数创建了一个临时static的Sales_data对象
```

§7. 结构体、类和对象

7.13. 枚举

7.13.1. 含义

如果一个变量的取值有限且离散，那么将这个变量所有可能的取值一一列举出来，并限定在某个集合内(将变量的值一一列举)

7.13.2. 声明形式

enum 枚举类型名 {枚举(元素)常量取值表};

(以标识符形式表示的整型量，表示枚举类型的取值集合)

声明枚举类型时，系统给每一个枚举元素一个指定的整数值（序号），默认从0开始，依次加1

```
enum weekday {sun, mon, tue ,wed, thu, fri, sat};
              0    1    2    3    4    5    6
```

可以在声明时另行指定枚举元素的序号值（部分指定）

```
enum weekday {sun=7, mon=1, tue ,wed, thu, fri, sat};
              7      1      2    3    4    5    6
enum weekday {sun, mon=4, tue ,wed, thu, fri, sat};
              0    4      5    6    7    8    9
```

如果定义的常量重复，编译器不报错，但后续使用中会有问题

```
enum weekday {sun=3, mon=1, tue ,wed, thu, fri, sat};
              3      1      2    3    4    5    6
```

声明枚举类型时系统已初始化，声明后不允许再对枚举元素赋值

int sun; 错误 error C2365: “main::sun”: 重定义；以前的定义是“枚举数”

sun=10; 错误 error C2440: “=”: 无法从“int”转换为“main::weekday”

§7. 结构体、类和对象

7.13. 枚举

7.13.3. 枚举变量的定义形式

声明枚举类型的同时定义变量

enum 枚举类型名 {枚举(元素)常量取值表}变量;

先声明枚举类型，再用已声明的枚举类型定义变量

enum 枚举类型名 {枚举(元素)常量取值表};

enum 枚举类型名 变量;

该变量的值只能在枚举常量中取

枚举变量只能参与赋值、比较和输入输出操作，运算时用其本身的整数值

```
enum weekday {sun, mon, tue ,wed, thu, fri, sat} w;
              0    1    2    3    4    5    6
```

直接输出：

```
w = wed;
cout << w << endl; //3
printf("w=%d", w); //w=3
```

直接输入：

```
cin >> w;          //人为保证输入0-6之间
scanf("%d", &w);   //人为保证输入0-6之间
```

间接输入：

```
char s[4];
cin >> s; //键盘输入sun
if(!strcmp(s, "sun"))
    w = sun;
else if (...)
```

```
int week;
cin >> week;
w = (weekday)week; //强制类型转换
```

间接输出：

```
switch(w) {
    case wed:
        cout << "wed";
        break;
...}
```

§7. 结构体、类和对象

7.14. 类型声明

7.14.1. 定义

为一种数据类型声明一个新名字（没有创造新类型），使得代码的可读性更高，意义更明确

7.14.2. 形式

typedef 原有类型名 新类型名

通常将typedef声明的类型名用大写字母表示以便与系统提供的标准类型标识符区别

例如：typedef int COUNTER;

COUNTER i, j; 等价于 int i, j;

```
struct Point
    double x;
    double y;
}; //声明结构体类型
typedef struct Point POINT;
POINT pt; //新类型名定义结构体变量
pt.x = 10.0;
pt.y = 20.0;
```

```
typedef struct Point { //在struct前加了typedef，表明是声明新类型名
    double x;
    double y;
} POINT; //POINT是新类型名，不是变量名
POINT pt; //新类型名定义结构体变量
pt.x = 10.0;
pt.y = 20.0;
```

§7. 结构体、类和对象

7.14. 类型声明

7.14.3. 声明类型的方法步骤

(1)先按定义变量的方法写出定义语句　　int i;

(2)将变量名换成新类型名　　int INTEGER;

(3)在最前面加typedef　　typedef int INTEGER;

(4)用新类型名去定义变量　　INTEGER i, j;

声明类型的方法步骤（再例）

(1)先按定义变量的方法写出定义语句　　int a[10];

(2)将变量名换成新类型名　　int ARR[10];

(3)在最前面加typedef　　typedef int ARR[10];

(4)用新类型名去定义变量　　ARR a, b[5];

ARR a; 等价于 int a[10];

//相当于a被定义为含有10个元素的一维整型数组

ARR b[5]; 等价于 int b[5][10];

//相当于数组b中每一个元素被定义被一个含有10个元素的一维整型数组，数组b即二维数组

§7.结构体、类和对象

7.14.类型声明

7.14.4.typedef和#define的差异

typedef是声明类型，不是定义变量

#define是宏定义（用一个标识符来表示一个常量），发生在预处理阶段，也就是编译之前

它只进行简单机械的字符串替换，不进行任何检查

```
#define PIONT char*              //预处理指令，仅将char*替换为标识符PIONT，替换完毕再编译
PIONT a, b 的效果同 char* a, b; //表示定义了一个字符型指针变量a和字符型变量b
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.1. 引入及含义

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到既能共享，又不会因误操作而改变，引入共用数据保护的概念

```
void fun(int *p)
{   *p=10;
}
int main()
{   int k=15;
    fun(&k);
}
```

```
void fun(int &p)
{   p=10;
}
int main()
{   int k=15;
    fun(k);
}
```

通过指针/引用，在fun中改变了main的局部变量k的值

	能否指向不同变量	能否通过指针变量修改变量值	是否需要定义时初始化
指向常量的指针变量	✓	×	×
常指针	×	✓	✓
指向常量的常指针	×	×	✓

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.2. 常对象

常对象：

const 类名 对象名(初始化实参表)

或 类名 const 对象名(初始化实参表)

```
const Time t1(15);
Time const t2(16, 30, 0);
```

T1始终是15:00:00
T2始终是16:30:00

★ 与常变量相同，在整个程序的执行过程中值不可再变化

★ 与常变量相同，必须在定义时进行初始化

★ 不能调用普通成员函数(即使不改变数据成员的值)

demo.cpp

```cpp
#include <iostream>
using namespace std;
class Time {
public:
    int hour, minute, sec;
    Time(int h = 0, int m = 0, int s = 0)
    {  hour = h; minute = m; sec = s;  }
    void display()
    {  cout << hour << minute << sec;  }
};
int main()
{
    const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译报错
    t2.sec = 27;    //编译报错
    t1.display();   //编译报错(虽然不改)
}
```

```
demo.cpp(15,7): error C3892: “t1”: 不能给常量赋值
demo.cpp(16,7): error C3892: “t2”: 不能给常量赋值
demo.cpp(17,16): error C2662: “void Time::display(void)”: 不能将“this”指针从“const Time”转换为“Time &”
demo.cpp(17,5): message : 转换丢失限定符
demo.cpp(8,10): message : 参见“Time::display”的声明
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

常对象成员：

常对象中的所有数据成员在程序执行过程中值均不可变，如果只需要限制部分成员的值在执行过程中不可变，则需要引入常对象成员的概念

- 常数据成员：该数据成员的值在执行中不可变
- 常成员函数：该函数只能引用成员的值，不能修改

常数据成员：

```
class 类名 {
    const 数据类型 数据成员名
 或 数据类型 const 数据成员名
};

class Time {
    private:
        const int hour;
        int const minute;
        int sec;
};
```

常成员函数：

```
class 类名 {
    返回类型 成员函数名(形参表) const;
};


class Time {
    public:
    void display() const;
};
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 常数据成员要在构造函数中初始化，使用中值不变，在构造函数中进行初始化时，必须用参数初始化表

```
#define <iostream>
using namespace std;

class Time {
  public:
    const int hour;
    int minute, sec;
    Time(int h=0, int m=0, int s=0)
    {
        hour   = h;
        minute = m;
        sec    = s;
    }
};
int main()
{
   Time t1;
   t1.hour = 10; //错误
}
```

错误

```
demo.cpp(7,9): error C2789: “Time::hour”: 必须初始化常量限定类型的对象
demo.cpp(6): message : 参见“Time::hour”的声明
demo.cpp(7,9): error C2789: “Time::minute”: 必须初始化常量限定类型的对象
demo.cpp(6): message : 参见“Time::minute”的声明
demo.cpp(7,9): error C2789: “Time::sec”: 必须初始化常量限定类型的对象
demo.cpp(6): message : 参见“Time::sec”的声明
demo.cpp(9,9): error C2166: 左值指定 const 对象
demo.cpp(10,9): error C2166: 左值指定 const 对象
demo.cpp(11,9): error C2166: 左值指定 const 对象
demo.cpp(17,7): error C3892: “t1”: 不能给常量赋值
```

```
#define <iostream>
using namespace std;

class Time {
  public:
    const int hour;
    int minute, sec;
    Time(int h=0, int m=0, int s=0) : hour(h)
    {
        minute = m;
        sec    = s;
    }
};
int main()
{
   Time t1;
   t1.hour = 10; //错误
}
```

正确

```
demo.cpp(17,7): error C3892: “t1”: 不能给常量赋值
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 常数据成员要在构造函数中初始化，使用中值不变，在构造函数中进行初始化时，必须用参数初始化表

★ 常成员函数只能引用类的数据成员(无论是否常数据成员)的值，而不能修改数据成员的值

普通成员函数

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    void set(int h=0,int m=0,int s=0)
    {  hour   = h;
       minute = m;
       sec    = s;
    }
};
int main()
{
   Time t1;
   t1.set(14,15,23);
}
```

正确

常成员函数

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    void set(int h=0,int m=0,int s=0) const
    {  hour   = h;
       minute = m;
       sec    = s;
    }
};
int main()
{
   Time t1;
   t1.set(14,15,23);
}
```

错误

```
demo.cpp(8,9): error C3490: 由于正在通过常量对象访问“hour”，因此无法对其进行修改
demo.cpp(9,9): error C3490: 由于正在通过常量对象访问“minute”，因此无法对其进行修改
demo.cpp(10,9): error C3490: 由于正在通过常量对象访问“sec”，因此无法对其进行修改
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 常成员函数写成下面形式，编译不报错但不起作用

const 返回类型 成员函数名(形参表)

或 返回类型 const 成员函数名(形参表)

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    const void set(int h, int m,int s)
    {   hour   = h;
        minute = m;
        sec    = s;
    }
};

int main()
{  Time t1;
   t1.set(14,15,23);
}
```

赋值正确，说明set不是常成员函数

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    void const set(int h, int m, int s)
    {   hour   = h;
        minute = m;
        sec    = s;
    }
};

int main()
{  Time t1;
   t1.set(14,15,23);
}
```

赋值正确，说明set不是常成员函数

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 常成员函数可以调用本类的另一个常成员函数，但不能调用本类的非常成员函数(即使该非常成员函数不修改数据成员的值)

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    void display()
    {   cout << hour << endl;
    }
    void fun() const
    {   display();
    }
};
int main()
{
   Time t1;
   t1.fun();
}
```

错误

```
demo.cpp(9,1): error C2662: “void Time::display(void)”: 不能将“this”指针从“const Time”转换为“Time &”
demo.cpp(9,9): message : 转换丢失限定符
demo.cpp(6,10): message : 参见“Time::display”的声明
```

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    void display() const
    {   cout << hour << endl;
    }
    void fun() const
    {   display();
    }
};
int main()
{
   Time t1;
   t1.fun();
}
```

正确

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 若希望常成员函数能强制修改数据成员，则要将数据成员定义为mutable

(适用场景:一个复杂大类，希望绝大多数成员函数不修改该值)

```
#include <iostream>
using namespace std;

class Time {
    public:
    int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {   hour   = h;
        minute = m;
        sec    = s;
    }
};
int main()
{  Time t1;
   t1.set(14,15,23);
}
```

错误

```
demo.cpp(8,9): error C3490: 由于正在通过常量对象访问“hour”，因此无法对其进行修改
demo.cpp(9,9): error C3490: 由于正在通过常量对象访问“minute”，因此无法对其进行修改
demo.cpp(10,9): error C3490: 由于正在通过常量对象访问“sec”，因此无法对其进行修改
```

```
#include <iostream>
using namespace std;
class Time {
    public:
        mutable int hour, minute, sec;
        void set(int h=0, int m=0, int s=0) const
        {   hour   = h;
            minute = m;
            sec    = s;
        }
};
int main()
{  Time t1;
   t1.set(14,15,23);
}
```

正确

```
#include <iostream>
using namespace std;
class Time {
    public:
        int mutable hour, minute, sec;
        void set(int h=0, int m=0, int s=0) const
        {   hour   = h;
            minute = m;
            sec    = s;
        }
};
int main()
{  Time t1;
   t1.set(14,15,23);
}
```

正确

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 若定义对象为常对象，则只能调用其中的常成员函数(不能修改数据成员的值)，而不能调用其中的普通成员函数(即使该成员不修改数据成员的值)

本节开始的例子

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    Time(int h=0, int m=0, int s=0) {  hour = h; minute = m; sec = s; }
    void display() {  cout << hour << minute << sec; }
};
int main()
{   const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译错误
    t2.sec = 27;    //编译错误
    t1.display();   //编译错误
}
```

```
#include <iostream>
using namespace std;

class Time {
  public:
    int hour, minute, sec;
    Time(int h=0, int m=0, int s=0) {  hour = h; minute = m; sec = s; }
    void display() const {  cout << hour << minute << sec; }
};
int main()
{   const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译错误
    t2.sec = 27;    //编译错误
    t1.display();   //编译正确
}
```

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.3. 常对象成员

使用：

★ 不能定义构造/析构函数为常成员函数(常识性问题)

★ 全局函数不能定义const

```
void fun() const //编译报错
{
    return;
}
int main()
{
    fun();
}
```

	普通数据成员	const 数据成员	mutable 数据成员	普通成员函数	const 成员函数
普通对象 公有	读写	读	读写	可调用	可调用
const对象	读	读	读写	不可调用	可调用
普通成员函数	读写	读	读写	可调用	可调用
const成员函数	读	读	读写	不能调用	可调用

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.4. 指向常对象的指针变量

⇔ 指向常量的指针变量

7.15.5. 指向对象的常指针

⇔ 常指针

7.15.6. 指向常对象的常指针

⇔ 指向常量的常指针

§7. 结构体、类和对象

7.15. 类的共用数据的保护

7.15.7. 指向变量/对象的常引用

```
const int k;
int const &ref2 = k;

const Time t1;
const Time &ref1 = t1;
```

① 指向常对象的指针变量
⇔ 指向常量的指针变量

② 指向对象的常指针
⇔ 常指针

③ 指向常对象的常指针
⇔ 指向常量的常指针

★ 常引用与常指针的使用基本类似

★ 引用必须在声明时进行初始化，指向同类型变量，整个生存期内不能再指向其它变量
=> 普通引用已符合②且不可能是①

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.1. 引入

希望在同一个类的多个对象间实现数据共享

★ 一个对象修改，另一个对象访问得到修改后的值

★ 类似与全局变量的概念，但属于类，仅供该类的不同对象间共享数据

7.16.2. 静态数据成员

定义：

```
class 类名 {
    private/public:
        static 数据类型 成员名;
        ...
};
```

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.2. 静态数据成员

使用：

★ 静态数据成员不属于任何一个对象，不在对象中占用空间，
单独在静态数据区分配空间(初值为0，不随对象的释放而释放)，
一个静态数据成员只占有一个空间，所有对象均可共享访问

★ 静态数据成员同样受类的作用域限制

★ 静态数据成员必须进行初始化，初始化位置在类定义体后，
函数体外进行(此时不受类的作用域限制)
数据类型 类名::静态数据成员名=初值;

★ 虽然可以通过this指针访问，但是数据不在一起

```
#include <iostream>
using namespace std;

int a=10;
class Time {
  private:
    static int hour;
    int minute, sec;
  public:
    Time(int h=0, int m=0, int s=0) //未对hour赋值
    {  minute = m;
       sec = s;
    }
    void display() //打印三个成员的地址，目的？
    {  cout << hour << ':'<< minute << ':'<< sec << endl;
       cout << &hour << endl;
       cout << &minute << endl;
       cout << &sec << endl;
    }
};

int Time::hour = 5; //虽然是private，可以

int main()
{  Time t1;
   cout << sizeof(t1) << endl;
   cout << &a << endl;
   t1.display();
}
```

将display函数中的成员访问
均加上this->，观察结果
(例：this->hout/&this->minute)

8
a的地址
5:0:0
hour的地址(与a相邻，与下面两个差别很大)
minute的地址
sec的地址

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.2. 静态数据成员

使用：

★ 不能通过参数初始化表进行初始化，但可以通过赋值方式初始化

```
#include <iostream>
using namespace std;

class Time {
    private:
        static int hour;
        int minute, sec;
    public:
        Time(int h=0, int m=0, int s=0) : hour(h)
        {
            minute = m;  sec = s;
        }
        void display()
        {
            cout << hour << minute << sec << endl;
        }
};

int Time::hour = 5; //必须要有，否则编译错

int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

error C2438: “hour”: 无法通过构造函数初始化静态类数据

错误

```
#include <iostream>
using namespace std;

class Time {
    private:
        static int hour;
        int minute, sec;
    public:
        Time(int h=0, int m=0, int s=0)
        {
            hour = h;  minute = m;  sec = s;
        }
        void display()
        {
            cout << hour << minute << sec << endl;
        }
};

int Time::hour = 5; //必须要有，否则编译错

int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

问1：hour初始化为5，在构造函数中又被赋值为h，最终是几？

问2：哪个先执行?为什么？

正确

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.2. 静态数据成员

使用：

★ 不能通过参数初始化表进行初始化，但可以通过赋值方式初始化

★ 既可以通过类型引用，也可以通过对象名引用

★ 结论：静态数据成员不是面向对象的概念，它破坏了数据的封装性，但方便使用，提高了运行效率

```
#include <iostream>
using namespace std;

class Time {
 public: //不符合规范，为了在main中直接访问
    static int hour;
    int minute;
    int sec;
  public:
    Time();
    Time(int h,int m,int s);
    void display();
};

int Time::hour = 10;

Time::Time()
{
    hour=0; minute=0; sec=0;
}
Time::Time(int h,int m,int s)
{
    hour=h; minute=m; sec=s;
}
void Time::display()
{   cout << hour << ":" << minute << ":" << sec << endl;
}
```

```
int main()
{
    Time t1(14,15,23), t2;
    t1.display();   0:15:23
    t2.display();   0:0:0
    t1.hour = 8;    //对象名引用
    t1.display();   8:15:23
    t2.display();   8:0:0
    Time::hour = 19;//类型引用
    t1.display();   19:15:23
    t2.display();   19:0:0

    return 0;
}
```

为什么既不是14:15:23，也不是10:15:23？

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.3. 静态成员函数

定义： class 类名 {

```
        private/public:
            static 返回类型 函数名(形参表);
            ...
    };
```

调用：

类名::成员函数名(实参表);

任意对象名.成员函数名(实参表);

使用：

★ 没有this指针，不属于某个对象

普通成员函数：

Time t1;

t1.display() ⇔ t1.display(&t1);

静态成员函数：

无

```
#include <iostream>
using namespace std;

class test {
    public:
        static void fun(int x)
        {
            cout << x << endl;
        }
};
int main()
{
    test t1;
    t1.fun(10);     //10
    test::fun(15);  //15
}
```

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.3. 静态成员函数

使用：

★ 没有this指针，不属于某个对象

★ 允许体内实现或体外实现

★ 静态成员函数中可以直接访问静态数据成员

体内实现

```
#include <iostream>
using namespace std;

class test {
  private:
    static int a;
  public:
    static void fun()
    {   cout << a << endl;
    }
};
int test::a = 10;
int main()
{   test t1;
    t1.fun();    //10
    test::fun();//10
}
```

```
#include <iostream>
using namespace std;

class test {
  private:
    static int a;
  public:
    static void fun()
    {   cout << this->a << endl;
    }
};
int test::a = 10;
int main()
{   test t1;
    t1.fun();
    test::fun();
}
```

静态函数没有this指针，显式加上后会报错!!!

error C2355: “this”：只能在非静态成员函数或非静态数据成员初始值设定项的内部引用

体外实现

```
#include <iostream>
using namespace std;
class test {
    private:
        static int a;
    public:
        static void fun();
};
int test::a = 10;
void test::fun()   //此处不能static
{   cout << a << endl;
}
int main()
{   test t1;
    t1.fun();    //10
    test::fun();//10
}
```

§7. 结构体、类和对象

7.16. 类的静态成员

7.16.3. 静态成员函数

使用：

★ 在静态成员函数中不能对非静态数据成员进行直接访问，而要通过对象参数的方式(不提倡!!!)

```
#include <iostream>
using namespace std;

class test {
   private:
      static int a;
      int b;
   public:
      static void fun()
      {
         a=10; //正确
         b=11; //错误，因为没有this指针，不知道
       }       //应该访问那个对象的b成员
};

int test::a = 5;

int main()
{
    test t1;
    t1.fun();
    test::fun();
}
```

```
demo.cpp(12,9): error C2597: 对非静态成员“test::b”的非法引用
demo.cpp(7): message : 参见“test::b”的声明
```

```
#include <iostream>
using namespace std;

class test {
   private:
      static int a;
      int b;
   public:
      static void fun(test &t)
      {
         a=10;    //正确
         t.b=11;  //正确
      }
};

int test::a = 5;

int main()
{
    test t1, t2;
    t1.fun(t1);
    test::fun(t2);
}
```

不提倡，建议静态成员函数只访问静态数据成员

§7. 结构体、类和对象

7.17. 类模板

7.17.1. 函数模板

§4. 函数

4.15. 函数模板

函数重载的不足：对于参数个数相同，类型不同，
而实现过程完全相同的函数，
仍要分别给出各个函数的实现

问题:两段一样的代码
能否合并为一段？

```
int max(int x, int y)
{
    return x>y?x:y;
}
double max(double x, double y)
{
    return x>y?x:y;
}
```

一段代码,两个功能
1、两个int型求max
2、两个double型求max

```
#include <iostream>
using namespace std;
template <typename T> //虚类型T
T my_max(T x, T y)
{   cout << sizeof(x) << ' ';
    return x > y ? x : y;
}
int main()
{   int    a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << my_max(a, b) << endl;
    cout << my_max(f1, f2) << endl;
    return 0;
}
```

```
4 15
8 23.45
```

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为函数模板，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

问题1：如果传入两个unsigned int型数据，
T的类型被实例化为什么？如何证明？
问题2：如果x,y的类型不同，自行摸索转换规律

§7. 结构体、类和对象

7.17. 类模板

7.17.1. 函数模板

§4. 函数

4.15. 函数模板

函数重载的不足：对于参数个数相同，类型不同，而实现过程完全相同的函数，仍要分别给出各个函数的实现

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为函数模板，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

使用：

★ 仅适用于参数个数相同、类型不同，实现过程完全相同的情况

★ typename可用class替代

★ 类型定义允许多个

```
template <typename T1, typename t2>
template <class T1, class t2>
```

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
char my_max(T1 x, T2 y)
{
    cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}
int main()
{
    int    a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << my_max(a, f1) << endl;
    cout << my_max(f2, b) << endl;
    return 0;
}
```

?

问：max(a,f1)时，T1/T2类型分别是?
max(f2,b)时，T1/T2类型分别是?

§7. 结构体、类和对象

7.17. 类模板

7.17.2. 类模板

引入：多个类，功能及实现完全相同，仅数据类型不同

```
class compare_int {
 private:
   int x, y;
 public:
   compare_int(int a, int b)
     { x=a;   y=b;   }
   int max()
     { return x>y?x:y; }
   int min()
     { return x<y?x:y; }
 };
```

```
class compare_float {
 private:
   float x, y;
 public:
   compare_float(float a, float b)
     { x=a;   y=b;   }
   float max()
     { return x>y?x:y; }
   float min()
     { return x<y?x:y; }
 };
```

合并为一个类型
为虚类型T的类

```
#include <iostream>
using namespace std;

template <class T>
class compare {
    private:
        T x, y;
    public:
        compare(T a, T b)
        { x=a;   y=b;
        }
        T max()
        { return x > y ? x : y;
         }
        T min()
        { return x < y ? x : y;
        }
};
int main()
{    compare <int>   c1(10,15);
     compare <float> c2(10.1f, 15.2f);
     cout << c1.max() << endl;   15
     cout << c2.min() << endl;   10.1
}
```

§7. 结构体、类和对象

7.17. 类模板

7.17.2. 类模板

使用：

★ 类模板使用的多个类需要满足下面的条件

- 数据成员个数相同，类型不同
- 成员函数个数相同，类型不同，实现过程完全相同

=> 推论：如果两个类大部分相同，可以通过定义不需要的数据成员及成员函数来达到使用类模板的目的(按实际情况决定)

```
class name_1 {
 private:
   int x, y, z;
 public:
   int f1(); //实际不需要
   void f2(int);
   void f3(int);
   void f4(int);
};
```

```
class name_1 {
 private:
   float x, y, z; //z实际不需要
 public:
   float f1();
   void f2(float);
   void f3(float);
   void f4(float);
};
```

§7. 结构体、类和对象

7.17. 类模板

7.17.2. 类模板

使用：

★ 类模板使用的多个类需要满足下面的条件

★ 类模板可以看作是类的抽象，称为参数化的类

★ 类模板成员函数体外实现时形式有所不同

体内实现

```
#include <iostream>
using namespace std;

template <class T>
class compare {
    private:
        T x, y;
    public:
        compare(T a, T b)
            {  x=a;   y=b;   }
        T max()
            {  return x>y?x:y; }
        T min()
            {  return x<y?x:y; }
};

int main()
{   compare <int>   c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl;
    cout << c2.min() << endl;
}
```

体外实现

```
#include <iostream>
using namespace std;

template <class T>
class compare {
    private:
        T x, y;
    public:
        compare(T a, T b);
        T max();
        T min();
};
template <class T>  //每个体外实现的函数前都要
compare<T>::compare(T a, T b)
{   x=a;
    y=b;
}
template <class T>  //每个体外实现的函数前都要
T compare<T>::max()
{  return x>y?x:y;
}
template <class T>  //每个体外实现的函数前都要
T compare<T>::min()
{  return x<y?x:y;
}
int main()
{    compare <int>   c1(10,15);
     compare <float> c2(10.1f, 15.2f);
     cout << c1.max() << endl;
     cout << c2.min() << endl;
}
```

普通类构造函数的体外实现：

```
test::test(int x, int y)
{
    …
}
```

普通类成员函数的体外实现：

```
int test::fun(int x, int y)
{
    …
}
```

§7. 结构体、类和对象

7.17. 类模板

7.17.2. 类模板

使用：

★ 类模板使用的多个类需要满足下面的条件

- 数据成员个数相同，类型不同
- 成员函数个数相同，类型不同，实现过程完全相同

=> 推论：如果两个类大部分相同，可以通过定义不需要的数据成员及成员函数来达到使用类模板的目的(按实际情况决定)

★ 类模板可以看作是类的抽象，称为参数化的类

★ 类模板成员函数体外实现时形式有所不同

★ 类型定义允许多个

```
template <class t1, class t2>
```

```
#include <iostream>
using namespace std;

template <class t1, class t2>
class test {
  private:
    t1 x;
    t2 y;
  public:
    test(t1 a, t2 b) {
      x=a;
      y=b;
      }
};

int main()
{
    test <int,float> c1(10,15.1);
    test <int,int>   c2(10,15);
}
```

§7. 结构体、类和对象

7.18. 枚举类

7.18.1. C方式传统枚举存在的问题

★ 不同enum的枚举常量值不能相同，但实际应用中有此需求

★ 不同enum的变量/常量不应该允许比较，但编译运行正确(无法解释语义)

★ 不同enum的变量/常量均可以直接整型输出，容易造成误解

★ C方式，enum可以直接用整型赋值，不检查范围

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围

§7. 结构体、类和对象

7.18. 枚举类

7.18.1. C方式传统枚举存在的问题

★ 不同enum的枚举常量值不能相同，但实际应用中有此需求

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue, wed, thu, fri, sat };
enum candidates { zhao, qian, sun, li };
int main()
{
    cout << wed << endl;
    return 0;
}
```

```
demo.cpp(4,31): error C2365: “sun”: 重定义；以前的定义是“枚举数”
demo.cpp(3): message : 参见“sun”的声明
```

★ 不同enum的变量/常量均可以直接整型输出，容易造成误解

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
enum candidates { zhao, qian, li };
int main()
{   cout << sun << ' ' << li << endl;
    printf("%d %d\n", sun, li);
    return 0;
}
```

```
2 0
2 0
```

★ 不同enum的变量/常量不应该允许比较，但编译运行正确(无法解释语义)

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
enum candidates { zhao, qian, li };
int main()
{   enum week w1=mon;
    enum candidates c1=qian;
    cout << (sun < li) << ' ' << (w1==c1)<< endl;
}
```

```
1 1
```

§7. 结构体、类和对象

7.18. 枚举类

7.18.1. C方式传统枚举存在的问题

★ C方式，enum可以直接用整型赋值，不检查范围

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围

注意：源程序必须.c形式

```
#include <stdio.h>

enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = 0, w3 = 4; //超定义范围
    printf("%d %d %d\n", w1, w2, w3);
    return 0;
}
```

```
1 0 4
```

cpp：直接整型，报错

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = 0, w3 = 4; //超定义范围
    cout << w1 << ' ' << w2 << ' ' << w3 << endl;
    return 0;
}
```

```
demo.cpp(6,31): error C2440: "初始化": 无法从"int"转换为"week"
demo.cpp(6,28): message : 转换为枚举类型需要显式强制转换(static_cast、C 样式强制转换或带圆括号函数样式强制转换)
demo.cpp(6,39): error C2440: "初始化": 无法从"int"转换为"week"
demo.cpp(6,36): message : 转换为枚举类型需要显式强制转换(static_cast、C 样式强制转换或带圆括号函数样式强制转换)
```

cpp：强转，VS报IntelliSense

```
#include <iostream>
using namespace std;
enum week { sun, mon, tue };
int main()
{
    enum week w1 = mon, w2 = week(0), w3 = week(4);
    cout << w1 << ' ' << w2 << ' ' << w3 << endl;
    return 0;
}
```

C26812 枚举类型"week"未设定范围。相比于 "enum"，首选 "enum class" (Enum.3)。

```
1 0 4
```

§7. 结构体、类和对象

7.18. 枚举类

7.18.2. 枚举类的定义与使用

定义：enum class 枚举类型 { 枚举常量1, ...,枚举常量n }

使用：类名::常量值

★ 枚举类常量为整型值，从0开始（同enum）

★ 枚举类常量不能直接输出(enum可直接输出为整型)

★ 其余使用方法基本同enum

★ 不同enum class可定义相同的枚举常量标识符(常量值可不同)

★ 不同enum class之间的常量值不允许直接比较

★ enum class的枚举变量/常量不允许直接输出，需要加强制类型转换

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围(同enum)

§7. 结构体、类和对象

7.18. 枚举类

7.18.2. 枚举类的定义与使用

★ 不同enum class可定义相同的枚举常量标识符(常量值可不同)

编译正确

```
#include <iostream>
using namespace std;

enum class week { sun, mon, tue, wed, thu, fri, sat };
enum class candidates { zhao, qian, sun, li };
int main()
{
    return 0;
}
```

★ 不同enum class之间的常量值不允许直接比较

编译报错

```
#include <iostream>
using namespace std;

enum class week { sun, mon, tue };
enum class candidates { zhao, qian, li };
int main()
{   cout << (week::sun < candidates::li) << endl;
    return 0;
}
```

error C2676: 二进制“<”：“week”不定义该运算符或到预定义运算符可接收的类型的转换

(week::sun < candidates::li)

VS中鼠标悬停在<上时给出的提示信息

没有与这些操作数匹配的 "<" 运算符
操作数类型为: week < candidates

联机搜索

§7. 结构体、类和对象

7.18. 枚举类

7.18.2. 枚举类的定义与使用

★ enum class的枚举变量/常量不允许直接输出，需要加强制类型转换

```
#include <iostream>
using namespace std;
enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{
    cout << wed << endl;
    printf("%d\n", mon);
    return 0;
}
```

enum class：直接写常量值，报未声明错

```
demo.cpp(6,13): error C2065: "wed": 未声明的标识符
demo.cpp(7,20): error C2065: "mon": 未声明的标识符
```

```
#include <iostream>
using namespace std;

enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{
    cout << week::wed << endl;
    return 0;
}
```

enum class：写类名::常量值 方式 cout直接输出报错

```
demo.cpp(6,23): error C2679: 二元"<<": 没有找到接受"week"类型的右操作数的运算符(或没有可接受的转换)
```

```
#include <iostream>
using namespace std;

enum class week { sun, mon, tue, wed, thu, fri, sat };
int main()
{   cout << int(week::end) << endl;
    printf("%d\n", week::mon);
    return 0;
}
```

enum class：写类名::常量值 方式
cout需强转后输出整型
printf直接输出整型值

```
3
1
```

§7. 结构体、类和对象

7.18. 枚举类

7.18.2. 枚举类的定义与使用

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围(同enum)

enum class无IntelliSense

```
#include <iostream>
using namespace std;
enum class week { sun, mon, tue };
int main()
{
    week w1 = week::mon, w2 = week(0), w3 = week(4);
    printf("%d %d %d\n", w1, w2, w3);
    return 0;
}
```

```
1 0 4
```

§7. 结构体、类和对象

7.18. 枚举类

7.18.3. enum与enum class的比较

★ 不同enum的枚举常量值不能相同，但实际应用中有此需求

=> enum class允许相同

★ 不同enum的变量/常量不应该允许比较，但编译运行正确(无法解释语义)

=> enum class不允许比较

★ 不同enum的变量/常量均可以直接整型输出，容易造成误解

=> enum class的cout不允许直接输出(加强制类型转换)，printf允许

★ C方式，enum可以直接用整型赋值，不检查范围

=> enum class不支持C方式

★ C++方式，enum不能直接整型赋值，需加强制类型转换，不检查范围

=> enum class同，且VS下无 IntelliSense

§7. 结构体、类和对象

7.19. 共用体

例： 定义一个用于一卡通管理系统的结构，要求包含卡号、余额、消费限额、消费密码等公共信息，
此外，若持卡人是学生，要包含学号、姓名、专业等学生特有的信息，
若持卡人是教师，则包含工号、姓名、职称等教师特有的信息

```
struct student {
    定义学生信息;
};
struct teacher {
    定义教师信息;
};
struct ykt {
    公共信息;
    student sinfo;
    teacher tinto;
};
int main()
{
    ykt y1;//定义变量
}
```

对y1的成员的访问：

```
int main()
{
    ykt y1;
    ...;
    y1.卡号
    y1.sinfo.学号
    y1.tinfo.工号
    ...;
    return 0;
}
```

缺陷：无论持卡人何种身份sinfo和tinfo中必然有一个是不需要填写任何信息的，从而导致存储空间的浪费
解决：能否使sinfo/tinfo共用一段空间，当持卡人是学生时，这段空间按student方式访问，
当持卡人是教师时按teacher方式访问

=>(共用体)

§7. 结构体、类和对象

7.19. 共用体

```
union 共用体名 {
    共用体成员1（类型名  成员名）
    ...
    共用体成员n（类型名  成员名）
};
```

```
union data {
    short a;
    long  b;
    char  c;
};
```

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

★ 给一个共用体成员赋值后，会覆盖其它成员的值，因此只有最后一次存放的成员是有效的

★ 其它所有定义、使用方法同结构体

```
#include <iostream>
using namespace std;

struct data1 {
    short a;
    long  b;
    char  c;
};

union data2 {
    short a;
    long  b;
    char  c;
};

int main()
{
    data1 d1;
    data2 d2;

    cout << sizeof(d1) << ' ' << sizeof(d2) << endl;
    cout << &d1.a << ' ' << &d1.b << ' ' << (void *)&d1.c << endl;
    cout << &d2.a << ' ' << &d2.b << ' ' << (void *)&d2.c << endl;

    return 0;
}
```

12:所有成员所占空间之和(含填充字节)

4 :所有成员中最大成员所占空间

12 4
地址：X X+4 X+8
地址：Y Y Y

Microsoft Visual Studio 调试控制台

```
12 4
005BFE40 005BFE44 005BFE48
005BFE34 005BFE34 005BFE34
```

§7. 结构体、类和对象

7.19. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

```
#include <iostream>
using namespace std;

union data {
   int   a;
   short b;
   char  c;
};

int main()
{
   union data d;
   d.a=70000;
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.b=7000;
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.c='A';
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   return 0;
}
```

70000 4464 p

72536 7000 X

72513 6977 A

70000 = 00000000 00000001 00010001 01110000

§7. 结构体、类和对象

7.19. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

```
#include <iostream>
using namespace std;

union data {
   int   a;
   short b;
   char  c;
};

int main()
{
   union data d;
   d.a=70000;
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.b=7000;        70000 4464 p
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.c='A';         72536 7000 X
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
                    72513 6977 A
   return 0;
}
```

70000 = 00000000 00000001 00011011 01011000

7000 = 00011011 01011000

§7. 结构体、类和对象

7.19. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

```
#include <iostream>
using namespace std;

union data {
   int   a;
   short b;
   char  c;
};

int main()
{
   union data d;
   d.a=70000;
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.b=7000;        70000 4464 p
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.c='A';         72536 7000 X
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
                    72513 6977 A
   return 0;
}
```

70000 = 00000000 00000001 00011011 01000001

A = 01000001

§7. 结构体、类和对象

7.19. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

```
#include <iostream>
using namespace std;

union data {
   int   a;
   short b;
   char  c;
};

int main()
{
   union data d;
   d.c='A';
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.b=7000;        不确定 不确定 A
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
   d.a=70000;       不确定 7000 X
   cout << d.a << ' ' << d.b << ' ' << d.c << endl;
                    70000 4464 p
   return 0;
}
```

d:低位在前存放	
2000	01000001
2001	???
2002	???
2003	???

d:低位在前存放	
2000	01011000
2001	00011011
2002	???
2003	???

d:低位在前存放	
2000	01110000
2001	00010001
2002	00000001
2003	00000000

§7. 结构体、类和对象

7.19. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

```
#include <iostream>
using namespace std;

union data {
    int   a;
    short b;
    char  c;
};

int main()
{
    union data d;
    d.a=70000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.b=7000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.c='A';
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;

    return 0;
}
```

70000 = 00000000 00000001 00010001 01110000

d:低位在前存放

2000	01110000	a	b	c
2001	00010001			
2002	00000001			
2003	00000000			

d:高位在前存放

2000	00000000	a	b	c
2001	00000001			
2002	00010001			
2003	01110000			

不同的CPU(和操作系统、编译器无关)有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，称为主机序，常见的有两种：

★ Little endian：将低序字节存储在起始地址，地址低位存储值的低位，地址高位存储值的高位(小头序/小字序)

★ Big endian ：将高序字节存储在起始地址，地址低位存储值的高位，地址高位存储值的低位(大头序/大字序)

思考：大字序系统中，本题的运行结果？

(注：本题无法通过Intel/AMD等小字序系统运行测试程序得到答案，需要手动计算)

§7. 结构体、类和对象

7.19. 共用体

例： 定义一个用于一卡通管理系统的结构，要求包含卡号、余额、消费限额、消费密码等公共信息，
此外，若持卡人是学生，要包含学号、姓名、专业等学生特有的信息，
若持卡人是教师，则包含工号、姓名、职称等教师特有的信息

```
struct student {
    定义学生信息;
};

struct teacher {
    定义教师信息;
};

struct ykt {
    公共信息;
    student sinfo;
    teacher tinto;
};

int main()
{
    ykt y1;//定义变量
}
```

空间浪费

```
struct student {
    定义学生信息;
};
struct teacher {
    定义教师信息;
};
union owner {
    student s;
    teacher t;
};
struct ykt {
    公共信息;
    char type; //持卡人类别
    owner info;
};
int main()
{
    ykt y1;//定义变量
}
```

此处保证s/t共用一段空间

```
int main()
{
    ykt y1;//定义变量
    ...;
    y1.卡号;
    if (y1.type=='s'){
        y1.info.s.学号;
        }
    else {
        y1.info.t.工号;
        }
    ...;

    return 0;
}
```

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

★ 一维数组的理解方法(下标法、指针法)

一维数组：

```
int a[12]={1,2,3,4,5,6,7,8,9,10,11,12};
```

a ： 数组名/数组的首元素地址(⇔&a[0])

由等价关系 a[i] ⇔ *(a+i)可得

&a[i] ： 第i个元素的地址 (下标法)

a+i ： 第i个元素的地址 (指针法)

a[i] ： 第i个元素的值 (下标法)

*(a+i) ： 第i个元素的值 (指针法)

&a[i] ⇔ a+i 地址

a[i] ⇔ *(a+i) 值

第0个元素的特殊表示：

a[0] ⇔ *(a+0) ⇔ *a

&a[0] ⇔ a+0 ⇔ a

a	2000	1	a[0]
	2004	2	a[1]
	2008	3	a[2]
	2012	4	a[3]
	2016	5	a[4]
	2020	6	a[5]
	2024	7	a[6]
	2028	8	a[7]
	2032	9	a[8]
	2036	10	a[9]
	2040	11	a[10]
	2044	12	a[11]

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

二维数组：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
    1   2   3   4
    5   6   7   8
    9   10  11  12
```

第5章的内容：
二维数组 int a[3][4]，
理解为一维数组，有3(行)个元素，
每个元素又是一维数组，有4(列)个元素

a是二维数组名，
a[0],a[1],a[2]是一维数组名

理解1：a [3][4]

理解2：a[3] [4]

a	2000	1	a[0][0]
	2004	2	a[0][1]
	2008	3	a[0][2]
	2012	4	a[0][3]
	2016	5	a[1][0]
	2020	6	a[1][1]
	2024	7	a[1][2]
	2028	8	a[1][3]
	2032	9	a[2][0]
	2036	10	a[2][1]
	2040	11	a[2][2]
	2044	12	a[2][3]

a	2000	1	a[0]
		2	
		3	
		4	
	2016	5	a[1]
		6	
		7	
		8	
	2032	9	a[2]
		10	
		11	
		12	

a	2000	1	a[0][0]
		2	[1]
		3	[2]
		4	[3]
	2016	5	a[1][0]
		6	[1]
		7	[2]
		8	[3]
	2032	9	a[2][0]
		10	[1]
		11	[2]
		12	[3]

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

★ 二维数组加一个下标的理解方法(下标法、指针法)

元素是指
4元素一维数组

```
int a[3][4]={1,…,12};
```

3种
理解方法

a ： ① 二维数组的数组名，即a
② 3元素一维数组的数组名，即a
③ 3元素一维数组的首元素地址,即&a[0]

&a[i] ： 3元素一维数组的第i个元素的地址

a+i ： 同上

行地址

a[i] ： 3元素一维数组的第i个元素的值
(即4元素一维数组的数组名
4元素一维数组的首元素的地址)

*(a+i) ： 同上

元素
地址

i:0-2(行)

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

★ 二维数组加两个下标的理解方法(下标法、指针法)

从第五章概念可知：

a[i][j]：第i行j列元素的值

&a[i][j]：第i行j列元素的地址

令x表示a[i]，则：

x[j]：第i行j列元素的值

&x[j]：第i行j列元素的地址

由一维数组的等价变换可得：

x[j]：第i行j列元素的值

&x[j]：第i行j列元素的地址

*(x+j)：第i行j列元素的值

x+j ：第i行j列元素的地址

所以，用a[i]替换回x，则可得：

a[i][j]:第i行j列元素的值

&a[i][j]:第i行j列元素的地址

*(a[i]+j):第i行j列元素的值

a[i]+j :第i行j列元素的地址

因为：对一维数组 a[i]⇔*(a+i)

所以：*(a[i]+j) ⇔ *(*(a+i)+j)(值)

a[i]+j ⇔ *(a+i)+j (元素地址)

a[i][j]:第i行j列元素的值

&a[i][j]:第i行j列元素的地址

*(a[i]+j):第i行j列元素的值

a[i]+j :第i行j列元素的地址

((a+i)+j):第i行j列元素的值

*(a+i)+j :第i行j列元素的地址

二维数组元素的值和元素的地址均有三种形式：

a[i][j] ⇔ *(a[i]+j) ⇔ *(*(a+i)+j) 值

&a[i][j] ⇔ a[i]+j ⇔ *(a+i)+j 元素地址

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

地址增量的变化规律

对一维数组a：

a+i 实际 a+i*sizeof(基类型)

a	2000	1	a[0]
	2004	2	a[1]
	2008	3	a[2]
	2012	4	a[3]
	2016	5	a[4]
	2020	6	a[5]
	2024	7	a[6]
	2028	8	a[7]
	2032	9	a[8]
	2036	10	a[9]
	2040	11	a[10]
	2044	12	a[11]

对二维数组a[m][n]：

a+i 实际 a+i*n*sizeof(基类型)

a[i]+j 实际 a+(i*n+j)*sizeof(基)

a	2000	1	a[0][0]
	2004	2	[1]
	2008	3	[2]
	2012	4	[3]
	2016	5	a[1][0]
	2020	6	[1]
	2024	7	[2]
	2028	8	[3]
	2032	9	a[2][0]
	2036	10	[1]
	2040	11	[2]
	2044	12	[3]

例：a+1：2016　行地址

a[1]+2：2024　元素地址

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

a	：地址(二维数组/第0行)	行地址
&a[i]	：地址(第i行)	
a+i	：地址(第i行)	
a[i]	：地址(第i行0列)	元素地址
*(a+i)	：地址(第i行0列)	
&a[i][j]	：地址(第i行j列)	
a[i]+j	：地址(第i行j列)	
*(a+i)+j	：地址(第i行j列)	
a[i][j]	：值(第i行j列)	值
*(a[i]+j)	：值(第i行j列)	
((a+i)+j)	：值(第i行j列)	

假设 int a[3][4] 存放在2000开始的48个字节中

2000
2016
2016
2016
2016
2024
2024
2024

假设 i=1
j=2

a+1是地址2016，*(a+1)取a+1的值，还是地址2016

a+1是行地址,*(a+1)取a+1的值,是元素地址

a[2]是地址2032，&a[2]取a[2]的地址，还是2032

a[2]是元素地址，&a[2]取a[2]的地址，是行地址

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

a	：地址(二维数组/第0行)	
&a[i]	：地址(第i行)	行地址
a+i	：地址(第i行)	
a[i]+0	：地址(第i行0列)	
*(a+i)+0	：地址(第i行0列)	元素地址
&a[i][j]	：地址(第i行j列)	
a[i]+j	：地址(第i行j列)	
*(a+i)+j	：地址(第i行j列)	
a[i][j]	：值(第i行j列)	值
*(a[i]+j)	：值(第i行j列)	
((a+i)+j)	：值(第i行j列)	

这两种情况虽然只看到一个下标，但要当做两个下标理解(i行0列的特殊表示)

&a[i]	：地址(第i行)
a+i	：地址(第i行)
a[i]	：地址(第i行0列)
*(a+i)	：地址(第i行0列)

由：&a[i]：行地址　　a[i]：元素地址

a+i　：行地址　　*(a+i)：元素地址

得：*行地址　　=> 元素地址（该行首元素）

如何证明?

&首元素地址 => 行地址　（必须首元素!!!）

如何证明?

进一步思考：

(1) &行地址　是什么？　&&行地址呢？

(2) *元素地址　是什么？　**元素地址呢？

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

实际运行一次，观察结果并思考!!!

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4];
    cout <<     a      << endl;    地址a
    cout <<   (a+1)    << endl;    地址a+16
    cout <<   (a+1)+1  << endl;    地址a+32
    cout << *(a+1)     << endl;    地址a+16
    cout << *(a+1)+1   << endl;    地址a+20
    cout <<   a[2]     << endl;    地址a+32
    cout <<   a[2]+1   << endl;    地址a+36
    cout << &a[2]      << endl;    地址a+32
    cout << &a[2]+1    << endl;    地址a+48(已超范围)
    return 0;
}
```

行地址

元素地址

行地址

说明：
每组打印地址后，
再打印地址+1，
目的是区分行地址及元素地址

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4];
    cout << sizeof(a)          << endl;
    cout << sizeof(a+1)        << endl;
    cout << sizeof(*(a+1))     << endl;
    cout << sizeof(*(a+1))     << endl;
    cout << sizeof(**(a+1))    << endl;
    cout << sizeof(a[2])       << endl;
    cout << sizeof(*(a[2]))    << endl;
    cout << sizeof(&a[2])      << endl;
    cout << sizeof(*(&a[2]))   << endl;
    return 0;
}
```

另一种验证方法!!!

行地址	48		数组大小
	4	即&a[1],是地址(指针)	a+1大小
	16	指针基类型是int[4]	a+1基类型 } 同
元素地址	16	即a[1],是数组(4元素)	*(a+1)大小 } 同
	4	数组元素是int	*(a+1)基类型
	16	a[2]是数组(4元素)	a[2]大小
	4	数组元素是int	a[2]基类型
行地址	4	数组a[2]的地址(指针)	&a[2]大小
	16	指针基类型是int[4]	&a[2]基类型
		*&a[2]⇔a[2],是数组(4元素)	

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

1.2. 指向二维数组元素的指针变量

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4], *p;
    p=a[0];
    p=&a[0][0];
    p=*a;
    p=a;
    p=&a[0];
}
```

编译正确，p指向a[0][0]

编译错误，因为a/&a[0]代表的是行地址

```
#include <iostream>
using namespace std;

int main()
{
   int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
   int *p = a[0];
   cout << sizeof(a)  << endl;   48        数组大小
   cout << sizeof(p)  << endl;   4         因为指针
   cout << sizeof(*p) << endl;   4         因为int
   cout << p      << endl;   地址a      元素[0][0]地址
   cout << p+5    << endl;   地址a+20   元素[1][1]地址
   cout << *(p+5) << endl;   6          a[1][1]的值
}
```

假设a的首地址是2000，则区别如下：

p=a[0]：p的值是2000，基类型是int，p+1的值为2004

p=a　：p的值是2000，基类型是int*4，p+1的值为2016

因为p是基类型为int的指针变量，所以：

p+i ⇔ p+i*sizeof(int)

p+5 ⇔ &a[1][1]

a	2000	1	a[0][0]
	2004	2	a[0][1]
	2008	3	a[0][2]
	2012	4	a[0][3]
	2016	5	a[1][0]
	2020	6	a[1][1]
	2024	7	a[1][2]
	2028	8	a[1][3]
	2032	9	a[2][0]
	2036	10	a[2][1]
	2040	11	a[2][2]
	2044	12	a[2][3]

§8. 指针进阶

1. 多维数组与指针

1.1. 二维数组的地址

1.2. 指向二维数组元素的指针变量

例：打印二维数组的值（以下四种方法均正确，均是按一维方式顺序循环）

```
int main()
{  int a[3][4]={...}, *p;
   for(p=a[0];p<a[0]+12;p++)
      cout << *p << ' ';
   cout << endl;
   return 0;
}
```

```
int main()
{  int a[3][4]={...};
   int i, j, *p = a[0];
   for(i=0; i<3; i++)
     for(j=0; j<4;j++)
        cout << *p++ << ' ';
   return 0;
}
```

```
int main()
{  int a[3][4]={...};
   int i, j, *p=&a[0][0];
   for(i=0; i<12; i++)
      cout << *p++ << ' ';
   return 0;
}
```

```
int main()
{  int a[3][4]={...}
   int i, j, *p=&a[0][0];
   for(; p-a[0]<12;)
     cout << *p++ << ' ';
   return 0;
}
```

§8. 指针进阶

1. 多维数组与指针

1.3. 指向由m个元素组成的一维数组的指针变量

```
#include <iostream>
using namespace std;
int main()
{   int a[3][4], (*p)[4];
    p=a[0];
    p=&a[0][0];
    p=*a;
    p=a;
    p=&a[0];
}
```

编译错误

编译正确

```
int a[3][4]={…};
int (*p)[4]=a;
```

(*p)有4个元素

每个元素类型是int

=> p是指向4个元素组成的一维数组的指针

```
int a[4]
```

a有4个元素

每个元素类型是int

*p+j / *(p+0)+j:取这个一维数组中的第j个元素

p+i 实际 p+i*4*sizeof(int)

*(p+i)+j 实际 p+(i*4+j)*sizeof(int)

★ 使用：

p：地址(m个元素组成的一维数组的地址)

*p：值(是一维数组的名称，即一维数组的首元素地址)

§8. 指针进阶

1. 多维数组与指针

1.3. 指向由m个元素组成的一维数组的指针变量

int a[3][4]={1,…,12}, (*p)[4];

p = a;

p+1 ： 行地址2016(a[1])

*p+1 ： 元素地址2004(a[0][1])　　p是行地址2000

*p是元素地址2000

*(*p+1) ： 元素值2(a[0][1])

*(p+1)+2 ： 元素地址2024(a[1][2])　　p+1是行地址2016

*(p+1)是元素地址2016

((p+1)+2)： 元素值7(a[1][2])

a	2000	1	a[0][0]
	2004	2	a[0][1]
	2008	3	a[0][2]
	2012	4	a[0][3]
	2016	5	a[1][0]
	2020	6	a[1][1]
	2024	7	a[1][2]
	2028	8	a[1][3]
	2032	9	a[2][0]
	2036	10	a[2][1]
	2040	11	a[2][2]
	2044	12	a[2][3]

```
#include <iostream>
using namespace std;
int main()
{  int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
   int (*p)[4] = a;
   cout << sizeof(a)  << endl;
   cout << sizeof(p)  << endl;
   cout << sizeof(*p) << endl;
   cout << p       << endl;
   cout << p+1     << endl;
   cout << *p      << endl;
   cout << *p+1    << endl;
   cout << *(*p+1) << endl;
}
```

48	数组大小
4	因为指针
16	因为int[4]
地址a	行地址
地址a+16	+1 = +16
地址a	元素地址
地址a+4	+1 = +4
2	a[0][1]的值

```
#include <iostream>
using namespace std;

int main()
{
    int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
    int (*pl)[4], *p;
    for (pl=a; pl < a+3; pl++) {     //行指针
        for (p=*pl; p < *pl+4; p++) //元素指针
            cout << *p << ' ';
        cout << endl; //每行一个回车
        }
}
```

§8. 指针进阶

1. 多维数组与指针

1.4. 用指向二维数组元素的指针做函数参数

★ 形参是对应类型的简单指针变量

```
#include <iostream>
using namespace std;

void fun(int *data)
{
    if (*data%2==0)
        cout << *data << endl;
}
int main()
{
    int a[3][4]={...}, *p;
    for(p=a[0]; p<a[0]+12; p++)
        fun(p);
    cout << endl;

    return 0;
}
```

实参是指向二维数组元素的指针变量
形参是对应类型的简单指针变量

§8. 指针进阶

1. 多维数组与指针

1.5. 用指向二维数组的指针做函数参数

思考：若f1/f2/f3中为sizeof(**x1/**x2/**x3)
则：结果是多少？为什么？

5.4. 用数组名作函数参数

5.4.3. 用多维数组名做函数实参

当时第5章的说法，都不准确，形参数组不存在
形参的本质是指针变量

★ 形参为相应类型的多维数组

★ 实、形参数组的列必须相等，形参的行可以不指定，或为任意值(实参传入二维数组的首地址，只要知道每行多少列实形参即可对应，不关心行数)

```
#include <iostream>
using namespace std;
void f1(int x1[][4])     //形参数组不指定行大小
{   cout << "x1_size=" << sizeof(x1) << endl;
}
void f2(int x2[3][4])    //形参数组行大小与实参相同
{   cout << "x2_size=" << sizeof(x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{   cout << "x3_size=" << sizeof(x3) << endl;
}
int main()
{   int a[3][4];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);
    f2(a);
    f3(a);
}
```

```
a_size=48
x1_size=4    因为 *
x2_size=4    因为 *
x3_size=4    因为 *
```

```
#include <iostream>
using namespace std;
void f1(int x1[][4])     //形参数组不指定行大小
{   cout << "x1_size=" << sizeof(*x1) << endl;
}
void f2(int x2[3][4])    //形参数组行大小与实参相同
{   cout << "x2_size=" << sizeof(*x2) << endl;
}
void f3(int x3[123][4]) //形参数组行大小与实参不同
{   cout << "x3_size=" << sizeof(*x3) << endl;
}
int main()
{   int a[3][4];
    cout << "a_size=" << sizeof(a) << endl;
    f1(a);
    f2(a);
    f3(a);
}
```

```
a_size=48
x1_size=16   因为int[4]
x2_size=16   因为int[4]
x3_size=16   因为int[4]
```

§8. 指针进阶

1. 多维数组与指针

1.5. 用指向二维数组的指针做函数参数

★ 形参是指向m个元素组成的一维数组的指针变量

★ 形参是相应类型的二维数组

（行的大小可省略，本质上仍然是指向m个元素组成的一维数组的指针变量）

例：二维数组名做实参

```
void output(int (*p)[4])
{
    int i,j;
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            cout << *(*(p+i)+j)  << " ";
            cout << *(p[i]+j)  << " ";
            cout << p[i][j]  << " ";

    cout << endl;
}

int main()
{
    int a[3][4]={...};
    output(a);
    return 0;
}
```

int p[3][4]
int p[][4]
int p[123][4]
本质都是行指针变量

二维数组值的三种形式

实参是二维数组名
形参是指向m个元素的一维数组的指针变量

3. 一维数组与指针中

★ 对一维数组而言，数组的指针和数组元素的指针，其实都是指向数组元素的指针变量(特指0/任意i)，因此本质相同(基类型相同)

★ 数组名代表数组首地址，指针是地址，但本质不同 (sizeof(数组名)/sizeof(指针)大小不同)

本处：

★ 对二维数组而言，数组的指针是指向一维数组的指针，数组元素的指针是指向单个元素的指针，两者的本质是完全不同的(基类型不同)

§8. 指针进阶

1. 多维数组与指针

1.5. 用指向二维数组的指针做函数参数

★ 形参是指向m个元素组成的一维数组的指针变量

★ 形参是相应类型的二维数组

（行的大小可省略，本质上仍然是指向m个元素组成的一维数组的指针变量）

二维数组做函数参数的实参/形参的四种组合

```
//形参是二维数组名
void fun(int p[][4])
{
    ...
}
```

```
//形参是指向m个元素组成的
  一维数组的指针变量
void fun(int (*p)[4])
{
    ...
}
```

```
//实参是二维数组名
int main()
{   int a[3][4]={...};

    fun(a);
}
```

```
//实参是指向m个元素组成的
  一维数组的指针变量
int main()
{   int a[3][4]={...};
    int (*p)[4];
    p=a;
    fun(p);
}
```

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

程序(代码)区
静态存储区
动态存储区

程序(代码)区:存放程序的执行代码

由若干函数的代码组成，每个函数占据一段连续内存空间

每个函数的内存空间的起始地址，称为函数的地址(指针)

函数名代表函数的首地址

2.2. 用函数指针变量调用函数

指向函数的指针变量的定义：

数据类型 (*指针变量名)(形参表)

int (*p)(int,int);

是指针变量

指针变量指向函数,形参为两个int

数据类型int是函数的返回类型

使用：

赋初值：指针变量名 = 函数名　　不要参数表

调用：　指针变量名(函数实参表列)

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

```
//例：简单的函数调用
#include <iostream>
using namespace std;

int max(int x, int y)
{
    return (x>y?x:y);
}

int main()
{
    int a,b,m;
    cin >> a >> b;
    m=max(a,b);
    cout << "max=" << m << endl;
    return 0;
}
```

```
//例：简单的函数调用
#include <iostream>
using namespace std;

int max(int x, int y)
{
    return (x>y?x:y);
}

int main()
{
    int a,b,m;
    int (*p)(int,int);//定义指向函数的指针变量
    p=max;            //赋初值,不带参数
    cin >> a >> b;
    m=p(a,b);         //函数调用，带实参表
    cout << "max=" << m << endl;
    return 0;
}
```

p和*p都是函数的首地址
m=p(a,b);
m=(*p)(a,b);
都正确，但一般不用后者

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

```
#include <iostream>
using namespace std;

int fun()
{
    return 37;
}

int main()
{
    int (*p)();
    p = fun;
    cout << fun() << endl;
    cout << fun   << endl;
    cout << *fun  << endl;
    cout << p()   << endl;
    cout << p     << endl;
    cout << *p    << endl;
}
```

函数名/函数指针
1、带()不带()
2、加*不加*
的含义区别

?

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

★ 指向函数的指针的型参表声明时，与被调用函数的型参表类型、顺序、数量一致，是否带形参变量名，形参变量名称是否一致不作要求

```
int max(int x, int y) {   ...   }
main()
{
    int (*p)(int, int); //不带形参变量名
    p=max;
}
```

```
int max(int x, int y) {   ...   }
main()
{
    int (*p)(int x,int y); //形参变量名相同
    p=max;
}
```

```
int max(int x, int y) {   ...   }
main()
{
    int (*p)(int p,int q); //形参变量名不同
    p=max;
}
```

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

★ 指向函数的指针的型参表声明时，与被调用函数的型参表类型、顺序、数量一致，是否带形参变量名，形参变量名称是否一致不作要求

★ 指向函数的指针变量进行指针运算是无意义的

p+n　：编译出错

p++　：编译出错

p<q　：编译出错/不出错但无意义

*p　：编译不错但无意义

```
#include <iostream>
using namespace std;

int max(int x, int y)
{   return (x>y?x:y);
}

int main()
{
   int (*p)(int, int);
   p=max;
   p++;      //编译报错
   p=p-2;    //编译报错
   return 0;
}
```

```
#include <iostream>
using namespace std;
int max(int x, int y)
{  return (x>y?x:y);
}
int min(int x, int y)
{  return (x<y?x:y);
}
int main()
{  int (*p)(int, int);
   int (*q)(int, int);
   p=max;
   q=min;
   cout << (p<q) << endl; //输出0/1,无意义
   return 0;
}
```

```
#include <iostream>
using namespace std;
int max(int x, int y)
{  return (x>y?x:y);
}
int fun(int x)
{  return x;
}
int main()
{  int (*p)(int, int);
   int (*q)(int);
   p=max;
   q=fun;
   cout << (p<q) << endl;
   return 0;
}
```

编译出错，因为p/q指向的两个函数的形参表列及返回值不完全相同

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

2.3. 指向函数的指针做函数参数

★ 适用于在函数中每次调用不同的函数

★ 被调用的函数必须有相同的返回类型和形参表列

★ C++可通过重载函数、多态性与虚函数等方法解决同样的问题，因此C++中这种方法不常用(纯C使用)

```
void f1(int x, int y)
{
    cout << x+y << endl;
}
void f2(int x, int y)
{
    cout << x-y << endl;
}
void f3(int x, int y)
{
    cout << x*y << endl;
}
void fun( void (*f)(int, int) )
{
   int a=10,b=15;
   f(a, b);
}
int main()
{
   fun(f1);    //25
   fun(f2);    //-5
   fun(f3);    //150
   return 0;
}
```

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

2.3. 指向函数的指针做函数参数

2.4. 指向类对象的成员函数的指针

```
/* 6.3.指向普通函数的指针 */
#include <iostream>
using namespace std;

void fun()
{
    cout << "fun()" <<endl;
}

int main()
{
    void (*p)();
    p=fun;  //赋值，正确
    p();    //调用，正确
}
```

全局函数的指针：
(1) 返回类型匹配
(2) 形参表匹配

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour;
  public:
    Time() {     //构造
       hour=0;
       }
    void display() { //打印
       cout << hour << endl;
       }
};
int main()
{  Time t1;
   void (*p)();
   p=t1.display;//赋值,错误
   p();         //调用,错误
}
```

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour;
  public:
    Time() {     //构造
       hour=0;
       }
    void display() { //打印
       cout << hour << endl;
       }
};
int main()
{  Time t1;
   void (Time::*p)();
   p=&Time::display;//赋值,正确
   (t1.*p)();       //调用,正确
}
```

成员函数的指针：
(1) 返回类型匹配
(2) 形参表匹配
(3) 类匹配

§8. 指针进阶

2. 函数与指针

2.1. 函数的地址

2.2. 用函数指针变量调用函数

2.3. 指向函数的指针做函数参数

2.4. 指向类对象的成员函数的指针

定义：成员函数返回类型 (类::*指针变量名)(形参表)

赋值：指针变量名 = &类::成员函数名

★ 对象的成员函数必须是public

使用：

(对象名.*指针变量名)(实参表)

```
Time t1, t2;
void (Time::*p)();
p=&Time::display;
(t1.*p)()   ⇔ t1.display()
(t2.*p)()   ⇔ t2.display()
(t1.p)();   //错误，t1无p成员
```

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

含义：元素类型是指针的数组

定义：数据类型 *数组名[数组长度]

int *p[4];

是一个数组

数组的元素类型是指针

指针的基类型是int

★ 指针数组与指向m个元素的一维数组的指针的比较

int *p[4];

p是数组名，有4个元素，每个元素是int *

p+1实际+4，因为数组类型为指针

int (*p)[4];

p是指针变量名，指向由4个元素组成的一维数组

p+1实际+16，因为p的基类型为int*4

使用：保证数组的每个元素为基类型为数据类型的指针，使用时匹配即可，可进行所允许的任何运算

```
#include <iostream>
using namespace std;
int main()
{
    int a=10,b[3]={11,12,13},c=27, *d=&c;
    int *p[4] = {&a, b, &b[2], d};
    cout << *p[0] << endl;
    *(p[1] + 1) = 32;
    cout << b[1] << endl;
    cout << p[2] - b << endl;
    cout << (*p[3] - *p[0]) << endl;

    return 0;
}
```

?

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

二维字符数组：

char a[3][10] = {"china", "student", "s"};

a[0]	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

a[1]	2010	s
	2011	t
	2012	u
	2013	d
	2014	e
	2015	n
	2016	t
	2017	\0
	2018	
	2019	

a[2]	2020	s
	2021	\0
	2022	
	2023	
	2024	
	2025	
	2026	
	2027	
	2028	
	2029	

赋初值方法

字符串常量 "china"(无名)

a[0]	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

优点：(1) 与无名字符串常量分占不同空间

(2) 字符串的值可以修改

缺点：(1) 有空间浪费

(2) 若要交换元素(例如排序)，则需要要整体移动元素

交换的方法：

```
char tmp[10];
strcpy(tmp, a[0]);
strcpy(a[0],a[1]);
strcpy(a[1],tmp);
```

a[0]	2000	c
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

a[1]	2010	s
	2011	t
	2012	u
	2013	d
	2014	e
	2015	n
	2016	t
	2017	\0
	2018	
	2019	

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

一维指针数组：

```
char *a[3] = {"china", "student", "s"};
```

a	2000	3000
	2004	3100
	2008	3200

字符串常量 "china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

字符串常量 "student"(无名)

3100	s
3101	t
3102	u
3103	d
3104	e
3105	n
3106	t
3107	\0

字符串常量 "s"(无名)

3200	s
3201	\0

优点：(1) 节约空间

(2) 交换是只需交换指针值即可，效率高

缺点：(1) 用指针指向无名字符串常量，无法改变字符串的值

a	2000	3100
	2004	3000
	2008	3200

交换的方法：

```
char *tmp;
tmp  = a[0];
a[0] = a[1];
a[1] = tmp;
```

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

- 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
- 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

```
#include <iostream>
using namespace std;
int main()
{   char a[3][10]={"china","student","s"};
    cout << a[0] << endl;      china
    cout << a[1] << endl;      student
    cout << a[2] << endl;      s
    a[0][0]-=32;
    cout << a[0] << endl;      China
    cout << a[1] << endl;      student
    cout << a[2] << endl;      s
    return 0;
}
```

a[0]	~~2000~~	c=>C
	2001	h
	2002	i
	2003	n
	2004	a
	2005	\0
	2006	
	2007	
	2008	
	2009	

a[1]	2010	s
	2011	t
	2012	u
	2013	d
	2014	e
	2015	n
	2016	t
	2017	\0
	2018	
	2019	

a[2]	2020	s
	2021	\0
	2022	
	2023	
	2024	
	2025	
	2026	
	2027	
	2028	
	2029	

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

- 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
- 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

```
#include <iostream>
using namespace std;
int main()
{   char *a[3]= {"china","student","s"};
    cout << a[0] << endl;      china
    cout << a[1] << endl;      student
    cout << a[2] << endl;      s
    a[0][0]-=32; //Dev有warning且运行错，为什么？
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    return 0;
}
```

VS编译：error
Dev编译：warning

VS：常量转换为变量（error）

```
error C2440: “初始化”: 无法从“const char [6]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
error C2440: “初始化”: 无法从“const char [8]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
error C2440: “初始化”: 无法从“const char [2]”转换为“char *”
message : 从字符串文本转换将丢失 const 限定符(请参阅 /Zc:strictStrings)
```

Dev：常量转换为变量（Warning）

```
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
[Warning] ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
```

```
D:\WorkSpace\VS2022-demo\demo-cpp\demo.exe
china
student
s

--------------------------------
Process exited after 1.812 seconds with return value 3221225477
请按任意键继续. . .
```

字符串常量"china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

字符串常量"student"(无名)

3100	s
3101	t
3102	u
3103	d
3104	e
3105	n
3106	t
3107	\0

字符串常量"s"(无名)

3200	s
3201	\0

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

- 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
- 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

```
int main()
{
    char *a[3]= {"china","student","s"};
    char  b[10]="hello";
    cout << a[0] << endl;        china
    cout << a[1] << endl;        student
    cout << a[2] << endl;        s
    a[0] = b;            //a[0]存放数组b的首地址
    cout << a[0] << endl;        hello
    cout << a[1] << endl;        student
    cout << a[2] << endl;        s
    a[0][0]-=32;         //修改数组b[0]元素的值
    cout << a[0] << endl;        Hello
    cout << a[1] << endl;        student
    cout << a[2] << endl;        s
    return 0;
}
```

VS ： error
Dev： warning

报错同前例

a	2000	3000->3300
	2004	3100
	2008	3200

```
D:\WorkSpace\VS2022-demo\demo-cpp\demo.exe
china
student
s
hello
student
s
Hello
student
s

--------------------------------
Process exited after 0.0557 seconds with return value 0
请按任意键继续. . .
```

Dev有warning但可运行，且表面现象正常，实际呢？

字符串常量 "china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

字符串常量 "student"(无名)

3100	s
3101	t
3102	u
3103	d
3104	e
3105	n
3106	t
3107	\0

字符串常量 "s"(无名)

3200	s
3201	\0

数组b[10]

3300	h=>H
3301	e
3302	l
3303	l
3304	o
3305	\0
3306	
3307	
3308	
3309	

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

- 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
- 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

```
int main()
{   char  a[3][10] = {"china","student","s"};
    char  b[10]="hello";
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    a[0] = b;
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    a[0][0]-=32;
    cout << a[0] << endl;
    cout << a[1] << endl;
    cout << a[2] << endl;
    return 0;
}
```

编译
- 正确，执行
  - 正确，运行结果？
  - 错误，哪句运行出错，为什么？
- 错误，哪句错？为什么？

如何修改，使正确运行并且运行结果与上例相同？

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

★ 二维字符数组和一维指针数组的区别

- 二维字符数组分配实际的字符串存储空间，在执行过程中可以修改字符串任意位置的值
- 一维指针数组不分配实际字符串存储空间，只是指向字符串常量，在执行过程中字符串值不能改变

```
//例：字符串进行选择排序（指针数组形式）
#define N 4
void sort(char *name[], 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(name[k],name[j])>0)
                k=j;
        if (k!=i) {
            temp=name[i];
            name[i]=name[k];
            name[k]=temp;
            }
        }
}
int main()
{   int i;
    const char *name[N] = { "PHP", "C++", "Python","Java"};
    for (i=0; i<N; i++)
        cout << name[i] << endl;
    sort(name, N);
    for (i=0; i<N; i++)
        cout << name[i] << endl;
}
```

对比并体会红蓝部分差异

思考：
左右两侧name中，字符串的长度有限制吗？

```
//例：字符串进行选择排序（二维字符数组形式）
#define N 4
#define LEN 8
void sort(char name[][LEN], int n)
{
    char temp[LEN];
    int i,j,k;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if (strcmp(name[k],name[j])>0)
                k=j;
        if (k!=i) {
            strcpy(temp,name[i]);
            strcpy(name[i],name[k]);
            strcpy(name[k],temp);
            }
        }
}
int main()
{   int i;
    char name[N][LEN] = {"PHP", "C++", "Python", "Java"};
    for (i=0; i<N; i++)
        cout << name[i] << endl;
    sort(name, N);
    for (i=0; i<N; i++)
        cout << name[i] << endl;
}
```

§8. 指针进阶

3. 指针数组和指向指针的指针

3.1. 指针数组

3.2. 指向指针的指针

定义：数据类型 **指针变量名

int **p;

是指针变量

指向一个指针变量

该指针变量的基类型是int

使用：

int i=10, *t, **p;

t=&i;　（普通变量的地址）

p=&t;　（指针变量的地址）

定义时赋初值的写法：

int i=10,*t=&i,**p=&t;

i	10	2000 2001

t	2000	2100 2103

p	2100	2200 2203

p=地址（指向普通变量的指针变量的地址）

*p=地址(普通变量的地址)

**p=值（普通变量）

```
#include <iostream>
using namespace std;
int main()
{
    const char *a[3] = {"china", "student", "s"}, **p;
    p=a;
    cout << p << endl;
    cout << p+1 << endl;
    cout << (int *)(*p) <<endl;
    cout << *p << endl;
    cout << *p+3 << endl;
    cout << *(*p+3) << endl;
}
```

Dev可不要(有warning)

地址a	
地址a+4	
地址b	串首地址
china	输出串
na	输出串
n	输出字符

1、p+1只加了4，证明p不是china的地址(不够)

2、*p的值(地址b)，是无名字符串常量"china"的首址

3、由2反证1中的地址a应该是数组元素a[0]的地址

§8. 指针进阶

3. 指针数组和指向指针的指针

3.2. 指向指针的指针

```
char *a[3] = {"china", "student", "s"}, **p;
p=a;
```

p+1　　: a[1]的地址(地址2004)

p++　　: p指向a[1](p的值变为地址2004)

*p　　: 取a[0]的值3000(字符串"china"的首地址)

*(p+1) : 取a[1]的值3100(字符串"student"的首地址)

*p++　　: 取a[0]的值3000，p指向a[1](地址2004)

(*p)++ : 取a[0]的值3000,再++为3001(字符'h'的地址)

*p+3　　: 取a[0]的值3000,再+3为3003(字符'n'的地址)

*(*p+3): 取a[0]的值3000,再+3为3003(字符'n')

p	2100	2000

a	2000	3000
	2004	3100
	2008	3200

字符串常量
"china"(无名)

3000	c
3001	h
3002	i
3003	n
3004	a
3005	\0

字符串常量
"student"(无名)

3100	s
3101	t
3102	u
3103	d
3104	e
3105	n
3106	t
3107	\0

字符串常量
"s"(无名)

3200	s
3201	\0

§8. 指针进阶

4. const指针

4.1. 共用数据的保护

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到既能共享，又不会因误操作而改变，引入共用数据保护的概念

`#include <iostream>`
`using namespace std;`
 	
`void fun(int *p)`
`{`
`  *p=10;`
`}`
 	
`int main()`
`{`
`  int k=15;`
`  fun(&k);`
`}`
	`#include <iostream>`
	`using namespace std;`
	
	`void fun(int *p)`
	`{`
	`  if (*(p+5)=10) //原意是 *(p+5)==10`
	`}`
	
	`int main()`
	`{`
	`  int a[]={…};`
	`  fun(a);`
	`}`

通过指针，fun中改变了main的局部变量k的值，如果这种改变不是预期中的，则可能会带来错误	通过指针，fun中改变了main的数组a中元素的值

§8. 指针进阶

4. const指针

4.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　或　数据类型 const *指针变量名

作用：

★ 不能通过指针修改变量的值(仍可以通过变量修改)

★ 指针变量可以指向其它同类型变量(不必在定义时初始化)

★ 适用于不希望通过指针修改变量值的情况

```
#include <iostream>
using namespace std;
int main()
{
    int a=12, b=15;
    const int *p;//int const *p;
    p = &a;
    cout << *p << endl;
    *p = 10;
    a = 10;
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
    return 0;
}
```

a并不是常量，
但无法通过p改变a的值
=> 理解为p指向一个常量

1、哪一个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

问：假设 a=10；也不允许，
如何操作？
答：

```
#include <iostream>
using namespace std;

void fun(const int *x)
{
    x++    / x+=2    等操作：可以
    *x=10 / (*x)++ 等操作：不可以
}
int main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    fun(a);
    return 0;
}
```

保证在fun函数中仅能访问
而不会改变实参数组的值
(防止误操作)

假设main由甲完成，fun由乙完成，数组a
为传递的参数，甲希望乙只能读a而不能
修改a中的值

§8. 指针进阶

4. const指针

4.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　或　　数据类型 const *指针变量名

作用：

★ 指向常量的指针变量可以指向常变量、普通变量，但是普通指针不能指向常变量

```
int a;
const int b;
const int *p;
p = &a; //正确
p = &b; //正确
```

```
const int a = 10;
int *p1;
const int *p2;
p1 = &a; //编译错
p2 = &a; //正确
```

=> 推论：应用于函数的实形参对应，则规律如下：

形参	属性	实参	属性	正确性
普通指针	RW	普通变量地址/指针	RW	对
普通指针	RW	常变量地址/指针	R	错
常变量指针	R	普通变量地址/指针	RW	对
常变量指针	R	常变量地址/指针	RW	对

基本原则：赋权不能大于原权

```
void f(int *p)
{   return;
}
int main()
{   int x;
    f(&x);
}
```

?

```
void f(int *p)
{   return;
}
int main()
{   const int x = 10;
    f(&x);
}
```

?

```
void f(const int *p)
{   return;
}
int main()
{   int x;
    f(&x);
}
```

?

```
void f(const int *p)
{   return;
}
int main()
{   const int x = 10;
    f(&x);
}
```

?

```
void f(int *p)
{   return;
}
int main()
{
    int x;
    const int *p;
    p = &x;
    f(p);
}
```

?

§8. 指针进阶

4. const指针

4.3. 常指针

形式：数据类型 *const 指针变量名

作用：

★ 可以通过指针修改变量的值

★ 指针变量指向固定变量(必须在定义时初始化)后，不能再指向其它同类型变量

★ 适用于希望指针始终指向某个变量的情况

```
#include <iostream>
using namespace std;

int main()
{
    int a=12, b=15;
    int *const p = &a;   //定义时必须初始化
    cout << *p << endl;
    *p = 10;
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
    return 0;
}
```

1、哪一个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

```
#include <iostream>
using namespace std;
void fun(int *const x)
{
    x++    / x+=2                    等操作：不可以
    if (x+2 < 另一个指针)            等操作：可以
    *(x+2)=10 / (*x)++ / *x==10      等操作：可以
}
int main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,10}, *p=&a;
    fun(p);
    return 0;
}
```

保证在fun函数中p始终指向a
并能读写a数组而不能被改变
(防止误操作)

假设main由甲完成，fun由乙完成，指针p为传递的参数，
甲希望乙可以通过p读写，但不要改变p的指向

§8. 指针进阶

4. const指针

4.3. 常指针

形式：数据类型 *const 指针变量名

作用：

★ 可以通过指针修改变量的值

★ 指针变量指向固定变量(必须在定义时初始化)后，不能再指向其它同类型变量

★ 适用于希望指针始终指向某个变量的情况

```
void f(int *p)
{   return;
}
int main()
{   int x;
    int *const p =&x;
    f(p);
}
```

?

```
void f(int *const p)
{   return;
}
int main()
{   int x;
    int *const p = &x;
    f(p);
}
```

?

```
void f(int *p)
{   return;
}
int main()
{   int x;
    f(&x);
}
```

?

```
void f(int *const p)
{   return;
}
int main()
{   int x;
    f(&x);
}
```

?

★ 常指针不能指向常变量（右侧两个例子均编译错）

```
const int x = 10;
int *const p = &x;
```

```
void f(int *const p)
{  return;
}
int main()
{  const int x=10;
   f(&x);
}
```

问：如何解决?
答：见9.4

§8. 指针进阶

4. const指针

4.2. 指向常量的指针变量

4.3. 常指针

4.4. 指向常量的常指针

形式：const 数据类型 *const 指针变量名

作用：

★ 不能通过指针值修改变量的值

★ 指针变量指向固定变量(必须在定义时初始化)后，不能再指向其它同类型变量

★ 适用于既希望始终指向固定变量，又希望不能通过指针修改变量值的情况

```
#include <iostream>
using namespace std;
int main()
{   int a=12, b=15;
    const int *const p=&a; //必须定义时初始化
    cout << *p << endl;
    *p = 10;
    a = 10;
    cout << *p << endl;
    p = &b;
    cout << *p << endl;
    return 0;
}
```

1、哪两个语句会编译报错?
2、注释掉报错语句后，
三句cout的输出是什么？

```
void f(int const *const p)
{   return;
}
```

```
int main()
{   int x;
    f(&x);
}
```

?

```
int main()
{   const int x = 10;
    f(&x);
}
```

?

```
void f(int *p)
{   return;
}
```

```
int main()
{   int x;
    int const *const p = &x;
    f(p);
}
```

?

```
int main()
{   const int x = 10;
    int const *const p = &x;
    f(p);
}
```

?

§8. 指针进阶

4. const指针

4.1. 共用数据的保护

4.2. 指向常量的指针变量

形式：　　const 数据类型 *指针变量名

　　或　数据类型 const *指针变量名

4.3. 常指针

形式：数据类型 *const 指针变量名

4.4. 指向常量的常指针

形式：const 数据类型 *const 指针变量名

问：在引入const指针的情况下，实形参之间参数传递的基本规则?

答：按读写/只读方式区分，实形参的组合一共四种

实参只读 => 形参只读

实参只读 => 形参读写

实参读写 => 形参只读

实参读写 => 形参读写

哪种有错?

§8. 指针进阶

5. void指针类型

2. 变量与指针
2.4. 指针变量的++/--
★ void可以声明指针类型，但不能++/--
(void不能声明变量，但可以是函数的形参及返回值)

```
void  k;  ×  错误，不知道该给k分配几字节的空间
void *p;  ✓  正确，因为知道p大小是4字节
p++;      ×  错误，因为不知道基类型的大小
p--;      ×  错误，同上
```

3. 一维数组与指针
3.4. 指针法引用数组元素
3.4.3. 指向数组的指针变量的运算
C. 两个基类型相同的指针变量相减后与整数做比较运算
★ void型的指针变量不能进行相互运算(不知道基类型)

```
void *p, *q;
cout << (p+2)   << endl;   //编译错
cout << (q--)   << endl;   //编译错
cout << (p-q)   << endl;   //编译错
cout << (p<q+1) << endl;   //编译错
```

含义：指向空类型的指针变量

使用：

★ 不能直接通过void指针访问数据(不知道基类型)，必须强制转换为某种确定数据类型后才能访问

★ 非void型的指针可直接赋值给void类型，void类型赋值给非void类型时必须强制转换

```
#include <iostream>
using namespace std;
int main()
{
    int i=10, *p1=&i, *p3;
    void *p2;

    cout << p1 << endl;   //  地址i
    cout << *p1 << endl;  //  10
    cout << *p2 << endl;  //  编译错误
    p2 = p1;
    p3 = p2;  //  编译错误，改为：p3=(int *)p2
    cout << *p3 << endl;  //  10
    return 0;
}
```

§8. 指针进阶

6. 有关指针的数据类型和指针运算的小结

6.1. 各种类型的指针变量

```
int *p              : 指向整型简单变量/数组元素的指针变量
int *p[n]           : 指针数组，数组元素为int *类型
int (*p)[n]         : 指向含n个int元素的一维数组的指针变量
int *p()            : 返回值为int *类型的函数
int (*p)()          : 指向函数的指针(形参为空，返回int)
int **p             : 指向int *类型指针的指针变量
int const *p        : 指向常量的指针变量
int *const p        : 常指针
const int *const p  : 指向常量的常指针
void *p             : 基类型为void的指针
```

```
int *(*p)()     :

int *(*p)[n]    : 思考：
                  1、指出这4种情况中的p是(指针/数组/函数)？
                  2、如果是指针，指向什么？
int (*p[n])()   : 3、如果是数组，数组元素是什么类型？
                  4、如果是函数，函数的形参及返回类型是什么？
int *(*p[n])()  : (下面的示例全部正确)
```

```
int *fun() { ...; }
int main ()
{
    int *(*p)();
    p = fun;
    return 0;
}
```

```
int main ()
{   int *a[10],*b[3][10];
    int *(*p)[10];
    p = &a; //特别关注!!!
    p = b;
    return 0;
}
```

```
int fun() { ...; }
int main ()
{
    int (*p[10])();
    p[0] = fun;
    return 0;
}
```

```
int *fun() { ...; }
int main ()
{
    int *(*p[10])();
    p[0] = fun;
    return 0;
}
```

§8. 指针进阶

6. 有关指针的数据类型和指针运算的小结

6.2. 空指针NULL(复习)

★ 指针允许有空值 NULL(系统宏定义#define NULL 0)，表示不指向任何变量(若定义指针变量未赋初值，则随机指向)

```
NULL与空字符串的区别：
char *s1 = NULL; //s1是指针，存放地址0，地址0中的内容不一定是'\0'，
                 //即strlen(s1)不一定为0
char *s2 = "";   //s2是指针，存放一个长度为0的无名字符串常量的首地址(非0)，
                 //strlen(s2)为0
```

```
char s3[]="";    //正确
char s4[]=NULL;  //错，不能用无{}的一个数字初始化
```

```
#include <iostream>
using namespace std;
int main()
{   char s3[]="";
    char s4[]=NULL;//编译报错
}
```

error C2440: "初始化": 无法从"int"转换为"char []"
message : 没有转换为数组类型，但有转换为数组的引用或指针

§8. 指针进阶

6. 有关指针的数据类型和指针运算的小结

6.2. 空指针NULL(复习)

★ 指针允许有空值 NULL(系统宏定义#define NULL 0)，表示不指向任何变量(若定义指针变量未赋初值，则随机指向，称野指针)

★ 系统的字符串操作函数若传入参数为NULL则会出错

(包括 strcpy/strcat/strcmp/strlen/strncpy/strncmp等，以及未出现过的同类函数)

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    int len;
    len=strlen(s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcpy(s2, s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char s2[80]="Hello";
    strcat(s2, s1);
}
```

错

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char *s1 = NULL;
    char *s2 = NULL;
    int k=strcmp(s1,s2);
}
```

错

=> 自行实现类似功能的字符串处理函数时，可以对NULL进行特殊处理(具体见作业，这不是标准，只是为了强行与系统函数不同)

- 求长度时为0
- 复制、连接、拷贝时当做空串进行处理

§8. 指针进阶

6. 有关指针的数据类型和指针运算的小结

6.1. 各种类型的指针变量

6.2. 空指针NULL(复习)

6.3. 不同基类型指针的相互赋值(复习)

★ 不同类型的指针变量不能相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;
int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;
    p1 = p; //编译错
    p2 = p; //编译错
    cout << *p  << endl;
    cout << *p1 << endl;
    cout << *p2 << endl;

    return 0;
}
```

强制类型转换

```
#include <iostream>
using namespace std;
int main()
{
    long a=70000, *p=&a;
    short *p1;
    char  *p2;
    p1 = (short *)p;
    p2 = (char *)p;
    cout << *p  << endl;
    cout << *p1 << endl;
    cout << *p2 << endl;

    return 0;
}
```

70000=00000000 00000001 00010001 01110000

p	2000
p1	2000
p2	2000

a:低位在前存放	
2000	01110000
2001	00010001
2002	00000001
2003	00000000

70000 (2000-2003)
4464 (2000-2001)
p (2000)

§8. 指针进阶

6. 有关指针的数据类型和指针运算的小结

★ 不同类型的指针变量不能相互赋值，若要赋值，则需要进行强制类型转换

```
#include <iostream>
using namespace std;
int main()
{
    char s[]="abcd";
    int *p=(int *)s;
    cout << hex << *p << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{   int k=0x41424344;
    char *s=(char *)&k;
    for (int i=0; i<4; i++)
        cout << s[i] << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int k=0x41424344;
    char *s=(char *)&k;
    cout << s << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int k=0x424344;
    char *s=(char *)&k;
    cout << s << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    double k=-123.456;
    char *s=(char *)&k;
    for (int i=0; i<8; i++)
        cout << hex << int(s[i]) << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x=0x99, y=0x88;
    char *s=(char *)&y;
    for (int i=0; i<16; i++)
        cout << hex << int(s[i]) << endl;
    return 0;
}
```

运行这些代码，观察结果并思考为什么？

问题：
1、如何知道double型数据的存储(三段制)？
2、同一个数据，如何知道float和double的存储差异？
3、如何知道某种编译器下两个相邻的变量间隔几字节？
4、如何知道数组和某相邻变量间隔几字节？中间的间隔值是什么？
5、如何知道某函数的原始执行代码？
6、...

§9. 动态内存申请

1. 结构体类型的定义及使用(复习)

★ 结构体类型的声明

★ 字节对齐

★ 结构体变量的定义和初始化（普通变量、数组、指针、引用）

★ 指向结构体变量的指针和指向结构体变量中某个成员的指针

★ 结构体（struct）和类（class）的区别

§9. 动态内存申请

2. 指向结构体变量的指针与链表

2.1. 链式结构的基本概念

★ 数组的不足

1、大小必须在定义时确定，导致空间浪费

是否可以按需分配空间

2、占用连续空间，导致小空间无法充分利用

是否可以充分利用不连续的空间

3、在插入/删除元素时必须前后移动元素

插入/删除时能否不移动元素

★ 链表

不连续存放数据,用指针指向下一数据的存放地址

在数组/链表含有大量数据时：
1、频繁在任意位置插入/删除，哪种方式好？
2、频繁存取第i个元素的值，哪种方式好？（i随机）

例：数据1，2，3，4，5，分别存放在数组和链表中

存放5个元素：
数组：连续的20字节
链表：非连续的40字节
（每个结点的8字节连续）

问：本例中，存储相同数量数据，链表所占空间是数组的两倍，为什么不把这个问题当做是链表的缺点？

数组

2000 2003	1
2004 2007	2
2008 2011	3
2012 2015	4
2016 2019	5

链表

2000 2007	1	3000
2100 2107	3	3200
2500 2507	5	NULL
3000 3007	2	2100
3200 3207	4	2500

§9. 动态内存申请

例：数据1，2，3，4，5，分别存放在数组和链表中

所有教材上
链表的表示方法

§9.动态内存申请

2.指向结构体变量的指针与链表

2.1.链式结构的基本概念

结点：存放数据的基本单位

- 数据域：存放数据的值
- 指针域：存放下一个同类型节点的地址

链表：由若干结点构成的链式结构

表头结点：第一个结点

表尾结点：链表的最后一个结点，指针域为NULL(空)

头指针：指向链表的表头节点的指针

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

指向结构体自身的指针
成员类型不允许是自身的结构体类型，
但可以是指针(因为指针占用空间已知)

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000 2011	?	?

(结点)

b	3000 3011	?	?

(结点)

c	2500 2511	?	?

(结点)

head	2100 2103	?

p	2200 2203	?

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000 2011	31001 89.5	?

(结点)

b	3000 3011	31003 90	?

(结点)

c	2500 2511	31007 85	?

(结点)

head	2100 2103	?

p	2200 2203	?

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000 2011	31001 89.5	3000	(结点)
b	3000 3011	31003 90	2500	(结点)
c	2500 2511	31007 85	NULL	(结点)

head	2100 2103	2000
p	2200 2203	?

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

```
31001 89.5
```

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

```
31001 89.5
31003 90
```

例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

```
31001  89.5
31003  90
31007  85
```

§9. 动态内存申请

2. 指向结构体变量的指针与链表

2.1. 链式结构的基本概念

2.2. 链表与数组的比较

数组	链表
大小在声明时固定	大小不固定
处理的数据个数有差异时，须按最大值声明	根据需要随时增加/减少结点
内存地址连续，可直接计算得到某个元素的地址	内存地址不连续，必须依次查找
逻辑上连续，物理上连续	逻辑上连续，物理上不连续

§9.动态内存申请

3.内存的动态申请与释放

3.1.C中的相关函数

★ void *malloc(unsigned size);

- 申请size字节的连续内存空间,返回该空间首地址，对申请到的空间不做初始化操作
- 如果申请不到空间，返回NULL

★ void *calloc(unsigned n, unsigned size);

- 申请n*size字节的连续内存空间,返回该空间首地址，对申请到的空间做初始化为0（\0）
- 如果申请不到空间，返回NULL

★ void *realloc(void *ptr, unsigned newsize);

- 稍后见专题讨论

★ void free(void *p);

释放p所指的内存空间(p必须是malloc/calloc/realloc返回的首地址)

- 因为是系统库函数，需要包含头文件(VS系列可不要)

```
#include <stdlib.h> //C方式
#include <cstdlib>  //C++方式
```

3.2.C++中的相关运算符

★ 用 new 运算符申请空间(如果申请不到空间，new缺省会抛出bad_alloc异常，需要使用try-catch方式处理异常；也可以在new时加nothrow来强制禁用抛出异常并返回NULL)

- try-throw-catch称为C++的异常处理机制，后面再专题介绍

★ 用 delete 运算符释放空间

- 因为是运算符，不需要包含头文件

★ 用malloc/calloc等申请的空间，用free释放，用new申请的空间，用delete释放

§9. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符

普通变量	形式1：先定义指针变量，再申请
	int *p;
	p = (int *)malloc(sizeof(int));
	p = (int *)calloc(1, sizeof(int));
		形式1：先定义指针变量，再申请
		int *p;
		p = new int;

	形式2：定义指针变量的同时申请
	int *p = (int *)malloc(sizeof(int));
	int *p = (int *)calloc(1, sizeof(int));
		形式2：定义指针变量的同时申请
		int *p = new int;

	说明： 虽然初次申请时也可以用 p = (int *)realloc(NULL, sizeof(int)); 但一般不用	形式3：申请空间时赋初值 int *p; 或 int *p=new int(10); p=new int(10);

§9. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符

一维数组	形式1：先定义指针变量，再申请
	int *p;
	p = (int *)malloc(10*sizeof(int));
	p = (int *)calloc(10, sizeof(int));
		形式1：先定义指针变量，再申请
		int *p;
		p = new int[10]; //申请10个int型空间

	形式2：定义指针变量的同时申请空间
	char *name = (char *)malloc(10*sizeof(char));
	char *name = (char *)calloc(10, sizeof(char));
		形式2：定义指针变量的同时申请空间
		char *name = new char[10]; //申请10个char

	说明：
	虽然初次申请时也可以用
	p = (int *)realloc(NULL, 10*sizeof(int));
	但一般不用
		形式3：申请空间时赋初值
		● 动态申请的一维数组可以在申请时赋初值，方法为后面跟{}，{}前不要加=，且[]内必须有数，其余规则同一维数组定义时初始化
		● 对于字符类型，Dev/Linux不支持字符串方式初始化
		
		例：int *p;
		p = new int[5] {1,2,3,4,5}; //正确
		p = new int[5] {1,2,3,4,5,6}; //错误
		p = new int[5] {1,2}; //后面自动为0
		p = new int[5]={1,2,3,4,5}; //错误
		p = new int[] {1,2,3,4,5}; //VS正确+Dev/Linux错误
		
		char *s;
		s = new char[5]{'H','e','l','l','o'};//正确
		s = new char[5] {"Hello"}; //错误
		s = new char[6] {"Hello"}; //VS正确+Dev/Linux错误

§9. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符

二维数组	形式1：先定义指针变量，再申请
	int (*p)[4];
	p = (int (*)[4])malloc(3*4*sizeof(int));
	p = (int (*)[4])calloc(3*4, sizeof(int));
		形式1：先定义指针变量，再申请
		int *p;
		p = new int[3][4]; //申请3行4列，错!!!
		int (*p)[4];
		p = new int[3][4]; //申请3行4列，对!!!

	形式2：定义指针变量的同时申请空间
	int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int));
	int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));
		形式2：定义指针变量的同时申请空间
		float (*f)[4]=new float[3][4];

	说明：
	虽然初次申请时也可以用
	p = (int (*)[4])realloc(NULL, 3*4*sizeof(int));
	但一般不用
		形式3：申请空间时赋初值
		● 动态申请的二维数组可以在申请时赋初值，方法为后面跟双层{}，{}前不要加=，且[]内必须有数，其余规则同二维数组定义时初始化
		● 对于字符类型，Dev/Linux不支持字符串方式初始化
		
		例：int (*p)[3];
		p = new int[2][3] {1,2,3,4,5,6}; //VS正确+Dev/Linux错误
		p = new int[2][3] {{1,2,3},{4,5,6}}; //正确
		p = new int[2][3] {{1,2},{3,4,5,6}}; //错误
		p = new int[2][3] {1,4}; //VS正确+Dev/Linux错误
		p = new int[2][3] {{1},{4}}; //正确
		
		例：char (*p)[6];
		p = new char[2][6] {'A', 'B','C'}; //VS正确+Dev/Linux错误
		p = new char[2][6] {{'A'},{'B','C'}}; //正确
		p = new char[2][6] {"Hello","China"}; //VS正确+Dev/Linux错误
		p = new char[2][6]{"Hello1","China"}; //错误
		注：字符型在使用字符串方式初始化时，VS允许一层{}

§9. 动态内存申请

3. 内存的动态申请与释放

释放对象	C的函数方式	C++的运算符

普通变量	int *p = (int *)malloc(sizeof(int));
	int *p = (int *)calloc(1, sizeof(int));
	free(p);
		int *p = new int;
		delete p;

一维数组	int *p = (int *)malloc(10*sizeof(int));
	int *p = (int *)calloc(10, sizeof(int));
	free(p);
		int *p = new int[10];
		delete []p;
		● 某些资料上说可以 delete name，因为一维数组地址可理解为首元素地址，不加[]（说法不准确，必须加）
		● 对于int/char等基本类型的数组，加不加均正确；但对于用户自定义的class，则必须加，错误例子见后

二维数组	int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int));
	int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));
	free(p);
		int (*p)[4] = new int[3][4];
		delete []p;
		● 二维以上必须加一个[]，否则编译警告

§9. 动态内存申请

3. 内存的动态申请与释放

释放对象	C++的运算符
一维数组	int *p = new int[10]; delete []p; ● 某些资料上说可以 delete name，因为一维数组地址可理解为首元素地址，不加[]（说法不准确，必须加） ● 对于int/char等基本类型的数组，加不加均正确；但对于用户自定义的class，则必须加，错误例子见后

```
#include <iostream>
using namespace std;

int main()
{
    int *p = new int[10];
    delete []p;

    return 0;
}
//篇幅问题，假设申请成功
```

```
#include <iostream>
using namespace std;

int main()
{
    int *p = new int[10];
    delete p;

    return 0;
}
//篇幅问题，假设申请成功
```

§9. 动态内存申请

3. 内存的动态申请与释放

释放对象	C++的运算符

一维数组	`int *p = new int[10];`
	`delete []p;`
	● 某些资料上说可以 delete name，因为一维数组地址可理解为首元素地址，不加[]（说法不准确，必须加）
	● 对于int/char等基本类型的数组，加不加均正确；但对于用户自定义的class，则必须加，错误例子见后

```
#include <iostream>
using namespace std;

class Time {
  private:
    int hour, minute, second;
  public:
    Time(int h=0,int m=0,int s=0);
    ~Time();
};
Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    second = s;
    cout << "Time Begin" << hour << endl;
}
Time::~Time()
{
    cout << "Time End" << hour <<endl;
}
```

```
int main()
{
    Time *t1 = new Time[10] {1,2,3,4,5,6,7,8,9,10};
    delete t1;
    return 0;
} //篇幅问题，假设申请成功
```

```
int main()
{
    Time *t1 = new Time[10] {1,2,3,4,5,6,7,8,9,10};
    delete []t1;
    return 0;
} //篇幅问题，假设申请成功
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：可通过强制类型转换将void型的指针转为其它类型

★ C++ ：申请时自动确定类型

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    p = (int *)malloc(10*sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
        }
    cout << *p << endl; //观察运行结果，是否进行了初始化
    free(p);
    return 0;
}
```

强制类型转换

申请10个int型的变量空间可以直接写成malloc(40),但不建议，因为适应型差

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    p = (int *)calloc(10, sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
        }
    cout << *p << endl; //观察运行结果，是否进行了初始化
    free(p);
    return 0;
}
```

强制类型转换

申请10个int型的变量空间可以直接写成calloc(10,4),但不建议，因为适应型差

```
int main()
{   int *p;
    p = new(nothrow) int[10];
    if (p==NULL) {
       cout << "No Memory" << endl;
       return 0;
       }
    ...
    delete p;
    ...
    return 0;
}
```

malloc(10*sizeof(int))
calloc(10,sizeof(int))
realloc(NULL,10*sizeof(int))
都表示申请连续的40字节空间，
结果一样，只是表示方式有差别
以及是否初始化有差别

§9. 动态内存申请

3. 内存的动态申请与释放

★ 静态数据区、动态数据区、动态内存分配区(称为堆空间)的地址各不相同

例：观察下列程序的输出

```
#include <iostream>
#include <cstdlib>
using namespace std;
int a;
int main()
{
    int b;
    int *c;
    c = (int *)malloc(sizeof(int)); //一个int
    if (c==NULL) {
        cout << "申请int失败" << endl;
        return -1;
        }
    cout << &a <<endl;
    cout << &b <<endl;
    cout << &c << ' ' << c <<endl;
    free(c);

    return 0;
}
```

问：静态数据区/动态数据区/堆空间的大小如何？如何验证？

§9. 动态内存申请

3. 内存的动态申请与释放

★ 打开Windows的任务管理器，观察下列程序的运行结果，理解"动态申请与释放"的概念

例：观察下列程序的输出

如果是Linux下测试，则使用top命令观察内存占用，具体请自行查阅资料

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    char *p;
    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB
    if (p==NULL) {
        cout << "申请空间失败，请减少申请值后重试" << endl;
        return -1;
        }
    cout << "申请完成，请在任务管理器中观察内存占用" <<endl;
    getchar(); //暂停，不释放内存

    free(p);
    cout << "释放完成，请在任务管理器中观察占用" <<endl;
    getchar(); //暂停，不退出程序
    return 0;
}
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：动态申请返回的指针可以进行指针运算，但释放时必须给出申请返回时的首地址，否则释放时会出错

（以下几种情况均是编译不错执行错，用多编译器观察运行结果）

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i, *p;
    p = &i;
    free(p);
    return 0;
}
                    //p不是动态申请的空间
```

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;

int main()
{
    int i, *p;
    p = &i;
    delete p;
    return 0;
}
                    //p不是动态申请的空间
```

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    int *p;
    p=(int*)malloc(sizeof(int));//未判断
    p++;
    free(p);
    return 0;
}                   //p已不指向动态申请的空间
```

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;

int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    p++;
    delete p;
    return 0;
}                   //p已不指向动态申请的空间
```

特别说明：

1、虽然申请一个int空间不可能申请失败，但从程序规范角度出发，要求每次申请后均需要判断申请是否成功

2、本例及后续课件中，为了节约空间，部分示例程序省略了是否成功的判断，特此说明

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：动态内存申请的空间若不释放，则会造成内存泄露，这种情况不会导致即时错误，但最终会耗尽内存

```
#include <iostream>
#include <cstdlib>   //malloc系列函数用
using namespace std;
int main()
{
    char *p;
    int num = 0;
    while(1) {
        p = (char *)malloc(1024*1024*sizeof(char));
        if (p==NULL)
            break;
        num ++;
        }
    cout << num << " MB" << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;

int main()
{
    char *p;
    int num = 0;
    while(1) {
        p = new(nothrow) char[1024*1024];
        if (p==NULL)
            break;
        num ++;
        }
    cout << num << " MB" << endl;
    return 0;
}
```

耗尽内存的例子：
1、每次申请1MB空间
2、申请完成后不释放，且p不再指向，导致内存泄露
3、循环1-2至内存耗尽

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：动态内存申请的空间若不释放，则会造成内存泄露，这种情况不会导致即时错误，但最终会耗尽内存

```
#include <iostream>
using namespace std;
int main()
{
    char *p;
    int count = 0;
    while (1) {
        try {
            p = new char[1024 * 1024];
        }
        catch (const bad_alloc &mem_fail) {
            cout << mem_fail.what() << endl; //打印原因
            break;
            }
        count++;
        }
    cout << count << " MB" << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char *p;
    int count = 0;
    try {
        while (1) {
            p = new char[1024 * 1024];
            count++;
            }
        }
    catch (const bad_alloc &mem_fail) {
        cout << mem_fail.what() << endl; //打印原因
        }
    cout << count << " MB" << endl;
    return 0;
}
```

耗尽内存的例子：
1、每次申请1MB空间
2、申请完成后不释放，且p不再指向，导致内存泄露
3、循环1-2至内存耗尽

在新版C++标准中，new申请失败会抛出异常bad_alloc，需要使用try-catch来处理异常

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：动态内存申请的空间若不释放，则会造成内存泄露，这种情况不会导致即时错误，但最终会耗尽内存（坚决反对此种用法，且不是所有的操作系统都支持）

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *p;
    p=(int *)malloc(...);
    ...;
    return 0;
}
```

p所申请的空间在程序运行结束后由操作系统回收

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;

int main()
{
    int *p;
    p = new ...;
    ...;
    return 0;
}
```

p所申请的空间在程序运行结束后由操作系统回收

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{   int *p;
    /* 假设ATM机取款 */
    for(;;) {
        ...; //等待用户刷卡
        p=(int*)malloc(...);
        ...;
        }
    return 0;
}
```

会逐渐耗尽内存

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;

int main()
{   int *p;
    /* 假设ATM机取款 */
    for(;;) {
        ...; //等待用户刷卡
        p = new ...;
        ...;
        }
    return 0;
}
```

会逐渐耗尽内存

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：对简单变量、一维/多维数组没有分别，只算总大小

★ C++ ：不同情况申请方法不同

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int));//未判断
    *p = 10;
    cout << *p << endl;
    free(p); //记得释放
    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    *p = 10;
    cout << *p << endl;
    delete p; //记得释放
    return 0;
}
```

申请一个int型空间

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：对简单变量、一维/多维数组没有分别，只算总大小

★ C++ ：不同情况申请方法不同

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i, *p;
    p = (int *)malloc(10*sizeof(int));
    for(i=0; i<10; i++)
        p[i] = (i+1)*(i+1); //赋初值
    for(i=0; i<10; i++)
        cout << *(p+i) << endl;//打印
    free(p); //记得释放
    return 0;
}   //未判断申请是否成功
```

```
#include <iostream>

using namespace std;

int main()
{
    int i, *p;
    p = new(nothrow) int[10];
    for(i=0; i<10; i++)
        p[i] = (i+1)*(i+1); //赋初值
    for(i=0; i<10; i++)
        cout << *(p+i) << endl; //打印
    delete []p; //记得释放
    return 0;
}   //未判断申请是否成功
```

申请10个int型空间，
当一维数组用
指针法/下标法均可

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：对简单变量、一维/多维数组没有分别，只算总大小

★ C++ ：不同情况申请方法不同

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i, *p, *head;
    p = (int *)malloc(10*sizeof(int));
    head = p;
    for(i=0; i<10; i++)
        *p++ = (i+1)*(i+1); //赋初值
    for(p=head; p-head<10; p++)
        cout << *p << endl; //打印
    free(head); //记得释放
    return 0;
}   //未判断申请是否成功
```

```
#include <iostream>

using namespace std;

int main()
{
    int i, *p, *head;
    p = new(nothrow) int[10];
    head = p;
    for(i=0; i<10; i++)
        *p++ = (i+1)*(i+1); //赋初值
    for(p=head; p-head<10; p++)
        cout << *p << endl; //打印
    delete []head; //记得释放
    return 0;
}   //未判断申请是否成功
```

申请10个int当一维数组用
用head记住申请的首地址，
便于复位和释放，p可++/--

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：对简单变量、一维/多维数组没有分别，只算总大小

★ C++ ：不同情况申请方法不同

申请12个int型空间
当做二维数组使用
指针法/下标法均可

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i, j, (*p)[4];
    p=(int (*)[4]) malloc(3*4*sizeof(int));
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(i=0; i<3; i++) {
        for(j=0; j<4; j++)
            cout << *(*(p+i)+j) << ' ';
        cout << endl; //每行加回车
        }
    free(p); //记得释放
    return 0;
}   //未判断申请是否成功
```

```
#include <iostream>

using namespace std;

int main()
{
    int i, j, (*p)[4];
    p = new(nothrow) int[3][4];
    for(i=0; i<3; i++)
       for(j=0; j<4; j++)
          p[i][j] = i*4+j; //赋初值
    for(i=0; i<3; i++) {
       for(j=0; j<4; j++)
          cout << *(*(p+i)+j) << ' ';
       cout << endl; //每行加回车
       }
    delete []p; //记得释放
    return 0;
}   //未判断申请是否成功
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ C　：对简单变量、一维/多维数组没有分别，只算总大小

★ C++ ：不同情况申请方法不同

申请12个int当二维使用
p为行指针,p_element为
元素指针,head记住首址

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i, j, (*p)[4], (*head)[4], *p_element;
    head = p = (int (*)[4]) malloc(3*4*sizeof(int));
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(p=head; p-head<3; p++) {
        for(p_element=*p; p_element-*p<4; p_element++)
            cout << *p_element << ' ';
        cout << endl; //每行加回车
        }
    free(head); //记得释放
    return 0;
}   //未判断申请是否成功
```

```
#include <iostream>

using namespace std;

int main()
{
    int i, j, (*p)[4], (*head)[4], *p_element;
    head = p = new(nothrow) int[3][4];
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(p=head; p-head<3; p++) {
        for(p_element=*p; p_element-*p<4; p_element++)
            cout << *p_element << ' ';
        cout << endl; //每行加回车
        }
    delete []head; //记得释放
    return 0;
}   //未判断申请是否成功
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：动态申请的内存，只能通过首指针释放一次，若重复释放，则会导致运行出错

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int));//未判断
    *p = 10;
    cout << *p << endl;
    free(p); //释放
    free(p); //再次释放，致运行出错

    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    *p = 10;
    cout << *p << endl;
    delete p; //释放
    delete p; //再次释放，致运行出错

    return 0;
}
```

重复释放导致错误

§9. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ ：如果出现需要嵌套进行动态内存申请的情况，则按从外到内的顺序进行申请，反序进行释放

嵌套申请
先student变量，再name成员

```
#include <iostream>
#include <cstdlib>
using namespace std;

struct student {
   int num;
   char *name;
};
int main()
{
    student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

嵌套申请
先student变量，再name成员

```
#include <iostream>

using namespace std;

struct student {
   int num;
   char *name;
};
int main()
{
    student *s1;
    s1 = new(nothrow) student; //申请8字节
    s1->name = new(nothrow) char[6];//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    delete []s1->name;//释放6字节
    delete s1;//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

s1	2000 2003	???

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//释放6字节
    free(s1);//释放8字节
    return 0;
}   //为节约篇幅，未判断申请是否成功
```

s1自身所占4字节
由操作系统回收

free的顺序不能反

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

函数形式：

```
void *realloc(void *ptr, unsigned newsize);
```

- 表示为指针ptr重新申请newsize大小的空间
- ptr必须是malloc/calloc/realloc返回的指针
- 如果ptr为NULL，则等同于malloc
- 如果ptr非NULL，newsize为0，则等同于free，并返回NULL
- 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间
- 对申请到的空间不做初始化操作
- 若申请不到，则返回NULL（此时已有指针ptr不释放）

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 如果ptr为NULL，则等同于malloc
- 对申请到的空间不做初始化操作

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *p;
    p = (int *)realloc(NULL, 10 * sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
        }

    for(int i=0; i<10; i++)
        cout << p[i] << endl;
    free(p);
    return 0;
}
```

强制类型转换

等价于 malloc(10 * sizeof(int))

观察运行结果，
是否进行了初始化

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 表示为指针ptr重新申请newsize大小的空间
- ptr必须是malloc/calloc/realloc返回的指针
- 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间

此处换成 ++p / p+1等形式，
多编译器观察程序的运行结果

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{   int i, *p, *q;
    p = (int *)malloc(10 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << endl; //地址
    for (i=0; i<10; i++)
        p[i] = i*i;  //为10个数赋初值

    q = (int *)realloc(p, 20 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << ' ' << q << endl; //观察地址是否相同
    for (i=0; i<20; i++)
        cout << p[i] << ' '; //观察前10个和后10个数
    cout << endl;
    free(q);
    return 0;
}
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int *p, *q;
    p = (int *)malloc(10 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << endl;

    q = (int *)realloc(p, 20 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << ' ' << q << endl;

    free(p);
    free(q);

    return 0;
}
```

1、多编译器观察程序的运行结果
2、注释掉free(p)，再观察结果
3、此处换成5(小于原大小即可)，再重复1、2

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 如果ptr非NULL，newsize为0，则等同于free，并返回NULL

```
//先打开Windows的任务管理器，再观察程序的运行

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    char *p, *q;

    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB，此处要保证成功
    if (p == NULL) {
        cout << "申请空间失败，请减少申请值后重试" << endl;
        return -1;
        }
    cout << "申请完成，请在任务管理器中观察内存占用" <<endl;
    getchar(); //暂停，不释放内存

    q = (char *)realloc(p, 0); //0字节
    cout << (q==NULL ? "NULL" : q) << endl; //NULL不能直接打印
    cout << "realloc 0字节完成，请在任务管理器中观察内存占用" <<endl;
    getchar(); //暂停，不退出程序

    return 0;
}
```

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 若申请不到，则返回NULL（此时已有指针ptr不释放）

realloc的不正确用法(网上常见)：
传入指针和返回指针用同一个时，
一旦申请失败，原内存就丢失了!!!!!

```
//先打开Windows的任务管理器，再观察程序的运行

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    char *p, *q;

    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB，此处要保证成功
    if (p == NULL) {
        cout << "申请空间失败，请减少申请值后重试" << endl;
        return -1;
        }
    cout << "申请完成，请观察内存占用" <<endl;
    getchar(); //暂停，不释放内存

    q = (char *)realloc(p, 2048U * 1024 * 1024 * sizeof(char)); //2GB，此处要保证失败，如果不失败，继续增大值
    if (q==NULL) //如果不提示失败，2048U继续增大
        cout << "realloc失败，请观察内存占用" <<endl;
    getchar(); //暂停，不退出程序

    free(p);
    return 0;
}
```

问题：为什么加U?

§9. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

★ C++中没有类似于realloc的renew，如果需要扩大/缩小原动态申请空间，要自己处理

```
int *renew(int *p, int oldsize, int newsize)
{
   int *pnew;
   pnew = new(nothrow) int[newsize];//申请新
   for(i=0; i<oldsize; i++)    //原内容=>新
       pnew[i] = p[i];
   delete p;    //释放原空间(main中的new int [100])
   p = pnew;    //原指针p指向新空间
   return p;    //pnew已无用，但不能delete
}

main中:  int *p = new int[100];
         ...
         p = renew(p, 100, 150);
         ...
         delete []p;//释放空间(renew中的new int[150])
```

左侧是一个不完整的renew，缺以下情况：

- newsize<oldsize的情况
- 申请失败的情况
- 其它基类型的指针(可重载/模板解决)

重要提示：

由renew这个例子可看出，C/C++的动态内存申请和释放，可能会在不同函数间进行；因此，在大型程序中要做到无内存泄露是一件很困难的事情，这也是C/C++相比较于其它语言的难点所在!!!

§9. 动态内存申请

3. 内存的动态申请与释放

★ 在C/C++的动态申请混合使用时，可能会出现问题

```
//例：C++的动态申请，内嵌string类（C++特有）
#include <iostream>
#include <string>    //C++特有的string类需要
using namespace std;

struct student {
   string name; //C++特有的string类
   int num;
   char sex;
};

int main()
{
    student *p;
    p = new(nothrow) student;
    if (p==NULL)    //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num  = 10123;
    p->sex  = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    delete p;

    return 0;
}
```

本例运行正确

```
//思考：下面的例子说明了什么？
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1;
    cout << sizeof(s1) << endl;
    s1 = "***";  //此处长度超sizeof
    cout << sizeof(s1) << endl;
    return 0;
}
```

```
//例：变化，C方式的动态申请，内嵌string类（C++特有）
#include <iostream>
#include <string>    //C++特有的string类需要
#include <stdlib.h>
using namespace std;

struct student {
   string name; //C++特有的string类
   int num;
   char sex;
};

int main()
{
    student *p;
    p = (student *)malloc(sizeof(student));
    if (p==NULL)    //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num  = 10123;
    p->sex  = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    free(p);

    return 0;
}
```

多编译器运行，哪些编译器下运行错误？表现是什么？为什么？
（提示：和左侧的思考题、下面的建议结合在一起思考）

建议：
C++下的动态内存申请不建议采用C函数方式，不要为了realloc的便捷性而降级

§9. 动态内存申请

3. 内存的动态申请与释放

例：用动态内存申请方式建立一个有5个结点的链表，学生的基本信息从键盘进行输入

假设键盘输入为

```
Zhang 1001 m
Li    1002 f
Wang  1003 m
Zhao  1004 m
Qian  1005 f
```

```
struct student {
    string name;
    int num;
    char sex;
    struct student *next;//指向结构体自身的指针(下个结点)
    };
```

成员类型不允许是自身的结构体类型，
但可以是自身结构体类型的指针
(因为指针占用空间已知)

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

初始状态

head	2000 2003	???
p	2100 2103	???
q	2200 2203	???

例：用动态内存申请方式建立一个有5个结点的链表

i=0的循环

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

i=0的循环

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

i=0的循环

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

i=0的循环

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=0的循环

一般教科书的表示

例：用动态内存申请方式建立一个有5个结点的链表

i=1的循环

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=1的循环

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=1的循环

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=1的循环

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=1的循环

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

i=1的循环

例：用动态内存申请方式建立一个有5个结点的链表

i=2-4自行画图

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的
        cin >> p->name >> p->num >> p->sex;
        p->next = NULL;
        }
```

i=4的循环结束

例：用动态内存申请方式建立一个有5个结点的链表

i=4的循环结束

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

一般教科书的表示

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        if (i>0)
            q=p;
        p = new(nothrow) student;
        if (p==NULL)
            return -1;
        if (i==0)
            head = p; //head指向第1个结点
        else
            q->next = p;
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
        p->next = NULL;
        }
```

循环完成

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

刚才建立链表的循环

注意：不能用free

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
```

```
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
```

Zhang 1001 m

```
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
```

```
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
```

后续输出自行画图理解

```
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
```

```
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
```

p:^

最后一个结点输出

循环结束

```
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

Zhang	1001	m
Li	1002	f
Wang	1003	m
Zhao	1004	m
Qian	1005	f

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

后续结点释放自行画图理解

例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;  int i;
    for(i=0; i<5; i++) {
        }
    p=head; //p复位，指向第1个结点
    while(p!=NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p=p->next;
        }
    p=head; //p复位，指向第1个结点
    while(p) {  //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
        }
    return 0;
}
```

最后一个结点被释放后

循环结束，new申请的5个空间已被释放，指针变量head/p/q自身不是动态申请空间，由操作系统回收

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

~~C语言方法：~~
```
Time *p;
申请：p = (Time *)malloc(sizeof(Time));
      p = (Time *)malloc(10*sizeof(Time));
      if (p==NULL) { ... }
释放：free(p); //统一方法释放 单Time/Time数组
```

★ C++中一般不建议使用C方法动态申请

- C方式动态内存申请和释放时不会调用构造和析构函数(见4.2例)
- 前例中，struct中有string类对象，则 malloc/free 会出错

C++方法：
```
Time *p;
申请：p = new(nothrow) Time;
      p = new(nothrow) Time[10];
      if (p==NULL) { ... }
释放：delete p;    //释放单Time
      delete []p; //释放Time数组
```

★ C++中delete时，只要是数组，必须加[]

- VS系列编译器会运行出错
- GNU系列(DevC++/Linux)虽然表面不出错，但若含有动态内存申请，则因为不调用析构函数，仍会导致内存丢失

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

4.2. 动态申请对象的构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int
      s=0);
      ~Time();
   };
Time::Time(int h, int m, int s)
{   hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
int main()
{
    cout << "main begin" <<endl;
    Time *t1 = new time;
    cout << "new end" <<endl;
    delete t1;
    cout << "main end" <<endl;
}
```

```
main begin
Time Begin
new end
Time End
main end
```

Microsoft Visual Studio 调试控制台

```
main begin
Time Begin
new end
Time End
main end
```

```
int main()
{
    cout << "main begin" <<endl;
    Time *t1 = (Time *)malloc(sizeof(Time));
    cout << "new end" <<endl;
    free(t1);
    cout << "main end" <<endl;
}
```

★ C++下采用C语言方式的动态内存申请，不调用构造及析构函数

main begin
~~Time Begin~~
new end
~~Time End~~
main end

Microsoft Visual Studio 调试控制台

```
main begin
new end
main end
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

4.2. 动态申请对象的构造函数与析构函数的调用时机

★ new时调用构造函数，delete时调用析构函数

★ C++下采用C语言方式的动态内存申请，不调用构造及析构函数(淘汰)

构造函数与析构函数的调用时机

构造函数：

★ 自动对象(形参)　　：函数中变量定义时

★ 静态局部对象　　：第一次调用时

★ 静态全局/外部全局对象：程序开始时

★ 动态申请的对象　　：荣誉课内容(略)

析构函数：

★ 自动对象(形参)　　：函数结束时

★ 静态局部对象　　：程序结束时(在全局之前)

★ 静态全局/外部全局对象：程序结束时

★ 动态申请的对象　　：荣誉课内容(略)

main开始前

main结束后

```
//例：变化，C方式的动态申请，内嵌string类（C++特有）
#include <iostream>
#include <string>    //C++特有的string类需要
#include <stdlib.h>
using namespace std;

struct student {
   string name; //C++特有的string类
   int num;
   char sex;
};

int main()
{
    student *p;
    p = (student *)malloc(sizeof(student));
    if (p==NULL)    //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num  = 10123;
    p->sex  = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    free(p);

    return 0;
}
```

更进一步的解释：

★ malloc不激活student的构造函数（缺省的无参空体，自然也不会激活成员string name的构造函数，导致p->name访问时的内存错误）

前例：

多编译器运行，哪些编译器下运行错误？表现是什么？为什么？

建议：

C++下的动态内存申请不建议采用C函数方式，不要为了realloc的便捷性而降级

§9. 动态内存申请

4. 含动态内存申请内存的类和对象
4.1. 对象的动态建立和释放
4.2. 动态申请对象的构造函数与析构函数的调用时机
4.3. 在构造和析构函数中进行动态内存的申请与释放

析构函数

引入：在对象被撤销时(生命期结束)时被自动调用，完成一些善后工作(主要是内存清理)，但不是撤销对象本身

形式：

~类名();

★ 无返回值(非void,也不是int)，无参，不允许重载
★ 对象撤销时被自动调用，用户不能显式调用
★ 析构函数必须公有
★ 若不指定析构函数，则系统缺省生成一个析构函数，形式为无参空体
★ 若用户定义了析构函数，则缺省析构函数不再存在
★ 析构函数既可以体内实现，也可以体外实现
★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数

```
class Time {
   ...
   public:
      Time();
      ~Time(); //声明
   };
Time::Time()
{  ...
}
Time::~Time() //体外实现
{  cout << "Time End" << endl;
}
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数

★ 在有数据成员需要动态内存申请的情况下，也可以不定义析构函数而通过其他方法释放(不提倡)

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour;
        int minute;
        int sec;
        char *s;
    public:
        Time();
        ~Time();
};
Time::Time()
{   hour   = 0;
    minute = 0;
    sec    = 0;
    s = new char[80];//申请
}
Time::~Time()
{   delete []s; //释放
}
int main()
{
    Time t1; //不需要显式调用构造，自动申请
    ...
}//不需要显式调用析构，自动释放
```

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour;
        int minute;
        int sec;
        char *s;
    public:
        Time();
        Release();
}; //未定义析构
Time::Time()
{   hour   = 0;
    minute = 0;
    sec    = 0;
    s = new char[80];
}
Time::Release()
{   delete []s; //释放
}
int main()
{  Time t1; //不需要显式调用构造，自动申请
   ...
   t1.Release();
}//必须显式调用Release()
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数

★ 在有数据成员需要动态内存申请的情况下，也可以不定义析构函数而通过其他方法释放(不提倡!!!)

★ 不要与对象的动态申请混淆

new申请
16字节

delete释放
16字节

问：t1占的4个字节，谁负责释放？
答：操作系统
=>基本规则：谁负责分配，谁负责释放
(适合大程序的分工与组织)

构造函数
申请80字节

析构函数
释放80字节

隐含但很明确的规则：
new Time和构造函数
中的new，哪个在前？

隐含但很明确的规则：
delete t1和析构函数
中的delete，哪个在前？

```
int main()
{
    Time *t1 = new Time; //申请
    cout << "main begin" <<endl;
    delete t1; //释放
    cout << "main end" <<endl;
}
```

```
#include <iostream>
using namespace std;

class Time {
    private:
        int hour;
        int minute;
        int sec;
        char *s;
    public:
        Time();
        ~Time();
};
Time::Time()
{
    hour   = 0;
    minute = 0;
    sec    = 0;
    s = new char[80]; //申请
}
Time::~Time()
{
    delete []s;  //释放
}
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数

★ 在有数据成员需要动态内存申请的情况下，也可以不定义析构函数而通过其他方法释放(不提倡!!!)

★ 不要与对象的动态申请混淆

★ 如果出现需要嵌套进行动态内存申请的情况，则按定义顺序进行申请，反序进行释放

```
#include <iostream>
using namespace std;

struct student {
   int num;
   char *name;
};

int main()
{
    student *s1;
    s1 = new(nothrow) student; //申请8字节
    s1->name = new(nothrow) char[6];//申请6字节

    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;

    delete []s1->name;//释放6字节
    delete s1;//释放8字节

    return 0;
}  //为节约篇幅，未判断申请是否成功
```

前例改写，可有效简化main的复杂度

```
#include <iostream>
using namespace std;

struct student {
    int num;
    char *name;
    student()
    {
        name = new char[6]; //申请6字节
    }
    ~student()
    {
        delete []name; //释放6字节
    }
};
int main() //当main中多次对student申请/释放时
{
   student *s1;
   s1 = new(nothrow) student; //申请8字节
   s1->num = 1001;
   strcpy(s1->name, "zhang");
   cout << s1->num << ":" << s1->name << endl;
   delete s1;//释放8字节
   return 0;
}
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

对象的赋值

含义：将一个对象的所有数据成员的值对应赋值给另一个已存在对象的数据成员

形式：类名 对象名1，对象名2;

...

对象名1=对象名2;//执行语句

```
Time t1(14,15,23), t2;

t2=t1;
```

★ 两个对象属于同一个类，通过赋值语句实现(不能是定义时赋初值)

★ 系统默认的赋值操作是将右对象的全部数据成员的值对应赋给左对象的全部数据成员(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现(通过=运算符的重载实现，荣誉课内容)

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int  a;
    int  b;
    char c[20];
  public:
    test(const char *s="A") {
      a=0;   b=0;
      strcpy(c, s);
      }
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

无动态内存申请
执行正确

带缺省参数的构造函数

重新为c赋值

打印c的内容

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

用多编译器分别
运行下面两个例子

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

```
hello
A

hello

china
china
```

有动态内存申请
执行结果错(与期望不同)

注意：
1、篇幅问题，假设申请成功
2、程序不完整，仅在构造函数中动态申请，未定义析构函数进行释放

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

```
hello

A

hello

china
china
```

有动态内存申请
执行结果错(与期望不同)

同左例，加入析构函数后，
不但执行结果错，而且
VS下会有错误弹窗，
GNU下虽然没有错误弹窗，
但仍然是错误的

4.4. 含动态内存

4.4.1. 含动态内

★ 若对象数据

错误原因的图解及具体解释：

1、造成4000-4019这20个字节的内存丢失

2、t1/t2的c成员同时指向一块内存，通过t1的c修改改内存块，会导致t2的c值同时改变

3、若定义了析构函数，则main函数执行完成后系统会调用析构函数(按t2,t1的顺序)，t2调用析构函数释放3000-3019后，再调用t1的析构函数会导致重复释放3000-3019，错!!!

如何保证有动态内存时的赋值正确性?

后续模块 重载=运算符

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();                hello
    t2.display();                A
    t2=t1;
    t2.display();                hello
    t1.set("china");
    t1.display();                china
    t2.display();                china
}
```

有动态内存申请
执行结果错(与期望不同)

同左例，加入析构函数后，不但执行结果错，而且VS下会有错误弹窗，GNU下虽然没有错误弹窗，但仍然是错误的

§9. 动态内存申请

4. 含动态内存申请内存的类和对象
4.4. 含动态内存申请的对象的赋值与复制
4.4.1. 含动态内存申请的对象的赋值
4.4.2. 含动态内存申请的对象的复制

对象的复制

含义：建立一个新对象，其值与某个已有对象完全相同

形式：

类 对象名(已有对象名)
类 对象名=已有对象名

两种形式
本质一样

```
Time t1(14,15,23), t2(t1), t3=t1; //定义语句中
```

★ 与对象赋值的区别：定义语句/执行语句中

```
Time t1(14,15,23),t2, t3=t1; //复制(已有对象t1 => 新对象t3)
t2 = t1; //赋值(已有对象t1 => 已有对象t2)
```

★ 系统默认的复制操作是将已有对象的全部数据成员的值对应赋给新对象的全部数据成员(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现
(通过重定义复制/拷贝构造函数来实现，荣誉课内容)

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

含义：建立一个新对象, 其值与某个已有对象完全相同

对象复制的实现：建立新对象时自动调用复制构造函数（也称为拷贝构造函数）

复制构造函数：

类名(const 类名 &引用名)

★ 用一个对象的值去初始化另一个对象

★ 若不定义复制构造函数，则系统自动定义一个，参数为const型引用，函数体为对应成员内存拷贝

★ 若定义了复制构造函数，则系统缺省定义的消失

★ 允许体内实现或体外实现

★ 复制构造函数和普通构造函数（可能多个）的地位平等，调用其中一个后就不再调用其它构造函数

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

```
class Time {
  private:
    int hour, minute, second;  //三个私有成员
  public:
    Time(int h=0, int m=0, int s=0) { //构造,适合0-3参
        hour = h; minute = m; second = s;
        }
    void display() {  //打印
       cout << hour << ":" << minute << ":" << second << endl;
       }
};
int main()
{
   Time t0, t1(14,15,23), t2(t1), t3=t1;
   t0.display();
   t1.display();
   t2.display();
   t3.display();
}
```

```
15:0:0
14:15:23
14:15:23
14:15:23
```

问题：t2(t1)、t3=t1 匹配哪个构造函数？

现有的 Time(int h=0,int m=0, int s=0) 可用于一个整数参数，例：Time t0(15)，但类型不匹配!!!

匹配了缺省的复制构造函数，相当于：

```
Time(int h=0, int m=0, int s=0);
Time(const Time &);  //缺省为拷贝12 字节
```

这两个构造函数重载即使都是一个参数，也可以区分：

```
t2(t1)
t2(14)
```

★ 复制构造函数和普通构造函数(可能多个)的地位平等，调用其中一个后就不再调用其它构造函数

```
//本例中复制构造函数的显式定义
Time(const Time &t);
//本例中复制构造函数的体外实现
Time::Time(const Time &t)
{   hour   = t.hour;
    minute = t.minute;
    sec    = t.sec;
}
```

```
class Time {
      ...
   public:
      Time(int h=0);

      Time(int h, int m, int s=0);

      Time(const Time &t);
};
```

```
int main()
{
   Time t1;
   Time t2(10);
   Time t3(1,2,3);
   Time t4(4,5);
   Time t5(t2);
   Time t6=t4;
}
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

复制构造函数的调用时机：

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

★ (调用3)函数的返回类型是对象时

★ (不调用1)不包括形参为引用的情况(引用为实参别名)

★ (不调用2)不包括执行语句中的赋值(=)操作，执行赋值(=)操作通过赋值运算符(=)的重载来实现(后续模块)

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{   cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{  hour   = t.hour   - 1;
   minute = t.minute - 1;
   sec    = t.sec    - 1;
   cout << "复制构造" << hour << endl;
}
```

★ (不调用1)不包括形参为引用的情况(引用为实参别名，不调用)

- 用VS编译运行
- 用Dev/Linux编译运行

多编译器一致

```
void fun(Time &t)
{
    t.display();
}
int main()
{
    Time t1(14,15,23);
    fun(t1);
    return 0;
}
```

本例证明不调用1

```
普通构造14
14:15:23
析构14
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{   cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{  hour   = t.hour   - 1;
   minute = t.minute - 1;
   sec    = t.sec    - 1;
   cout << "复制构造" << hour << endl;
}
```

★ (不调用1)不包括形参为引用的情况(引用为实参别名，不调用)

★ (不调用2)不包括执行语句中的赋值(=)操作，执行赋值(=)操作通过赋值运算符(=)的重载来实现(后续内容)

- 用VS编译运行
- 用Dev/Linux编译运行

  多编译器一致

```
int main()
{
    Time t1(14,15,23), t2;
    t2 = t1;
    return 0;
}
```

本例证明不调用2

```
普通构造14
普通构造0
析构14
析构14
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{   cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{  hour   = t.hour   - 1;
   minute = t.minute - 1;
   sec    = t.sec    - 1;
   cout << "复制构造" << hour << endl;
}
```

★ (调用1)用已有对象初始化一个新建立的对象时

- 用VS编译运行
- 用Dev/Linux编译运行

多编译器一致

```
int main()
{
    Time t1(14,15,23),t2(t1);
    t2.display();
    return 0;
}
```

本例证明调用1

```
普通构造14
复制构造13
13:14:22
析构13
析构14
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{   cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{  hour   = t.hour   - 1;
   minute = t.minute - 1;
   sec    = t.sec    - 1;
   cout << "复制构造" << hour << endl;
}
```

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

- 用VS编译运行
- 用Dev/Linux编译运行

多编译器一致

```
void fun(Time t)
{
    t.display();
}
int main()
{
    Time t1(14,15,23);
    fun(t1);
    return 0;
}
```

本例证明调用2

```
普通构造14
复制构造13
13:14:22
析构13
析构14
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{   hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{   cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{  hour   = t.hour   - 1;
   minute = t.minute - 1;
   sec    = t.sec    - 1;
   cout << "复制构造" << hour << endl;
}
```

★ (调用1)用已有对象初始化一个新建立的对象时
★ (调用2)函数形参为对象，实参向形参进行单向传值时
★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

  返回自动变量时，多编译器有差异
  返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

//VS	//Dev+Linux
普通构造14 复制构造13 13:14:22 析构13 析构14	普通构造14 14:15:23 析构14

```
Time fun()
{   static Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

普通构造14
复制构造13
13:14:22
析构13
析构14

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU(Dev/Linux)的解释：

采用NRV(NRV=Named Return value)优化技术,当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称(t2直接利用t1原空间)，从而提高运行速度(少复制一次内存)

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

//VS	//Dev+Linux
普通构造14	普通构造14
复制构造13	14:15:23
13:14:22	析构14
析构13	
析构14	

VS系列

Gnu系列

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU(Dev/Linux)的解释：

采用NRV(NRV=Named Return value)优化技术,当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称(t2直接利用t1原空间)，从而提高运行速度(少复制一次内存)

VS系列

Gnu系列

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14,15,23);
    cout << "&t1=" << &t1 << endl;
    return t1;
}
int main()
{   Time t2 = fun();
    cout << "&t1=" << &t1 << endl;
    t2.display();
}
```

NRV技术讨论：（如何验证）

用不同编译器运行，观察t1和t2的地址是否相同，解释？

VS：
普通构造14
&t1=某地址
复制构造13
析构14
&t2=某地址(与t1不同)
13:14:22
析构13

VS：
7行输出，观察两个地址

GNU：
5行输出，观察两个地址

GNU：
普通构造
&t1=某地址
&t2=某地址(同t1)
14:15:23
析构14

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

★ (调用1)用已有对象初始化一个新建立的对象时
★ (调用2)函数形参为对象，实参向形参进行单向传值时
★ (调用3)函数的返回类型是对象时
- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

NRV技术讨论：

VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU(Dev/Linux)的解释：

采用NRV(NRV=Named Return value)优化技术,当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称(t2直接利用t1原空间)，从而提高运行速度(少复制一次内存)

★ "被返回值"不是自动对象时，不能进行NRV优化
(只有当"被返回值"在函数运行结束后被释放时才适用NRV优化)

```
Time fun()
{   static Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

普通构造14
复制构造13
13:14:22
析构13
析构14

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

★ 可能不满足某些特殊要求

（例：本测试样例中，希望复制构造函数将Time的三个成员hour/minute/sec各减1）

=> 合理推断：应有编译选项设置，可指定不采用NRV技术

-fno-elide-constructors

Dev：工具 - 编译选项 - 编译器卡片 - 加入命令

Linux：直接加入编译参数

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}
int main()
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

//VS	//Dev+Linux
普通构造14	普通构造14
复制构造13	14:15:23
13:14:22	析构14
析构13	
析构14	

1. 本例程序，GNU下加编译选项是7行，不加是3行，为什么？
2. 为什么VS是5行？

```
//Dev+Linux(加选项)
普通构造14
复制构造13
析构14
复制构造12
析构13
12:13:21
析构12
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

NRV的进一步讨论：

在完全无NRV的情况下，①②两处应该调用两次复制构造函数

① t1 => 无名临时对象　13:14:22

② 无名临时对象 => t2　12:13:21

输出应为：

普通构造14

复制构造13　//①处复制构造，t1=>临时对象

析构14

复制构造12　//②处复制构造，临时对象=>t2

析构13　//临时对象析构

12:13:21

析构12

结论：

1、GNU加 -fno-elide-constructors 后输出符合预期

2、VS默认设置下，在①/②的某处仍然用了NRV（如何判断哪处？）

★ (调用1)用已有对象初始化一个新建立的对象时

★ (调用2)函数形参为对象，实参向形参进行单向传值时

★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}          ①
int main()  ②
{   Time t2 = fun();
    t2.display();
}
```

本例证明调用3

```
//VS
普通构造14
复制构造13
13:14:22
析构13
析构14
```

```
//Dev+Linux
普通构造14
14:15:23
析构14
```

1. 本例程序，GNU下加编译选项是7行，不加是3行，为什么？
2. 为什么VS是5行？

```
//Dev+Linux(加选项)
普通构造14
复制构造13
析构14
复制构造12
析构13
12:13:21
析构12
```

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}       ①
int main()
{
    Time t2;
      ②
    t2 = fun();//前例(不调用1)已证明②处不调用复制构造函数
    t2.display();
}
```

```
//VS
普通构造0
普通构造14
复制构造13
析构14
析构13
13:14:22
析构13
```

★ (调用1)用已有对象初始化一个新建立的对象时
★ (调用2)函数形参为对象，实参向形参进行单向传值时
★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

  返回自动变量时，多编译器有差异

  返回静态局部变量时，多编译器一致

本例证明调用3

```
Time fun()
{   Time t1(14,15,23);
    return t1;
}       ①
int main()
{   Time t2 = fun();  ②
    t2.display();
}
```

```
//VS
普通构造14
复制构造13
13:14:22
析构13
析构14
```

综合分析两处输出，可得到结论：
VS在_①_处调用了复制构造函数(未用NRV)
而在_②_处使用了NRV技术

NRV的进一步讨论：

在完全无NRV的情况下，①②两处应该调用两次复制构造函数

① t1 => 无名临时对象　13:14:22

② 无名临时对象 => t2　12:13:21

结论：

1、GNU加 -fno-elide-constructors 后输出符合预期

2、VS默认设置下，在①/②的某处仍然用了NRV（如何判断哪处？）

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

注意：
t2调用缺省的复制构造函数，
不再调用普通构造函数
(未申请20字节空间)，
因此本例中没有内存丢失

```
//系统缺省的构造函数实现方法
test(const test &s)
{   a=s.a;
    b=s.b;
    c=s.c;
}
```

实质上是执行系统函数 memcpy：
memcpy(this, &s, sizeof(test));
相当于一次直接复制了12个字节

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
       a=0;    b=0;
       c=new char[20];
       strcpy(c, s);
       } //带缺省参数的构造函数
    ~test(){
       delete []c;
       } //析构函数
    void set(const char *s) {
       strcpy(c, s);
       }
    void display() {
       cout << c << endl;
       }
};

int main()
 {
    test t1("hello"),t2(t1);
    t1.display();      hello
    t2.display();      hello
    t1.set("china");
    t1.display();      china
    t2.display();      china
 }                     出错!!!
```

复制错误示例

main完成后，
按t2,t1的顺序调用析构函数
导致3000-3019被重复释放

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

复制正确示例

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
       a=0;    b=0;
       c=new char[20];
       strcpy(c, s);
       } //带缺省参数的构造函数
    test(const test &t);  //复制构造函数的声明
    ~test(){    delete []c;    } //析构函数
    void set(const char *s) {    strcpy(c, s);    }
    void display() {    cout << c << endl;    }
};
test::test(const test &s)  //复制构造的体外实现
{   a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
}
int main()
 {
    test t1("hello"),t2(t1);
    t1.display();      hello
    t2.display();      hello
    t1.set("china");
    t1.display();      china
    t2.display();      hello
}
```

仔细体会两者实现的区别

```
//系统缺省的构造函数实现方法
test(const test &s)
{   a=s.a;
    b=s.b;
    c=s.c;
}
```

实质上是执行系统函数 memcpy：
memcpy(this, &s, sizeof(test));
相当于一次直接复制了12个字节

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
       a=0;    b=0;
       c=new char[20];
       strcpy(c, s);
       } //带缺省参数的构造函数
    test(const test &t);  //复制构造函数的声明
    ~test(){    delete []c;    } //析构函数
    void set(const char *s) {    strcpy(c, s);    }
    void display() {    cout << c << endl;    }
};
test::test(const test &s)  //复制构造的体外实现
{   a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
}
int main()
 {  test t1("hello"),t2(t1);
    t1.display();      hello
    t2.display();      hello
    t1.set("china");
    t1.display();      china
    t2.display();      hello
    t2 = t1;
}
```

复制正确
仍无法解决
赋值错误示例

注意：虽然解决了定义时赋初值
问题(test t2(t1) / t2=t1;)，
但仍无法解决赋值问题(t2 = t1;)

赋值仍然会错，具体要用
后续模块的=运算符重载来解决

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.2. 含动态内存申请的对象的复制

★ NRV技术的进一步讨论：

在GNU系列（Dev/Linux）中，如果含有二次申请，且自定义了复制构造函数，NRV是否仍有效？

VS系列

Gnu系列：有效！

VS：
普通构造1
复制构造2
析构1
Hello
析构2

GNU：
普通构造1
Hello
析构1

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
       a=0;  b=0;
       c=new char[20];
       strcpy(c, s);
       cout << "普通构造" << (void *)c << endl;
       }
    test(const test &s);
    ~test() {
         cout << "析构" << (void *)c << endl;
         delete []c;
       }
    void set(const char *s) {    strcpy(c, s);    }
    void display() {     cout << c << endl;    }
};
test::test(const test &s)
{   a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
    cout << "复制构造" << (void *)c << endl;
}
test fun()
{  test t1("Hello");
   return t1;
}
int main()
{  test t2 = fun();
   t2.display();
}
```

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.5. 浅拷贝与深拷贝

★ 浅拷贝(Shallow Copy)：只复制对象的指针，而不是复制对象自身；
新旧对象共享内存；
修改其中一个的值则另一个会随之改变；
两个对象是联动的

★ 深拷贝(Deep Copy)　：复制对象自身；
新旧对象分别占用不同内存；
修改其中一个的值不会影响另一个；
两个对象是完全独立的

注：1、网上浅拷贝/深拷贝的资料很多，可读性/易懂程度/示例语言等各不相同，但归根结底，基本的原理就是本节的内存分析
2、某些无动态内存申请的语言，其内部实现仍然是封装了动态内存申请，只是不需要用户掌握而已
3、部分封装类，可能不同函数分别浅/深拷贝
（例：对象.assign() - 浅 / 对象.copy() - 深）
4、浅/深拷贝具体采用哪种，根据实际需求决定

§10. 输入输出流

1. C++的输入与输出

1.1. 有关输入输出基本概念的回顾

§3. 结构化程序设计基础

3.4. C++的输入与输出

3.4.1. 流的基本概念

流的含义：流是来自设备或传给设备的一个数据流，由一系列字节组成，按顺序排列（字节流）

★ C/C++的原生标准中没有定义输入/输出的基本语句

★ C语言用printf/scanf等函数来实现输入和输出，通过#include <stdio.h>来调用

★ C++通过cin和cout的流对象来实现，通过#include <iostream>来调用

cout：输出流对象　　<<：流插入运算符

cin：输入流对象　　>>：流提取运算符

§10. 输入输出流

1.C++的输入与输出

1.2.流和缓冲区

★ 流：C++将输入和输出看作字节流。输入时，程序从输入流中抽取字节；输出时，程序将字节插入输出流。

★ 缓冲区：用作中介的内存块，将信息从设备传输到程序或从程序传输到设备的临时存储工具，可以提高数据的读取速率。

★ 刷新缓冲区：输出时，程序首先填满缓冲区，然后把整块的数据传输给硬盘，并清空缓冲区，这被称为刷新缓冲区。

★ 为了管理流，C++提供了一系列头文件：

streambuf类为缓冲区提供了内存，并提供了填充缓冲区、访问缓冲区、刷新缓冲区和管理缓冲区内存的类方法；

ios_base类表示流的一般特征，如是否可读取，是二进制流还是文本流等；

ios类基于ios_base；

ostream类提供了输出方法；

istream类提供了输入方法；

iostream提供了输入和输出方法。

★ C++为了能够处理需要16位国际字符集或更宽的字符类型

传统的8位char(窄)类型　新添加wchar_t(宽)类型

有专门的输入输出对象用于处理宽字符，如wcin/wcout

★ iostream头文件会自动生成八个流对象（四个是宽类型的）：

cin标准输入流；

cout标准输出流；

cerr标准错误流；

clog标准错误流。

§10. 输入输出流

1. C++的输入与输出

1.3. 输入输出的基本概念

输入输出的种类：

系统设备：标准输入设备：键盘

(标准I/O) 标准输出设备：显示器

其它设备：鼠标、打印机、扫描仪等

外存文件：从文件中得到输入

(文件I/O) 输出到文件中

内存空间：输入/输出到一个字符数组/string中

(串I/O)

★ 操作系统将所有系统设备都统一当作文件进行处理

C++输入/输出的特点：

★ 与C兼容，支持printf/scanf

★ 对数据类型进行严格的检查，是类型安全的I/O操作

★ 具有良好的可扩展性，可通过重载操作符的方式输入/输出自定义数据类型

C++的输入/输出流：

★ 采用字符流方式，缓冲区满或遇到endl才输入/输出

★ cin/cout不是C++的语句，也不是函数，是类的对象，>>和<<的本质是左移和右移运算符，被重载为输入和输出运算符

§10. 输入输出流

2. 标准输出流

2.1. cout，cerr和clog流

cout：向控制台进行输出，缺省是显示器

cerr：向标准出错设备进行输出，缺省是显示器(直接输出，不必等待缓冲区满或回车)

clog：向标准出错设备进行输出，缺省是显示器(放在缓冲区中，等待缓冲区满或回车才输出)

★ 三者的使用方法一样

★ 缺省都是显示器，可根据需要进行输出重定向

2.2. 格式输出

需要掌握的基本格式：

不同数制：dec、hex、oct

设置宽度：setw

左右对齐：setiosflags(ios::left/right)

其余当作手册来查：

setfill、setprecision等

★ 输出格式可用控制符控制，也可以流成员函数形式

```
cout << setw(20)          ⇔ cout.width(20)
cout << setprecision(9)   ⇔ cout.precision(9)
...
```

§10. 输入输出流

2. 标准输出流

2.3. 流成员函数put

形式：cout.put(字符常量/字符变量)

★ 功能与putchar相同，输出一个字符

```
char a='A';
cout.put(a);        //变量
cout.put('A') ;     //常量
cout.put('\x41'); //十六进制转义符
cout.put('\101'); //八进制转义符
cout.put(65) ;      //整数当作ASCII码
cout.put(0x41) ;    //整数当作ASCII码(十六)
cout.put(0101) ;    //整数当作ASCII码(八)
```

★ 允许连续调用

```
#include <iostream>
using namespace std;

int main()
{
    cout.put(72).put(0x65).put('l').put(0154).put('a'+14);
    return 0;
}
```

Hello

§10. 输入输出流

2. 标准输出流

2.4. 与cout有关的成员函数的基本使用

2.4.1. 用于字符输出的流成员函数：

cout.put(字符常量/字符变量)	功能：向标准输出设备输出一个字符
cout.write(字符串常量/变量，输出长度)	功能：向标准输出设备输出n个字符(不会判断字符串的结尾)

2.4.2. 用于字符输出控制的流成员函数：

cout.setf(控制标记)	功能：设置指定的控制标记
cout.unsetf(控制标记)	功能：清除指定的控制标记
cout.width(宽度)	功能：设置指定的输出宽度
cout.fill(字符常量/字符变量)	功能：设置填充字节
cout.precision(精度)	功能：设置浮点数的输出精度

2.4.3. 常用控制标记及其作用：

ios::fixed	设置浮点数以固定的小数位数显示
ios::scientific	设置浮点数以科学计数法(即指数形式)显示
ios::left	输出数据左对齐
ios::right	输出数据右对齐
ios::skipws	忽略前导的空格(适用于cin，不适用于cout)
ios::uppercase	在以科学计数法输出E和十六进制输出字母X时以大写表示
ios::showpos	输出正数时，给出"+"号

§10. 输入输出流

3. 标准输入流

3.1. cin流

★ cin提取数据后，会根据数据类型是否符合要求而返回逻辑值

```
#include <iostream>
using namespace std;
int main()
{
    int a=-9;
    cin >> a;
    cout << a << " " << (cin ? 1 : 0) << endl;
    return 0;
} //不同编译器，cin为0时，a值可能不同
```

输入	cout的结果	
10	10	1
ab	0	0
12ab	12	1
很大的数字	2147483647	0

- 当cin返回为1/true时，读入的值才可信
  =>正确的处理逻辑：cin读入后，先判断cin，为1再取值
- 不同编译器，cin为0时，a的值可能不同(不可信)

★ 允许进行输入重定向

★ 循环一直执行，直到输入非数字格式

```
double a;
while(cin >> a);
```

§10. 输入输出流

3. 标准输入流

3.1. cin流

★ 流状态-设置状态

clear()方式将状态设置为它的参数：

```
clear();            //默认参数为0，清除全部三个状态位(eofbit,badbit,failbit)
clear(eofbit);      //将状态设置为eofbit，另外两个状态位清除
```

setstate()方法只影响其参数中已设置的位，而不会影响其他位：

```
setstate(eofbit);  //只设置eofbit，不影响其他位
```

★ 流状态-I/O异常

exceptions()方法用来控制异常如何被处理；

exceptions()方法返回一个位字段它包含3位分别对应于eofbit、failbit和badbit，指出哪些标记导致异常被引发；

修改流状态后clear()方法将当前的流状态与exceptions()返回的值进行比较。如果在返回值中某一位被设置而当前状态中的对应位也被设置则clear()将引发ios_base::failure异常，该异常包含what()方法；

如果exceptions()返回goodbit，则不会引发任何异常；

exceptions()的默认设置为goodbit，不会引发异常；

重载的exceptions(iostate)函数可以控制其行为；

```
cin.exceptions(badbit);
```

位运算符|使得能够指定多位。

```
cin.exceptions(badbit|eofbit);
```

§10. 输入输出流

3. 标准输入流

3.2. 文件结束符与文件结束标记

文件结束符：表示文件结束的特殊标记

★ 设备也当作文件处理

★ 一般用CTRL+Z表示键盘输入文件结束符

文件结束标记：判断文件是否结束的标记，用宏定义EOF来表示

★ 不同系统EOF的值可能不同，不必关心

★ 一般用于字符流输入的判断，对其它类型一般不用

3.3. 用于字符输入的流成员函数

★ cin.get()	功能：从输入流中读一个字符并返回该字符
★ cin.get(字符变量)	功能：从输入流中读一个字符给字符变量，返回cin(流对象自身)
★ cin.get(字符数组，字符个数n，中止字符)	功能：从输入流中读n-1个字符，若遇中止字符则结束返回cin(流对象自身)
★ cin.getline(字符数组，字符个数n，中止字符)	功能：同三个参数的cin.get()
★ cin.eof()	功能：判断是否遇到了文件结束符EOF，返回逻辑值(遇到EOF为真)
★ cin.peek()	功能：返回输入流中的下一个字符(不提取)(遇见文件结束符则返回EOF)
★ cin.putback(字符变量/字符常量)	功能：将字符变量/常量插入到输入流的头部
★ cin.ignore(字符个数n，中止字符)	功能：跳过n个字符，或遇到中止字符时提前结束

§10. 输入输出流

3. 标准输入流

3.4. 单字符输入

(1)成员函数get(char&)

(2)成员函数get(void)

特征	cin.get(ch)	ch=cin.get()
传输输入字符的方法	赋给参数ch	将函数返回值赋给ch
字符输入时函数的返回值	指向istream对象的引用	字符编码（int值）
达到文件尾时函数的返回值	转换为false	EOF

3.5. 字符串输入

(1)cin.get(...)

读取一行输入，直到到达换行符，将换行符保留在输入序列

(2)cin.getline(...)

读取一行输入，直到到达换行符，并丢弃换行符

```
istream & get(char*, int, char);      //第三个参数是分界符，遇到分界符输入停止，并保留在输入序列
istream & get(char*, int);
istream & getline(char*, int, char); //第三个参数是分界符，遇到分界符输入停止，并丢弃分界符
istream & getline(char*, int);
```

方法	行为
getline(char *, int)	如果没有读取任何字符(换行符视为读取一个字符)，则设置failbit 如果读取了最大数目的字符，且行中还有其他字符，则设置failbit
get(char *, int)	如果没有读取任何字符，则设置failbit

§10. 输入输出流

4. 文件操作与文件流

4.1. 文件的基本概念

文件及文件名：

文件：存储在外存储器上的数据的集合

文件名：操作系统用于访问文件的依据

文件的分类：

★ 按设备分：

输入文件：键盘等输入设备

输出文件：显示器、打印机等输出设备

磁盘文件：存放在磁盘(光盘、U盘)上的文件

★ 按文件的类型分：

程序文件：执行程序所对应的文件(.exe/.dll等)

数据文件：存放对应数据的文件(.cpp/.doc等)

★ 按数据的组织形式：

ASCII码文件(文本文件)：按数据的ASCII代码形式存放的文件

二进制文件：按数据的内存存放形式存放的文件

§10. 输入输出流

4. 文件操作与文件流

4.1. 文件的基本概念

文件的分类：

★ 按数据的组织形式

ASCII码文件(文本文件)：按数据的ASCII代码形式存放的文件

二进制文件：按数据的内存存放形式存放的文件

例：int型整数100000：ASCII文件为6个字节

二进制文件为4个字节

\x31	\x30	\x30	\x30	\x30	\x30

\x00	\x01	\x86	\xA0

100000=0x186A0

双精度数123.45：ASCII文件为6个字节

二进制文件为8个字节

\x31	\x32	\x33	\x2E	\x34	\x35

d	o	u	b	l	e	格	式

IEEE754

字符串"China"：ASCII文件为6个字节

\x43	\x68	\x69	\x6E	\x61	\x0

\x43	\x68	\x69	\x6E	\x61	\x0

§10. 输入输出流

4. 文件操作与文件流

4.1. 文件的基本概念

C++对文件的访问：

低级I/O：字符流方式输入/输出

高级I/O：转换为数据指定形式的输入/输出

4.2. 文件流类及文件流对象

与磁盘文件有关的流类：

输入　　：ifstream类，从istream类派生而来

输出　　：ofstream类，从ostream类派生而来

输入/输出 ：fstream类，从iostream类派生而来

流对象的建立：

ifstream 流对象名 ：用于输入文件的操作

ofstream 流对象名 ：用于输出文件的操作

fstream　流对象名 ：用于输入/输出文件的操作

§10. 输入输出流

4. 文件操作与文件流

4.3. 文件的打开与关闭

文件的打开：

文件流对象名.open(文件名，打开方式)；

★ 加#include <fstream>

★ 有多种打开方式

★ 各个打开方式可用"位或运算符|"进行组合

ios::nocreate
ios::noreplace

DevC++/Linux不支持

在VS2022下是

ios::_Nocreate
ios::_Noreplace

方　式	作　用
ios :: in	以输入方式打开文件
ios :: out	以输出方式打开文件(这是默认方式),如果已有此名字的文件,则将其原有内容全部清除
ios :: app	以输出方式打开文件,写入的数据添加在文件末尾
ios :: ate	打开一个已有的文件,文件指针指向文件末尾
ios :: trunc	打开一个文件,如果文件已存在,则删除其中全部数据;如文件不存在,则建立新文件。如已指定了 ios :: out 方式,而未指定 ios :: app,ios :: ate,ios :: in,则同时默认此方式
ios :: binary	以二进制方式打开一个文件,如不指定此方式则默认为 ASCII 方式
ios :: nocreate	打开一个已有的文件,如文件不存在,则打开失败。nocreat 的意思是不建立新文件
ios :: noreplace	如果文件不存在则建立新文件,如果文件已存在则操作失败,noreplace 的意思是不更新原有文件
ios :: in | ios :: out	以输入和输出方式打开文件,文件可读可写
ios :: out | ios :: binary	以二进制方式打开一个输出文件
ios :: in | ios :: binary	以二进制方式打开一个输入文件

§10. 输入输出流

4. 文件操作与文件流

4.3. 文件的打开与关闭

文件的打开：

文件流对象名.open(文件名，打开方式);

★ 文件名允许带全路径，若不带路径，则表示与可执行文件同目录

```
ofstream out;
out.open("aa.dat", ios::out);
out.open("..\\C++\\aa.dat", ios::out);
out.open(".\\C++\\aa.dat", ios::out);
out.open("\\C++\\aa.dat", ios::out | ios::app);
out.open("c:\\C++\\aa.dat", ios::out);
```

1、路径有绝对路径和相对路径两种
2、..表示父目录，.表示当前目录

● VS2022等编译器，如在集成环境内运行，则当前目录是指源程序文件(*.cpp)所在的目录，如果离开集成环境(例如用cmd命令行运行)，则当前目录是指可执行文件(*.exe)所在目录

★ 用"\\"表示目录间分隔符的方式在Linux下无效，用"/"方式则在Windows/Linux下均有效

```
out.open("aa.dat", ios::out);
out.open("../C++/aa.dat", ios::out);
out.open("./C++/aa.dat", ios::out);
out.open("/C++/aa.dat", ios::out | ios::app);
out.open("c:/C++/aa.dat", ios::out);
```

§10. 输入输出流

4. 文件操作与文件流

4.3. 文件的打开与关闭

文件的打开：

文件流对象名.open(文件名，打开方式);

★ 可在声明文件流对象时直接打开

```
ofstream out("aa.dat", ios::out);
```

★ 打开方式与文件流对象之间要兼容,否则无意义

```
ifstream in;
in.open("aa.dat", ios::out);  //in对象用out打开，无意义，缺省仍为ios::in
```

★ 每个文件被打开后，都有一个文件指针，初始指向开始/末尾的位置(根据打开方式决定)

★ 执行open操作后，要判断文件是否打开成功

`if (!outfile)`
`if (outfile.is_open()==0)`
`if (!outfile.is_open())`
	两种方法均可以
	1. 判断流对象自身
	2. is_open函数

`if (outfile.open("f1.dat",ios::app)==0)` //open返回void `if (!outfile.open("f1.dat",ios::app))` `if (outfile==NULL)` //outfile不是指针	错

文件的关闭：

文件流对象名.close();

§10. 输入输出流

4. 文件操作与文件流

4.3. 文件的打开与关闭

文件的打开：

文件流对象名.open(文件名，打开方式)；

文件的关闭：

文件流对象名.close()；

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{   ifstream in;
    in.open("bb.dat", ios::in);
    if (!in.is_open()==0)
        cout << "open failed." << endl;
    else
        cout << "open success." << endl;
    in.close();

    return 0;
}
```

不存在时：失败
存在时：成功

若换成ios::out，则打开不受影响，但后续读写会有问题

ifstream

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{   ofstream out;
    out.open("aa.dat", ios::out);
    if (out.is_open()==0)
        cout << "open failed." << endl;
    else
        cout << "open success." << endl;
    out.close();

    return 0;
}
```

不存在时：成功(创建)
存在时：成功(覆盖)
存在并只读：失败

ofstream

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{   ofstream out;
    out.open("bb.dat", ios::out | ios::_Nocreate);
    if (out.is_open()==0)
        cout << "open failed." << endl;
    else
        cout << "open success." << endl;
    out.close();
}
```

不存在时：失败
存在时：成功(覆盖)
存在并只读：失败

ofstream

§10. 输入输出流

4. 文件操作与文件流

4.4. 对ASCII文件的操作

基本方法：将文件流对象名当作cin/cout对象，用>>和<<进行格式化的输入和输出，同时前面介绍的关于cin/cout的get/getline/put/eof/peek/putback/ignore等成员函数也可以被文件流对象所使用

★ >>和<<使用时的注意事项与cin、cout时相同

cin >> 变量 => infile >> 变量

cout << 变量 => outfile << 变量

★ 成员函数的使用方法与前面相同

cout.put('A') => outfile.put('A')

★ 流对象与打开方式、流插入/流提取运算符之间要求匹配

● 错误的例子：ifstream打开的写文件用流插入运算符

```
//例：打开d:\test\data.txt文件,并写入"Hello"
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream out;
    char ch;
    out.open("d:/test/data.txt", ios::out); //ios::out无效
    if (out.is_open()==0) {
        cout << "文件打开失败" << endl;
        return -1;
        }
    out << "Hello" << endl; //编译错
    out.close();
    return 0;
}
```

error C2676: 二进制“<<”：“std::ifstream”不定义该运算符或到预定义运算符可接收的类型的转换

§10. 输入输出流

4. 文件操作与文件流

4.4. 对ASCII文件的操作

```
//例：键盘输入10个int，输出到文件中
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   int a[10];
   ofstream outfile("f1.dat", ios::out);
   if (!outfile.is_open()) {
      cerr << "open error!" << endl;
      exit(1);  //结束程序运行，向操作系统返回1
      }
   cout << "enter 10 integer numbers:" << endl;
   for(int i=0; i<10; i++) {
      cin >> a[i];              //键盘输入
      outfile << a[i] << " ";   //int型输出到文件
      }
   outfile.close();
   return 0;
}
```

运行两次，观察结果

```
//例：键盘输入10个int，输出到文件中（变化，加ios::app）
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   int a[10];
   ofstream outfile("f1.dat", ios::out | ios::app);
   if (!outfile.is_open()) {
      cerr << "open error!" << endl;
      exit(1);  //结束程序运行，向操作系统返回1
      }
   cout << "enter 10 integer numbers:" << endl;
   for(int i=0; i<10; i++) {
      cin >> a[i];              //键盘输入
      outfile << a[i] << " ";   //int型输出到文件
      }
   outfile.close();
   return 0;
}
```

运行两次，观察结果

§10. 输入输出流

4. 文件操作与文件流

4.4. 对ASCII文件的操作

```
//例：从文件中读入10个int，输出到屏幕上
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    int a[10];
    ifstream infile("f1.dat", ios::in);
    if (!infile.is_open()) {
        cerr << "open error!" << endl;
        exit(1);  //结束程序运行，向操作系统返回1
        }
    for(int i=0; i<10; i++) {
        infile >> a[i];        //从文件中读10个int放入a数组
        cout << a[i] << " ";   //int型输出到屏幕
        }

   infile.close();
   return 0;
}
```

利用上例生成的f1.dat
自己编辑完全正确的f1.dat
自己编辑含错误的f1.dat

§10. 输入输出流

4. 文件操作与文件流

4.4. 对ASCII文件的操作

```
//例：打开d:\test\data.txt文件,并将内容输出到屏幕上
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream in;
    char ch;
            "d:/test/data.txt"
    in.open("d:\\test\\data.txt", ios::in);//双斜杠
    if (in.is_open()==0) {          //!in.is_open()
        cout << "文件打开失败\n";   //cerr
        return -1;                  //exit(1)
        }

    while(!in.eof()) {
        ch = in.get();     //in.get(ch);
        putchar(ch);       //cout.put(ch);
        }

    in.close();
    return 0;
}
```

```
while((ch=in.get())!=EOF)
    cout.put(ch);

EOF是系统定义的文件结束标记
```

```
//例：将 d:\test\data.txt 文件复制为 d:\demo\data2.txt
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream in;
    ofstream out;
    char ch;
            "d:/test/data.txt"
    in.open("d:\\test\\data.txt", ios::in);
    if (!in.is_open()) {
        cout << "无法打开源文件" << endl;
        return -1;
        }
    out.open("d:\\demo\\data2.txt", ios::out);
    if (!out.is_open()) {
        cout << "无法打开目标文件" << endl;
        in.close();
        return -1;
        }

    while(in.get(ch))
        out.put(ch);

    in.close();
    out.close();
    return 0;
}
```

```
while(!in.eof()) {
    ch = in.get();
    out.put(ch);
    }
```

问：1、左右两种复制方式，哪种方式复制后的字节大小与原文件不同？为什么？
2、在保持读源文件的函数不变的情况下如何改正？
3、左侧的输出到屏幕有同样问题吗？

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 用ASCII文件的字符方式进行操作

- 仅能按字节读写
- 如果文件中有0x1A则无法继续读取(文本文件不可能有此字符)

★ 用read/write进行操作

文件流对象名.read(内存空间首指针，长度);

从文件中读长度个字节，放入从首指针开始的空间中

文件流对象名.write(内存空间首指针，长度);

将从首指针开始的连续长度个字节写入文件中

- read/write均为纯字节，无尾零等任何附加信息
- read/write一般仅用于二进制读写，如果用于十进制读写，则仅受长度的限制，不考虑格式化(是否有尾零/数据是否合理等，相当于二进制)

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 用read/write进行操作

```
//例：将内存以ASCII方式写入文件中（二进制转十进制）
#include <iostream>
#include <fstream>
#include <cstdlib>   //exit
using namespace std;

struct student {
   char name[20];
   int  num;
   int  age;
   char sex;
}; //含填充共32字节
int main()
{
    student stud[3]={"Li",1001,18,'f', "Fan",1002,19,'m',"Wang",1004,17,'f'}; //每个数组的存储为机内二进制形式
    ofstream outfile("stud.dat", ios::out);
    if (!outfile.is_open()) {
        cerr << "open error!" << endl;
        exit(-1);  //强制结束程序
        }

    for(int i=0; i<3; i++)
        outfile << stud[i].name << stud[i].num  << stud[i].age << stud[i].sex << endl; //通过 << 转换为十进制形式

    outfile.close();
    return 0;
}
```

以ASCII文件方式写入

生成的stud.dat文件共36字节：

Li100118f
Fan100219m
Wang100417f

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 用read/write进行操作

```
//用read方式读ASCII文件（stud.dat由上例生成）
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream infile("stud.dat",ios::in);
    if (!infile.is_open()) {
       cerr << "open error!" << endl;
       exit(-1);
       }
    char name[20];
    infile.read(name, 20);
    cout << '*' << name << '*' << endl;
    infile.close();
}
```

1、观察read后name的输出
2、将read后的20改为2、200，再观察
3、如果read的长度超过文件长度，如何处理？
4、改成ios::in|ios::binary，为什么少读了一个字符？

```
//用流对象方式读ASCII文件（stud.dat由上例生成）
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream infile("stud.dat", ios::in);
    if (!infile.is_open()) {
       cerr << "open error!" << endl;
       exit(-1);
       }
    char name[20];
    infile >> name;
    cout << '*' << name << '*' << endl;
    infile.close();
}
```

1、观察read后name的输出
2、用编辑软件将stud.dat文件的第3个字符改为空格，再观察

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 用read/write进行操作

```
//例：将内存以原始二进制方式写入文件中
#include <iostream>
#include <fstream>
using namespace std;

struct student {
   char name[20];
   int  num;
   int  age;
   char sex;
}; //含填充共32字节

int main()
{
    student stud[3]={"Li",1001,18,'f', "Fun",1002,19,'m',"Wang",1004,17,'f'}; //每个数组的存储为机内二进制形式
    ofstream outfile("stud.dat", ios::binary);
    if (!outfile.is_open()) {
        cerr << "open error!" << endl;
        exit(-1);  //强制结束程序
        }

    for(int i=0; i<3; i++) //一次写一个数组元素（32字节）
        outfile.write((char *)&stud[i], sizeof(stud[i]));

    outfile.close();
    return 0;
}
```

以二进制文件方式写入

stud.dat文件共96字节，前32字节为：

```
4C 69 00 ?? ?? ?? ?? ??
?? ?? ?? ?? ?? ?? ?? ??
?? ?? ?? ?? E9 03 00 00
12 00 00 00 66 ?? ?? ??
```

```
4C6900     => "Li"(含尾零,多余17个)
E9030000   => 0x000003E9 => 1001
12000000   => 0x00000012 => 18
66         => 'f'(后3个是填充字节)
```

```
outfile.write((char *)stud, sizeof(stud));//整个数组96字节)
```

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 用read/write进行操作

以二进制文件方式读取

```
//用read方式读二进制文件（stud.dat由上例生成）
#include <iostream>  //书上缺，要补上
#include <fstream>
using namespace std;
struct student {
   char name[20];  //不能是string name，必须是char name[20]，[19]或[21]都不行，必须要保证与文件的32字节一致
   int  num;
   int  age;
   char sex;
}; //含填充共32字节
int main()
{   student stud[3];
    int i;
    ifstream infile("stud.dat", ios::binary); //stud.dat的内容是由上例生成的二进制文件
    if (!infile.is_open()) {
         cerr << "open error!" << endl;
         exit(-1);  //退出
         }
    for(i=0; i<3; i++)  //一次读入一个数组元素（32字节）
         infile.read((char *)&stud[i], sizeof(stud[i]));
    infile.close();
    for(i=0; i<3; i++) {
         cout << "No." << i+1 << endl;
         cout << "name:" << stud[i].name << endl;
         cout << "num:" << stud[i].num << endl;
         cout << "age:" << stud[i].age << endl;
         cout << "sex:" << stud[i].sex << endl << endl; //多空一行
         }
    return 0;
}
```

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 与文件指针有关的流成员函数

适用于输入文件的：

gcount()　　：返回最后一次读入的字节

tellg()　　：返回输入文件的当前指针

seekg(位移量，位移方式)：移动输入文件指针

适用于输出文件的：

tellp()　　：返回输出文件的当前指针

seekp(位移量，位移方式)：移动输出文件指针

位移方式：

ios::beg：从文件头部移动，位移量必须为正

ios::cur：从当前指针处移动，位移量可正可负

ios::end：从文件尾部移动，位移量必须为负

★ 随机访问二进制数据文件

在文件的读写过程中，可前后移动文件指针，达到按需读写的目的

- ifstream无tellp，ofstream无tellg
- fstream的tellg/tellp是同步移动的

§10. 输入输出流

4. 文件操作与文件流

4.5. 对二进制文件的操作

★ 与文件指针有关的流成员函数

★ 随机访问二进制数据文件

★ 关于二进制访问的几个注意事项

- read/write虽然是内存首地址，实际编程中用字符数组，但注意不是字符串，不处理\0
- read参数中的长度是最大读取长度，不是实际读取长度，因此read后要用gcount()返回真实读到的字节数
- 如果读写方式打开(ios::in | ios::out)，则只有一个文件指针，seekg()和seekp()是同步的，tellg()和tellp()也是同步的
- 在文件的操作超出正常范围后(例：read()已到EOF、seekg()/seekp()超文件首尾范围等)，再次对文件进行seekg()/seekp()/tellg()/tellp()等操作都可能会返回与期望不同的值，建议在文件操作过程中多用good()/fail()/eof()/clear()等函数

§10. 输入输出流

5. C++的字符串流(sstream)

5.1. 基本概念

以内存中的string类型变量为输入/输出对象

★ 可以存放各种类型的数据

★ 与标准输入输出流相同，进行文本和二进制之间的相互转换

向string存数据⇔cout：二进制 => ASCII

从string取数据⇔cin ：ASCII => 二进制

● 推论：可用于不同数据类型的转换

★ 不是文件，不需要打开和关闭

5.2. 相关流对象的建立

字符串输出流对象：

ostringstream 对象名

字符串输入流对象：

istringstream 对象名

字符串输入/输出流对象：

stringstream 对象名

★ 加 #include <sstream>

§10. 输入输出流

5.3. 字符串输出流对象的使用

例1：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    ostringstream out;
    out << "Hello" << 10 << 11.2 << endl;

    string s1 = out.str();
    cout << s1 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello1011.2
```

例2：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    ostringstream out;
    out << "Hello" << 10 << 11.2 << endl;


    cout << out.str() << endl;//等价例1

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello1011.2
```

★ 成员函数str()的作用：将ostringstream的内容转换为string格式

★ ostringstream最简单的用法：将多个格式化内容拼在一起，集中输出

- ostringstream类的成员函数str()：返回一个被初始化为缓冲区内容的字符串对象

§10. 输入输出流

5.4. 字符串输入流对象的使用

例1：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 10 11.2");
    cout << in.str() << endl;

    char  s[10];
    short i;
    float f;
    in >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;

    cout << in.good() << endl;
    cout << in.str() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello 10 11.2
Hello-10-11.2
0
Hello 10 11.2
```

例2：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 10 11.2 xyz");
    cout << in.str() << endl;

    char  s[10];
    short i;
    float f;
    in >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;

    cout << in.good() << endl;
    cout << in.str() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello 10 11.2 xyz
Hello-10-11.2
1
Hello 10 11.2 xyz
```

★ 可用str()打印现有内容
★ 读完后，内容仍在
★ 如果现有内容全部读完，goodbit会置0

读取string对象中的格式化信息或将格式化信息写入string对象中被称为内核格式化(incore formatting)

§10. 输入输出流

5.4. 字符串输入流对象的使用

例3：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 10 11.2");
    cout << in.str() << endl;

    char  s1[10], s2[10]="xyz";
    short i1,     i2=123;
    float f1,     f2=0.456F;
    in >> s1 >> i1 >> f1;
    cout << s1 << '-' << i1 << '-' << f1 << endl;
    cout << s2 << '-' << i2 << '-' << f2 << endl;

    in.clear();
    in.seekg(0, ios::beg);
    in >> s2 >> i2 >> f2;
    cout << s2 << '-' << i2 << '-' << f2 << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello 10 11.2
Hello-10-11.2
xyz-123-0.456
Hello-10-11.2
```

★ istringstream的内容可重复读取

§ 10. 输入输出流

5.4. 字符串输入流对象的使用

例4：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 70000 11.2");

    char  s[10];
    short i;
    float f;

    in >> s;

    in >> i;


    in >> f;
    cout << s << '-' << i << '-' << f << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello-32767--1.07374e+08
```

例5：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 70000 11.2");

    char  s[10];
    short i;
    float f;

    in >> s;
    cout << in.good() << endl;
    in >> i;
    cout << in.good() << endl;

    in >> f;
    cout << s << '-' << i << '-' << f << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1
0
Hello-32767--1.07374e+08
```

例6：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 70000 11.2");

    char  s[10];
    short i;
    float f;

    in >> s;
    cout << in.good() << endl;
    in >> i;
    cout << in.good() << endl;
    in.clear();
    in >> f;
    cout << s << '-' << i << '-' << f << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1
0
Hello-32767-11.2
```

★ 如果数据超范围，后续会错，可clear()恢复

§ 10. 输入输出流

5.4. 字符串输入流对象的使用

例7：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 10 11.2");

    char  s[10];
    short i;
    float f;
    in >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;

    cout << in.good() << endl;
    in.str("tongji 123 0.123");

    in >> s >> i >> f;
    cout << s << '=' << i << '=' << f << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello-10-11.2
0
=10=11.2
```

例8：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("Hello 10 11.2 12345");

    char  s[10];
    short i;
    float f;
    in >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;

    cout << in.good() << endl;
    in.str("tongji 123 0.123");

    in >> s >> i >> f;
    cout << s << '=' << i << '=' << f << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello-10-11.2
1
tongji=123=0.123
```

★ 可用带参str()再次赋新内容，但注意goodbit

"=10=11.2"是因为读s出错，s置空，i/f未变

§ 10. 输入输出流

5.5. 字符串输入/输出流对象的使用

例1：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
    stringstream ss("Hello 10 11.2");
    char  s[10];
    short i;
    float f;
    ss >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;
    cout << ss.tellg() << endl;

    ss << "xyz 123 0.456";
    cout << ss.str() << endl;
    cout << ss.tellg() << endl;



    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello-10-11.2
-1
Hello 10 11.2
-1
```

例2：观察cout的输出

```
#include <iostream>
#include <sstream>
using namespace std;
int main()
{
    stringstream ss("Hello 10 11.2");
    char  s[10];
    short i;
    float f;
    ss >> s >> i >> f;
    cout << s << '-' << i << '-' << f << endl;
    cout << ss.tellg() << endl;
    ss.clear();
    ss << "xyz 123 0.456";
    cout << ss.str() << endl;
    cout << ss.tellg() << endl;
    ss.seekg(0, ios::beg);
    ss >> s >> i >> f;
    cout << s << '=' << i << '=' << f << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello-10-11.2
-1
xyz 123 0.456
13
xyz=123=0.456
```

★ stringstream可读可写，但注意goodbit

§10. 输入输出流

5.5. 字符串输入/输出流对象的使用

例3：综合应用

```
//先从流对象中输入数据，再把排序后的结果输出到流对象中
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    stringstream ss("12 34 65 -23 -32 33 61 99 321 32");
    int a[10], i, j, t;

    for (i = 0; i < 10; i++)
        ss >> a[i];  //ss中的内容逐个读入int a[10]中

    cout << "array a:";
    for (i = 0; i < 10; i++)  //输出int a[10]的内容
        cout << a[i] << " ";
    cout << endl;

    //进行排序
    for (i = 0; i < 9; i++)
        for (j = 0; j < 9 - i; j++)
            if (a[j] > a[j + 1]) {
                t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }

    //输出到ss中（ss刚才用做了输入流，现在覆盖其中的内容）
    ss.clear();
    ss.seekg(0, ios::beg);
    for (i = 0; i < 10; i++)
        ss << a[i] << " ";
    ss << endl;
    cout << "array a after sort:" << ss.str() << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
array a:12 34 65 -23 -32 33 61 99 321 32
array a after sort:-32 -23 12 32 33 34 61 65 99 321
```

总结：

★ 存储形式为string，不需要用户考虑空间

★ 使用方式同iostream/fstream基本相似(部分细节可能不同)

★ 如果结果与预期不同，多判断good()/fail()

★ C++还有一个strstream系列，但是在新标准中已是deprecated

● 要求：能读懂别人用strstream写的代码

§10. 输入输出流

5.6. 用字符串流对象实现不同数据类型的转换

例1：字符串转double

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream in("123.456");
    double d;

    in >> d;
    cout << d << endl;

    return 0;
}
```

例2：double转字符串

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    ostringstream out;
    double d = 123.456;
    char str[10];

    out << d;
    strcpy(str, out.str().c_str());
    cout << str << end;

    return 0;
}
```

§10. 输入输出流

5.6. 用字符串流对象实现不同数据类型的转换

例3：多种类型转字符串（重载方式）

```
#include <iostream>
#include <sstream>
using namespace std;
string tj_to_string(const double d)
{
    ostringstream out;
    out << d;
    return out.str();
}
string tj_to_string(const int i)
{
    ostringstream out;
    out << i;
    return out.str();
}
string tj_to_string(const char ch)
{
    ostringstream out;
    out << ch;
    return out.str();
}
int main()
{
    string s1 = tj_to_string(123.456);
    string s2 = tj_to_string(12345);
    string s3 = tj_to_string('A');

    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;
}
```

例4：多种类型转字符串（模板方式）

```
#include <iostream>
#include <sstream>
using namespace std;

template <class T>
string tj_to_string(const T &value)
{
    ostringstream out;
    out << value;
    return out.str();
}

int main()
{
    string s1 = tj_to_string(123.456);
    string s2 = tj_to_string(12345);
    string s3 = tj_to_string('A');

    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;
    return 0;
}
```

例5：多种类型互转（模板方式）

```
#include <iostream>
#include <sstream>
using namespace std;

template <class out_type, class in_type>
out_type tj_convert(const in_type &value)
{
    stringstream ss;
    ss << value;

    out_type ret;
    ss >> ret;
    return ret;
}
int main()
{
    string s = tj_convert<string, double>(123.456);
    cout << s << endl;

    double d = tj_convert<double, string>("12.34abc");
    cout << d << endl;

    int k = tj_convert<int, double>(123.456);
    cout << k << endl;

    return 0;
}
```

§10. 输入输出流

5.7. C方式类似字符串流的操作：sscanf与sprintf函数

5.7.1. 向字符串输出函数(将格式化输出的内容放入字符串中)：

int sprintf(字符数组, "格式串", 输出表列);

返回值是输出字符的个数(同printf)

字符数组要有足够空间容纳输出的数据(否则越界错)

格式串同printf

VS下需加 #define _CRT_SECURE_NO_WARNINGS

5.7.2. 从字符串中输入函数(从字符串中进行格式化输入)：

int sscanf(字符数组, "格式串", 输入地址表列);

返回值是正确读入的输入数据的个数(同scanf)

格式串同scanf

VS 下需加 #define _CRT_SECURE_NO_WARNINGS

```c
//将不同数据输出到字符数组中
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    char c[80];
    int ret;
    ret = sprintf(c, "%s%d%.1f", "Hello", 10, 11.2);
    printf("%s\n", c);
    printf("ret=%d\n", ret); //理解ret的含义

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello1011.2
ret=11
```

```c
//将结构体的内容输出到一维字符数组中
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

struct student {
    int num;
    char name[20];
    float score;
};

int main()
{
    struct student stud[3]={1001,"Li",78, 1002,"Wang",89.5, 1004,"Fun",90};
    char c[50], *s = c;
    for (int i=0; i<3; i++)
        s+=sprintf(s, "%d %s %.1f", stud[i].num , stud[i].name, stud[i].score);
    printf("array c:%s\n", c);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
array c:1001 Li 78.01002 Wang 89.51004 Fun 90.0
```

多次向字符数组输出格式化数据

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    char  c[80] = "Hello 10 11.2";
    char  s[10];
    int   i, ret;
    float f;

    ret = sscanf(c, "%s%d%f", s, &i, &f);

    printf("%s%d%.1f\n", s, i, f);
    printf("ret=%d\n", ret);

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Hello1011.2
ret=3
```

§10. 输入输出流

6. C方式的文件操作

6.1. 文件指针

FILE *文件指针变量

★ FILE是系统定义的结构体

★ C语言中文件操作的基本依据，所有针对文件的操作均要依据该指针

★ #include <stdio.h> （VS2022可以不需要）

★ VS2022以为不安全，需要加 #define _CRT_SECURE_NO_WARNINGS

★ 文件读写后，文件指针会自动后移

§10. 输入输出流

6.2. 文件的打开与关闭

假设：FILE *fp定义一个文件指针

6.2.1. 文件的打开

FILE *fopen(文件名，打开方式)

fp = fopen("test.dat", "r");

fp = fopen("c:\\demo\\test.dat", "w");

//也可表示为："c:/demo/test.dat"

★ 打开的基本方式如下：

r：只读方式

w：只写方式

a：追加方式

+：可读可写

b：二进制

t：文本方式(缺省)

★ 打开的基本方式及组合见右表

★ 若带路径，则\必须用\\表示

★ 若打开不成功，则返回NULL，表示打开操作不成功

打开方式	意义
r/rt	只读方式打开文本文件(不存在则失败)
w/wt	只写方式打开或建立文本文件(存在则清零)
a/at	追加写方式打开或建立文本文件(头读尾写)
rb	只读方式打开二进制文件(不存在则失败)
wb	只写打开或建立二进制文件(存在则清零)
ab	追加写方式打开或建立二进制文件(头读尾写)
r+/rt+	读写方式打开文本文件(不存在则失败)
w+/wt+	读写方式创建文本文件(存在则清零)
a+/at+	读+追加写方式打开或建立文本文件(头读尾写)
rb+	读写方式打开二进制文件(不存在则失败)
wb+	读写方式创建二进制文件(存在则清零)
ab+	读+追加写方式打开二进制文件(头读尾写)

6.2.2. 文件的关闭

fclose(文件指针);

返回值：正常返回0，异常返回EOF(-1)，表示文件在关闭时发生错误

6.2.3. 文件的访问途径（文件打开模式）

文本模式：程序所见的内容与文本实际内容不同。程序会把本地环境表示的行末尾或者文件结尾映射为C模式。

二进制模式：程序可以访问文件的每个字节，故访问的内容就是文件中存放的内容。

§10. 输入输出流

6.3. 文本文件的读写

6.3.1. 按字符读写文件

读：int fgetc(文件指针)

- 返回读到字符的ASCII码(返回值同getchar)

写：int fputc(字符常量/变量，文件指针)

- 返回写入字符的ASCII码(返回值同putchar)

★ 必须保证文件的打开方式符合要求

```
char ch1;
ch1=fgetc(fp);

char ch2 = 'A';
fputc(ch2, fp);
```

6.3.2. 判断文件是否到达尾部

int feof(文件指针)

★ 若到达尾部，返回1，否则为0

6.3.3. 按格式读写文件

读：int fscanf(文件指针，格式串，输入表列)

- 返回读取正确的数量(返回值同scanf)

写：int fprintf(文件指针，格式串，输出表列)

- 返回输出字符的个数(返回值同printf)

★ 格式串、输入/输出表列的使用同scanf/printf

```
int i;
char ch;
fscanf(fp, "%d%c", &i, &ch);

int i=10;
char ch='A';
fprintf(fp, "%d%c", i, ch);
```

§10. 输入输出流

6.3. 文本文件的读写

6.3.4. 用文件方式进行标准输入输出

stdin ：标准输入设备
stdout ：标准输出设备
stderr ：错误输出设备

这三个是系统预置的FILE *，直接用，不需要打开关闭

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

int main()
{
    int i;
    char ch;                                  等价
    ch = fgetc(stdin);                  ⇔ getchar();
    putchar(ch);
    fputc('A', stdout);                 ⇔ putchar('A');
    fscanf(stdin, "%d", &i);            ⇔ scanf("%d", &i);
    fprintf(stdout, "i=%d\n", i);       ⇔ printf("i=%d\n", i);
    fprintf(stderr, "i=%d\n", i);       ⇔ cerr<<"i="<<i<< endl;
                                          //C方式无专用错误输出
    return 0;                             //perror()功能不同
}
```

§10. 输入输出流

6.3. 文本文件的读写

6.3.5. 用freopen重定向标准输入输出

★ FILE *freopen(文件名，打开方式，原FILE *);

功能：将已存在的FILE *映射为另一个新的FILE *（把预定义的标准流文件重定向到指定的文件中）

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    FILE *fp;
    if ((fp = freopen("out.txt", "w", stdout))==NULL) {
        printf("freopen failed!\n");
        return -1;
        }
    printf("Hello, world!\n");
    fclose(fp);

    return 0;
}
```

1、正常运行，观察运行结果
2、不删除已存在的out.txt，换成"r"，观察运行结果
3、先删除已存在的out.txt，换成"r"，观察运行结果
4、如果在fclose的后面再加printf，能否正常输出？如果可以，输出到哪里了？如果没有，为什么？

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    FILE *fp;
    int a, b;
    if ((fp = freopen("in.txt", "r", stdin))==NULL) {
        printf("freopen failed!\n");
        return -1;
        }
    scanf("%d %d", &a, &b);
    printf("a=%d b=%d\n", a, b);
    fclose(fp);

    return 0;
}
```

1、在当前目录下建立in.txt文件，写入两个整数，观察运行结果
2、在当前目录下没有/删除in.txt的情况下运行，观察运行结果
3、如果在fclose的后面再加scanf，能否正常输入？如果可以，从哪里读？如果不行，为什么？

§10. 输入输出流

6.3. 文本文件的读写

6.3.5. 用freopen重定向标准输入输出

★ FILE *freopen(文件名，打开方式，原FILE *);功能：将已存在的FILE *映射为另一个新的FILE *

★ 用freopen可以重定向普通文件(一般不用)

★ 返回值：正常返回：所指定文件的指针
异常返回：NULL

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

int main()
{
    FILE *fp, *fdup;
    if ((fp = fopen("out.txt", "w")) == NULL) {
        printf("fopen out.txt failed!\n");
        return -1;
        }
    fprintf(fp, "Hello, world!\n");

    if ((fdup = freopen("out_dup.txt", "w", fp)) == NULL) {
        printf("freopen failed!\n");
        fclose(fp);
        return -1;
        }
    fprintf(fp, "I am a student.\n");

    fclose(fp);
    fclose(fdup);
    return 0;
}
```

§10. 输入输出流

6.3. 文本文件的读写

6.3.6. 用popen/pclose与系统命令进行交互

★ VS2022下是_popen与_pclose

★ Linux下的popen与pclose

★ Dev C++下popen/pclose/_popen/_pclose均可

★ Windows示例(分两步操作)

```
//假设编译为 D:\VS2019-Demo\Debug\demo-cpp.exe
#include <iostream>
using namespace std;
int main()
{
    cout << "Welcome to Tongji University!" << endl;
    return 0;
}
```

Step1

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{   FILE* fp = _popen("D:\\VS2019-Demo\\Debug\\demo-cpp.exe", "r");
    if (fp == NULL) {
        printf("popen failed!\n");
        return -1;
        }
    char ch;
    while (1) {
        ch = fgetc)fp);
        if (feof(fp);
            break;
        putchar(ch);
        }
    _pclose(fp);
    return 0;
}
```

再换为 "dir C:\Windows"

Step2

★ Linux示例(分两步操作)

```
//假设编译为 /home/u1234567/test
#include <iostream>
using namespace std;
int main()
{
    cout << "Welcome to Tongji University!" << endl;
    return 0;
}
```

Step1

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{   FILE* fp;
    if ((fp = popen("/home/u1234567/test", "r")) == NULL) {
        printf("popen failed!\n");
        return -1;
        }
    char ch;
    while (1) {
        ch = fgetc(fp);
        if ( feof(fp) )
            break;
        putchar(ch);
        }
    pclose(fp);
    return 0;
}
```

再换为 "ls -l /etc"

Step2

§10. 输入输出流

6.4. 二进制文件的读写

6.4.1. 按字符读写文件

读：int fgetc(文件指针)

- 返回读到字符的ASCII码(返回值同getchar)

写：int fputc(字符常量/变量，文件指针)

- 返回写入字符的ASCII码(返回值同putchar)

★ 必须保证文件的打开方式符合要求

★ 同C++方式，仅能按字符读写，且文件中不能有0x1A

6.4.2. 按块读写文件

读：int fread(缓冲区首址，块大小，块数，文件指针)

★ 返回读满的块数

写：int fwrite(缓冲区首址，块大小，块数，文件指针)

★ 返回写入成功的块数

§10. 输入输出流

6.5. 文件指针的移动

6.5.1. 指针复位(使指示文件位置的指针重新返回到文件开始)

void rewind(文件指针)

例：rewind(fp);

6.5.2. 任意移动

int fseek(文件指针，位移量，位移方式)

例：fseek(fp, 123, SEEK_SET)： 从开始移动

fseek(fp, 78, SEEK_CUR)：
fseek(fp, -25, SEEK_CUR)： 从当前位置移动

fseek(fp, -57, SEEK_END)： 从最后移动

★ SEEK_SET的位移必须为正

SEEK_CUR的位移可正可负

SEEK_END的位移必须为负

★ 返回值：正常返回：0

异常返回：-1(表示定位操作出错)

6.5.3. 求文件指针的当前位置

long ftell(文件指针)

例：ftell(fp);

★ 返回值：距文件开始处的字节数

§10. 输入输出流

6.5.4.其他随机读写函数

fseek()和ftell()潜在的问题是它们都把文件大小限制在long类型能表示的范围内

ANSI C新增了两个处理较大文件的新定位函数：fgetpos()和fsetpos()

这两个函数不使用long类型的值表示位置，它们使用一种新类型：fpos_t(代表file position type 文件定位类型)

6.5.4.1.fgetpos函数

函数原型：int fgetpos(FILE* fp, fpos_t* pos);

功能说明：获取流fp的当前文件位置，并把该fpos_t类型的值放在pos指向的位置上

参数说明：fp ：文件指针

pos：fpos_t类型的指针

返 回 值：正常返回：0

异常返回：非0

6.5.4.2.fsetpos函数

函数原型：int fsetpos(FILE* fp, const fpos_t* pos);

功能说明：使用pos指向位置上的fpos_t类型值来设置文件指针fp指向该值指定的位置

(fpos_t类型的值应通过之前调用fgetpos()获得)

参数说明：fp ：文件指针

pos：fpos_t类型的指针

返 回 值：正常返回：0

异常返回：非0

§10.输入输出流

6.5.4.其他随机读写函数

6.5.4.3.ungetc函数

函数原型：int ungetc(int c, FILE* fp);

功能说明：把c指定的字符放回输入流中，下次调用标准输入函数时将读取该字符

参数说明：c ：要被放回的字符，以其对应的int值进行传递

fp：文件指针

返 回 值：正常返回：被放回的字符

异常返回：EOF

6.5.4.4.fflush函数

函数原型：int fflush(FILE* fp);

功能说明：会强迫将缓冲区内的数据写回参数fp指定的文件中，如果参数fp为NULL，fflush()会将所有打开的文件数据更新

参数说明：fp：文件指针

返 回 值：正常返回：0(成功刷新/指定的流没有缓冲区/只读打开)

异常返回：EOF

§10. 输入输出流

6.5.4.其他随机读写函数

6.5.4.5.setvbuf函数

函数原型：int setvbuf(FILE* fp, char* buf, int mode, size_t size);

功能说明：设置文件的缓冲区

参数说明：指针fp识别待处理的流

buf指向待使用的存储区(期望缓冲区地址)，如果buf的值不是NULL，则必须创建一个缓冲区变量

size告诉setvbuf()数组的大小

说　　明：mode的选择如下：

_IOFBF表示完全缓冲(在缓冲区满时刷新)

_IOLBF表示行缓冲(在缓冲区满时或写入一个换行符时)

_IONBF表示无缓冲(没有缓冲区，直接从流中读入/写入数据)

返 回 值：正常返回：0

异常返回：非0

注　　意：打开文件之后且未对流进行其它操作之前调该函数

6.5.4.6.ferror函数

函数原型：int ferror(FILE* fp);

功能说明：判断文件是否出错

参数说明：fp：文件指针

返 回 值：正常返回：0

异常返回：非0

注　　意：同一文件每次调用输入输出函数，均产生一个新的ferror函数值

在执行fopen函数时，ferror函数的初始值自动置为0

§10. 输入输出流

6.6. 文件的读写函数

C语言提供四种顺序读写函数：字符读写、字符串读写、格式读写和数据块读写

fputc函数：向文件写入字符
fgetc函数：从文件读取字符
fputs函数：向文件写入字符串
fgets函数：从文件读取字符串
fprintf函数：向文件格式化写入数据
fscanf函数：从文件格式化读取数据
fwrite函数：向文件写入数据块
fread函数：从文件读取数据块

§10. 输入输出流

6.6. 文件的读写函数

6.6.1. fputc函数：向文件写入字符

函数原型：int fputc(int ch, FILE* fp);

函数说明：把ch中的字符写入fp所指的文件当前位置处，并使文件指针后移一个字符位置

参数说明：ch：整型变量，内存要写到文件中的字符

fp：文件指针，指向要在其中写入字符的文件

返 回 值：正常返回：写入的字符

异常返回：EOF(-1)

6.6.2. fgetc函数：从文件读取字符

函数原型：int fgetc(FILE* fp);

功能说明：从fp所指文件中读取一个字符，并使文件指针后移一个字符位置

参数说明：fp：文件指针，指向要从中读取字符的文件

返 回 值：正常返回：读取的字符

异常返回：EOF(-1)

返回类型的说明：用于区分有效数据和输入的结束(EOF)

若返回类型为char：一个字节（无法区分）

EOF ：0xFF

数据：0xFF

若返回类型为int：四个字节（正确区分）

EOF ：0xFFFFFFFF

数据：0x000000XX

§10. 输入输出流

6.6.文件的读写函数

6.6.3.fputs函数：向文件写入字符串

函数原型：int fputs(char* str, FILE* fp);

功能说明：将str指向的字符串的内容输出到fp所指向文件的当前位置，同时将fp自动向前移动strlen(str)个字符位置

参数说明：str：可以是字符串常量、字符串指针或字符数组名等

fp ：文件指针，指向字符串要写入其中的文件

返 回 值：正常返回：非负整数

异常返回：EOF(-1)

说　　明：字符串结束符('\0')不输出到文件

不自动在字符串末尾添加换行符

6.6.4.fgets函数：从文件读取字符串

函数原型：char* fgets(char* str, int n, FILE* fp);

功能说明：从由fp指出的文件中读取n-1个字符，并把它们存档到由str指出的字符数组中去，最后自动加上一个尾零

参数说明：str：接受字符串的内存地址，可以是数组名或指针

n ：指出要读取字符的个数

fp ：文件指针，指向要从中读取字符的文件

返 回 值：正常返回：字符串的内存首地址，即str的值

异常返回：NULL

说　　明：读入n-1个字符到文件，遇到换行符或文件结束符则提前结束文件

读入结束后，系统将自动在最后加上尾零

§10. 输入输出流

6.6.文件的读写函数

6.6.5.fprintf函数：向文件格式化写入数据

函数原型：int fprintf(FILE* fp, const char* format, arg_list);

功能说明：将变量表列(arg_list)中的数据，按照format指出的格式，写入由fp指定的文件

参数说明：fp　　：文件指针，指向将数据写入的文件

　　format　：指向写出数据格式字符串的指针

　　arg_list：要写入文件的变量表列，各变量之间用逗号分隔

返 回 值：正常返回：输出的字节数

　　异常返回：负值

示　　例：fprintf(fp, "&d%s", 4, "China");

　　表示将整数4和字符串"China"写入fp所指的文件中

6.6.6.fscanf函数：从文件格式化读取数据

函数原型：int fscanf(FILE* fp, const char* format, add_list);

功能说明：按照format指出的格式，从fp指定的文件中读取数据存放至地址表列(add_list)的变量中

参数说明：fp　　：文件指针，指向要从中读取数据的文件

　　format　：指向读取数据格式字符串的指针

　　add_list：存档读取数据的变量的地址表列

返 回 值：正常返回：成功读取的参数的个数(类似scanf)

　　异常返回：EOF(-1)

示　　例：fscanf(fp, "%d$d", &x, &y);

　　表示从fp所指的文件中顺序读取两个整数给变量x和y

§10. 输入输出流

6.6. 文件的读写函数

6.6.7. fwrite函数：向文件写入数据块

函数原型：size_t fwrite(void* buffer, size_t size, size_t count, FILE* fp);

功能说明：将以buffer为起始地址的长度为size的count个数据块输出到文件指针fp所指的位置去

参数说明：buffer：指向存放输出数据存储区的首地址的指针

size ：数据块的字节数，即一个数据块的大小

count ：一次输出数据块的数量

fp ：文件指针，指向要从其中写入数据的文件

返 回 值：正常返回：实际输出数据块的个数，即count

异常返回：0

6.6.8. fread函数：从文件读取数据块

函数原型：size_t fread(void* buffer, size_t size, size_t count, FILE* fp);

功能说明：从文件指针fp所指的文件的当前位置读取字节数为size大小的数据块共count个，存到buffer所指的内存储区中

参数说明：buffer：指向存放输入数据存储区的首地址的指针

size ：数据块的字节数，即一个数据块的大小

count ：一次读入数据块的数量

fp ：文件指针，指向要从其中读出数据的文件

返 回 值：正常返回：<=count(正常情况下，返回值就是count，但如果出现读取错误或读到文件结尾，返回值就会比count小)

异常返回：0

§10. 输入输出流

6.6. 文件的读写函数

6.6.7. fread和fwrite常用于读写二进制文件

例：fp指向以二进制形式打开的可读写文件，并有如下的说明

```
float f;
double d[10];
```

常见的块读写应用示例：

```
fwrite(&f, sizeof(float), 1, fp);      //把浮点数f写入文件
fwrite(d, sizeof(double), 10, fp);     //把数组d中所有数写入文件
fread(&f, sizeof(float), 1, fp);       //从文件中以块形式读一浮点数到变量f中
fread(d, sizeof(d), 1, fp);            //从文件中一次性读一个d大小的数据块到数组d中
fwrite(&d[0], sizeof(float), 1, fp);   //参数类型不匹配会导致数据错误(编译不错)
```

上述标准I/O函数都是面向文本的，用于处理字符和字符串

思考：如何要在文件中保存数值数据？

例：

```
double num = 1.0/3;
fprintf(fp, "%f", num);  //储存为8个字符：0.333333
```

一般而言，用fprintf()函数和%f转换说明只是把数值保存为字符串，fprintf()把数值转换为字符数据，这种转换可能会改变值

以二进制形式储存数据，不存在从数值形式到字符串的转换过程，保证数值在储存前后一致

§11. 运算符重载

1. 重载的引入与分类

函数重载(第04模块)：同一作用域内的不同函数具有相同的名称

★ 适用于完成不同个数、不同类型数据的相同操作

★ 函数参数的个数、类型不能完全相同

重载函数调用时的匹配查找顺序：

(1) 寻找参数个数、类型完全一致的定义 (严格匹配)

(2) 通过系统定义的内部转换寻找匹配函数(系统转换)

(3) 通过用户定义的内部转换寻找匹配函数(自定义转换)

★ 若某一步匹配成功，则不再进行下一顺序的匹配

★ 若每一步中发现两个以上的匹配则出错

运算符重载：同一运算符可以针对不同的数据类型完成相同的操作

```
#include <iostream>
using namespace std;
int max(int x, int y)
{
    cout << sizeof(x) << ' ';   1
    return (x > y ? x : y);
}
double max(double x, double y)
{
    cout << sizeof(x) << ' ';   2
    return (x > y ? x : y);
}
int main()
{
    cout << max(10,    15)      << endl;      严格匹配1
    cout << max(10.2,  15.3)    << endl;      严格匹配2
    cout << max(10,    int(15.3)) << endl;    系统转换1
//  cout << max(5+4i,  15.3)    << endl;      需自定义转换
    return 0;
}
```

```
4 15
8 15.3
4 15
```

复数形式目前编译会错，如何定义复数以及定义复数向double的转换，具体见荣誉课相关内容

§11. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法1：传统C方法
#include <iostream>
using namespace std;
struct Complex {
   double real;
   double imag;
};
struct Complex complex_add(struct Complex a, struct Complex b)
{
   Complex c;
   c.real = a.real+b.real;
   c.imag = a.imag+b.imag;
   return c;
}
int main()
{
   Complex c1={3,4}, c2={4,5}, c3;
   c3=complex_add(c1, c2);
   cout << c3.real << "+" << c3.imag << "i" << endl;    //输出时未考虑imag
}                                                       //为负,此处忽略
```

§11. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法2：C++成员函数方法
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r;   imag = i; }
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    Complex complex_add(Complex &b);
};
Complex Complex::complex_add(Complex &b)
{  Complex c;
   c.real = this->real+b.real;
   c.imag = this->imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4), c2(4,5), c3;
   c3=c1.complex_add(c2);
   c3.display();
}
```

this指针

```
因页面篇幅，未考虑imag为负,本章后面都相同
void Complex::display()
{   cout << real;
    if (imag>=0)   //考虑负数的体外实现方式
        cout << '+';
    cout << imag << 'i' << endl;
}
```

c3 = c1.complex_add(c2)时
this指针指向c1，形参为引用，
不调用复制构造
返回时调用复制构造函数

§11. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法3：C++运算符重载方式
#include <iostream>
using namespace std;

class Complex {
   ...
};

int main()
{  Complex c1, c2, c3;
   ...
   c3 = c1+c2;
}
```

```
c3 = c1.complex_add(c2);
```

```
c3 = complex_add(c1, c2);
```

§11. 运算符重载

2. 运算符重载的方法

方法：定义一个重载运算符的函数，当执行运算符时，自动执行该函数，达到相应的目的

★ 转换为函数重载，符合函数重载的匹配规则

形式：

```
返回类型 operator 运算符(形参表)
{
    重载函数实现
}
```

★ 用 operator 运算符 来表示对应运算符的函数

operator + ⇔ +

operator * ⇔ *

★ 对象 运算符 另一个值(可以不是对象、可以无)被解释为 对象.operator运算符(另一个值)

c1 + c2 ⇔ c1.operator+(c2)

★ 运算符被重载后，原来用于其它数据类型上的功能仍然被保留(重载)，系统根据重载函数的规则匹配

+ 仍然能表示两个int/两个float的加

§11. 运算符重载

2. 运算符重载的方法

例：复数的相加(实部、虚部对应相加)

```
//方法3：C++运算符重载方式
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r;   imag = i; }
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    Complex operator+(Complex &b);
};
Complex Complex::operator+(Complex &b)
{  Complex c;
   c.real=real+b.real;
   c.imag=imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

实现部分与方法2相同
两者除函数名外没什么不同
因此实现相同

c3 = c1.operator+(c2); 转为函数
c3 = c1.complex_add(c2);方法2

若希望以cout<<c3的形式输出，要重载cout

§11. 运算符重载

3. 运算符重载的规则

共9条

★ 对已有运算符进行重载，不能定义新运算符

★ 除5个运算符外都允许重载(唯一的三目 ?: 不允许)

★ 不能改变操作对象的个数

★ 不能改变优先级

★ 不能改变结合性

★ 不允许带默认参数

★ 重载运算符的两侧至少有一个是类对象

★ =和&系统缺省做了重载，=是对应内存拷贝,&取地址

★ 应当使重载运算符的功能与标准相同/相似(建议)

前例：c3=c1+c2
+是重载，=是缺省重载
可根据需要自己写=的重载

不能重载的运算符只有 5 个：

.	(成员访问运算符)
*	(成员指针访问运算符)
::	(域运算符)
sizeof	(长度运算符)
?:	(条件运算符)

§9. 动态内存申请

4. 含动态内存申请内存的类和对象

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

4.4.2. 含动态内存申请的对象的复制

当对象数据成员包含动态申请的内存指针时，都可能错

解决方法：

赋值：后续模块 重载=运算符

复制：自定义复制构造函数替代系统缺省的复制构造

§9. 动态内存申请

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

用多编译器分别
运行下面两个例子

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

有动态内存申请
执行结果错(与期望不同)

注意：
1、篇幅问题，假设申请成功
2、程序不完整，仅在构造函数中动态申请，未定义析构函数进行释放

```
hello
A
hello
china
china
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

有动态内存申请
执行结果错(与期望不同)

同左例，加入析构函数后，
不但执行结果错，而且
VS下会有错误弹窗，
GNU下虽然没有错误弹窗，
但仍然是错误的

```
hello
A
hello
china
china
```

4.4. 含动态内存
4.4.1. 含动态内
★ 若对象数据

错误原因的图解及具体解释：
1、造成4000-4019这20个字节的内存丢失
2、t1/t2的c成员同时指向一块内存，通过t1的c修改改内存块，会导致t2的c值同时改变
3、若定义了析构函数，则main函数执行完成后系统会调用析构函数(按t2,t1的顺序)，t2调用析构函数释放3000-3019后，再调用t1的析构函数会导致重复释放3000-3019，错!!!

如何保证有动态内存时的赋值正确性?

后续模块 重载=运算符

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();                hello
    t2.display();                A
    t2=t1;
    t2.display();                hello
    t1.set("china");
    t1.display();                china
    t2.display();                china
}
```

有动态内存申请
执行结果错(与期望不同)

同左例，加入析构函数后，
不但执行结果错，而且
VS下会有错误弹窗，
GNU下虽然没有错误弹窗，
但仍然是错误的

§11. 运算符重载

3. 运算符重载的规则

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

§11. 运算符重载

3. 运算符重载的规则

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

问：返回类型test &,不能是test,为什么？

答：t2=t1理解为t2.operator=(t1),即this指针指向t2,而=的语义希望执行后t2被改变，因此返回test&

语义：若返回test，则返回时会调用复制构造函数，返回的就是临时对象而不是t2自身,因此返回不能是test

正确性：使用了缺省的复制构造函数导致运行出错

=> 续：具体错在哪里？如何分析内存？

另一个例子： complex operator+(complex &b);

c1+c2理解为c1.operator+(c2)，+的语义不能改变c1，因此+应该返回临时对象，所以返回值是 complex 而不是 complex&

结论：

函数重载的返回值应该由运算符的语义决定

问：operator=的返回类型 void 可以吗？

答：1、对本题而言，void正确

2、如果出现类似t3=(t2=t1) 就会错误

=> 第2章 赋值表达式的值等于左值

=> 不可以void

§11. 运算符重载

3. 运算符重载的规则

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;      //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

因为申请/释放都是20字节，
因此可以仍旧使用原来已申请的空间
(蓝色两句全部不要)

若要求test类按需申请，不浪费空间
则必须释放原空间再申请新空间

```
int main()
{
    test t1("hello"),t2;
    ...
    t2=t1;
}
```

```
//假设构造函数按需申请
test(char *s="A")
{   a=0;
    b=0;
    c=new char[strlen(s)+1];
    strcpy(c, s);
}
```

```
test &test::operator=(const test &t)
{     a = t.a;
      b = t.b;
      delete []c;                  //释放原2字节   必须有
      c=new char[strlen(t.c)+1]; //申请新6字节
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

§11. 运算符重载

3. 运算符重载的规则

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

本程序的=重载，在出现自赋值情况(即t2=t2/t1=t1等形式，虽然无意义，但语法并不禁止)时会怎样？

1、观察运行结果
2、分析错误原因
3、如何才能修改正确

§11. 运算符重载

3. 运算符重载的规则

```
//复制构造函数和重载=的区别
class test {
  ...
  public:
    test(const test &t);              //复制构造函数的声明
    test &operator=(const test &t);   //重载=的声明
  };

test::test(const test &s)    //复制构造函数的体外实现
{   a=s.a;
    b=s.b;
    c=new char[20];     //申请新空间
    strcpy(c, s.c);
}

test &test::operator=(const test &t)    //重载=的体外实现
{     a = t.a;
      b = t.b;
      delete []c;      //释放原空间    (构造不需要)
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;    //返回对象自身（构造不需要）
}

int main()
{
    test t1("hello");
    test t2(t1); //复制构造函数
    t2 = t1;     //赋值运算
}
```

	复制构造函数	重载赋值运算符
系统缺省	有，对应内存拷贝	有，对应内存拷贝
必须定义的时机	含动态内存申请时	含动态内存申请时
调用时机	定义时用对象初始化 函数形参为对象 函数返回值为对象	执行语句中的=操作
调用时处理	新对象生成时调用，此时不可能调用其它形式的构造函数 从未动态内存申请， 因此不考虑释放	已有对象=操作时调用，在=前对象已生成，即已调用过某种形式的构造函数(包括复制构造函数) 已有动态内存申请， 因此要考虑释放

```
class test {
    ...
    test(char *s="A");
    test(const test &t);
};
//普通构造函数的体外实现
test::test(char *s)
{   a=0;
    b=0;
    c=new char[20];
    strcpy(c, s);
}
```

```
//复制构造函数的体外实现
test::test(const test &s)
{   a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
}
```

```
int main()
{
   test t1("hello"), t2=t1, t3;
    t3 = t2; //调用此句前，t1/t2/t3
             //已调用过不同的构造函数
}            //并动态申请过空间
```

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

类成员函数：

c1 + c2 ⇔ c1.operator+(c2)

友元函数：

全局函数做友元函数　　：合适

其它类的成员函数做友元函数：不合适

其他类做友元　　：不合适

★ 因为要访问类的成员，所以需要定义为友元

★ 运算符的同一种重载实现，友元/成员只能选择一个

★ 用全局普通函数也能实现运算符重载，但因为只能访问public对象，或通过类的公有成员函数访问private部分，因此效率低，一般不用

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

★ 运算符的同一种重载实现，友元/成员只能选择一个

例：复数的相加(实部、虚部对应相加)

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+"<< imag <<"i" << endl;
        }
    friend Complex operator+(Complex &a, Complex &b);
};
Complex operator+(Complex &a, Complex &b) //全局函数
{  Complex c;
   c.real=a.real+b.real;
   c.imag=a.imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

全局友元函数，没有this指针
通过 对象.成员 的形式调用
返回时调用复制构造函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+"<< imag <<"i" << endl;
        }
    Complex operator+(Complex &b);
};
Complex Complex::operator+(Complex &b) //成员函数
{   Complex c;
    c.real  = real  + b.real;
    c.image = image + b.image;
    return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

成员函数，有this指针
直接 成员 的形式调用
返回时调用复制构造函数

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

★ 用全局普通函数也能实现运算符重载，但因为只能访问public对象，或通过类的公有成员函数访问private部分，因此效率低，一般不用

```
#include <iostream>
using namespace std;

class Complex {
  public:
    double real;
    double imag;
    Complex(double r=0,double i=0) {
        real=r;
        imag=i;
        }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
};
Complex operator+(Complex &a, Complex &b) //全局非友元
{  Complex c;
   c.real=a.real+b.real;
   c.imag=a.imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

通过直接访问公有成员方式实现
因为成员公有，可被任意外部访问
封装性受影响，无法保护内部数据

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0,double i=0) {
       real=r;
       imag=i;
       }
    void display(){
       cout << real <<"+" << imag <<"i" <<endl;
       }
    double getreal() { return real; }
    double getimag() { return imag; }
    void   setreal(double r) { real = r; }
    void   setimag(double i) { imag = i; }
};
Complex operator+(Complex &a, Complex &b) //全局非友元
{  Complex c;   double real, imag;
   real = a.getreal() + b.getreal();
   imag = a.getimag() + b.getimag();
   c.setreal(real);   c.setimag(imag);
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

通过公有成员函数
访问私有成员，可
保护内部数据，但
效率受影响

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对单目运算符

成员函数是空参数

友元函数是一个参数(必须是对象) ← 规则7

★ 对双目运算符

成员函数是一个参数(可不是对象)

对象(this) 双目运算符 参数

友元函数是两个参数(一个必须是, 一个可不是)

第一个参数 双目运算符 第二个参数

- 对两个都是对象的情况：没区别
- 对一个是对象的情况：

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望 c2 = 4+c1 正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
       {   real = r;    imag = i; }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    friend Complex operator+(Complex &a, double b);
  };
Complex operator+(Complex &a, double b) //全局函数
{  Complex c;
   c.real=a.real+b;//实部相加
   c.imag=a.imag;  //虚部不变
   return c;
}
int main()
{ Complex c1(3,4),c2;
  c2 = c1 + 4;  //正确
  c2 = 4 + c1;  //编译错
}
```

友元函数形式，
实现 Complex+double

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
 public:
    Complex(double r=0, double i=0)
       {   real = r;    imag = i; }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    Complex operator+(double b);
  };
Complex Complex::operator+(double b) //成员函数
{  Complex c;
   c.real=real+b;//实部相加
   c.imag=imag;  //虚部不变
   return c;
}
int main()
{ Complex c1(3,4),c2;
  c2 = c1 + 4;  //正确
  c2 = 4 + c1;  //编译错
}
```

成员函数形式，
实现 Complex+double

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);    //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望c2 = 4 + c1正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

```
class Complex {
      ...
  public:
    Complex operator+(double b);
    friend Complex operator+(double a, Complex &b);
  };
Complex Complex::operator+(double b)     //成员函数
{  Complex c;
   c.real=real+b;  //实部相加
   c.imag=imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)  //友元函数
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

友元/成员重载

c1 + 4

4 + c1

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望c2 = 4 + c1正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

无法做到两个成员函数重载，
因为 4 + c1 无法表示为成员函数形式
原因：第1个参数(左值)不是类

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

关于+交换律的说明：
1、两个对象+，即定义两个类的+重载后，无论 c1+c2
还是 c2+c1，都调用同一重载函数，
具体实现不同（ c1.operator+(c2) / c2.operator+(c1) ）
但结果相同，可以理解为交换律存在
2、对象+其它类型，例如 Complex+double，交换律不存在，
交换律的表面现象是通过多个函数的重载来实现的
3、同理适用于其它存在交换律的运算符

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double换成double &，观察区别

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);    //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double &b);  //友元函数
    friend Complex operator+(double  &a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double &b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double &a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //错
  c2.display();
  c3 = 5 + c1; //错
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double换成double &，观察区别

```
class Complex {
      ...
  public:
    friend Complex opera
    friend Complex opera
  };
Complex operator+(Comple
{  Complex c;
   c.real=a.real+b;   //
   c.imag=a.imag;     //
   return c;
}
Complex operator+(double
{  Complex c;
   c.real=a+b.real;   //
   c.imag=b.imag;     //
   return c;
}
```

情况：

Complex operator+(Complex &a, double b)

c2 = c1 + double型常量　正确

c2 = c1 + double型变量　正确

Complex operator+(Complex &a, double &b)

c2 = c1 + double型常量　错误

c2 = c1 + double型变量　正确

原因：引用是变量的别名，因此若函数形参为引用则实参只能是变量，不能是常量/表达式

两个友元重载

```
ble &b);  //友元函数
plex &b); //友元函数
```

+ 4

c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //错
  c2.display();
  c3 = 5 + c1; //错
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double &换成const double &，即可编译正确

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);    //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

两个友元重载

```
class Complex {
      ...
  public:
   friend Complex operator+(Complex &a, const double &b); //友元
   friend Complex operator+(const double  &a, Complex &b);//友元
  };
Complex operator+(Complex &a, const double &b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(const double &a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§11. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 建议对单目运算符采用成员函数方式，双目运算符采用友元函数方式

★ C++规定，某些运算符必须是成员函数形式(赋值=，下标[],函数())，某些运算符必须是友元函数形式(流插入<<，流提取>>)

§11. 运算符重载

5. 重载单目运算符

例：假设复数的-，规则为实部虚部全部换符号

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
    {   real = r; imag = i;
    }
    void display()
    {   cout << real <<"+" << imag <<"i" <<endl;
    }
    Complex operator-();
};
```

成员函数

为什么返回值不是complex & ？
答：不能返回自动变量的引用

```
Complex Complex::operator-( )      //成员函数
{   Complex c1;
    c1.real = -real;  //实部取反
    c1.imag = -imag;  //虚部取反
    return c1;
}
```

为什么不能如下方式实现 ？

```
Complex& Complex::operator-()
{  real = -real;  //实部取反
   imag = -imag;  //虚部取反
   return  *this;
}
```

答：因为-的语义不改变对象自身，所以要返回临时对象

同理：

```
int k = 10, m;
2*(-k); //k仍是10
m=-k;   //k仍是10
```

```
int main()
{   Complex c1(3,4),c2;
    c2 = -c1;
    c1.display();
    c2.display(); //输出-3+-4i形式
    return 0;
}
```

若仅有c2.display()，则看不出差异，测试用例很重要!!!

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
    {   real = r; imag = i;
    }
    void display()
    {   cout << real <<"+" << imag <<"i" <<endl;
    }
    friend Complex operator-(Complex &a);
};
```

友元函数

```
Complex operator-(Complex &a)  //友元函数
{   Complex c1;
    c1.real = -a.real;  //实部取反
    c1.imag = -a.imag;  //虚部取反
    return c1;
}
```

为什么不能如下方式实现 ？

```
Complex& operator-(Complex &a)
{  a.real = -a.real;//实部取反
   a.imag = -a.imag;//虚部取反
   return  a;
}
```

答：因为-的语义不改变对象自身，所以要返回临时对象

同理：

```
int k = 10, m;
2*(-k); //k仍是10
m=-k;   //k仍是10
```

```
int main()
{   Complex c1(3,4),c2;
    c2 = -c1;
    c1.display();
    c2.display(); //输出-3+-4i形式
    return 0;
}
```

若仅有c2.display()，则看不出差异，测试用例很重要!!!

§11. 运算符重载

5. 重载单目运算符

★ ++/--的前后缀区别

前缀：正常方式

后缀：多一个int型参数，不访问，仅进行区别

例：假设复数的++是实部++，虚部不动

为什么前缀++/--返回引用，后缀++/--返回对象？

答：根据语义，前缀是自身先++/--，再自身参与运算
因此返回引用，即对象自身，且不需调用复制构造
后缀是保存旧值，自身++/--，再旧值参与运算
因此返回对象，返回时调用复制构造产生临时对象

成员函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r; imag = i; }
    void display(){ cout << real <<"+" << imag <<"i" <<endl;}
    Complex& operator++();      //前缀
    Complex  operator++(int);   //后缀
};
Complex& Complex::operator++() //前缀
{  real++;              //前后缀无所谓
   return *this;        //返回自身
}
Complex Complex::operator++(int) //后缀
{  Complex c1(*this); //复制构造函数，用对象自身初始化c1
   real++;              //前后缀无所谓，注意不是c1.real
   return c1;           //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r; imag = i; }
    void display(){ cout << real <<"+" << imag <<"i" <<endl;}
    friend Complex& operator++(Complex &a);      //前缀
    friend Complex  operator++(Complex &a, int); //后缀
  };
Complex& operator++(Complex &a) //前缀
{  a.real++;            //前后缀无所谓
   return a;            //全局函数没有this指针
}
Complex operator++(Complex &a, int) //后缀
{  Complex c1(a);       //复制构造函数，用对象自身初始化c1
   a.real++;            //前后缀无所谓，注意不是c1.real
   return c1;           //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

§11. 运算符重载

6. 重载流插入运算符和流提取运算符

6.1. 形式

```
istream& operator >> (istream &, 自定义类 &);
ostream& operator << (ostream &, 自定义类 &);
```

★ >>本身是istream类(系统定义类)的成员函数，因此希望对istream类的 >> 运算符重载，使其能输入自定义类的内容

★ <<本身是ostream类(系统定义类)的成员函数，因此希望对ostream类的 << 运算符重载，使其能输出自定义类的内容

★ 不能用自定义类的成员函数方式来实现

假设：operator<< (Complex &c1, ostream &cout)

则意味着使用形式为： c1 << cout

结论：流插入和流提取不能通过重载Complex类来实现，而应该是重载istream/ostream类使其能接受自定义类

§11. 运算符重载

6. 重载流插入运算符和流提取运算符

6.2. 重载流提取运算符<<

例：假设复数的输出形式为 实部+虚部i

只能友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    friend ostream& operator << (ostream &out, Complex &a);
 };

ostream& operator<<(ostream &out, Complex &a)
{   out << a.real;
    if (a.imag>=0)
        out << '+';  //虚部小于0不需要+  (3-4i)
    out << a.imag << 'i';
    return out;
}


int main()
{   Complex c1(3,4);
    cout << c1 << endl;
}
```

§11. 运算符重载

6. 重载流插入运算符和流提取运算符

6.3. 重载流插入运算符>>

例：假设复数的输入形式为 实部 虚部

只能友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    friend istream& operator >> (istream &in, Complex &a);
 };
istream& operator>>(istream &in, Complex &a)
{   in >>  a.real >> a.imag;
    return in;
}
int main()
{   Complex c1;
    cin >> c1;
    c1.display();
}
```

§11. 运算符重载

7. 不同类型数据间的转换

7.1. 标准类型数据间的转换

隐式转换：　int + double;　//转换优先级、整型提升

　　int = double;　//以左值为准进行转换

显式转换：

　　C方式：　(int)89.5

　　C++方式：　int(89.5)　/ static_cast<int>(89.5)

★ 问题的引入：对于自定义类型，能否进行类型转换

```
class Time {
    ...; //假设包含hour/minute/second成员
};

int main()
{
   Time t1(14,15,23), t2;
   int sec;
   t2 = 3662; //期望t2为  1:1:2 (目前无法做到)
   sec = t1;  //期望sec为 51323 (目前无法做到)
}
```

```
class Complex {
    ...;  //假设实现0参和2参构造，无1参构造
};

int main()
{
   Complex c1(2.5, 3.5), c2;
   double d;
   c2 = 5.3; //期望c2为 5.3+0i (目前无法做到)
   d  = c1;  //期望d 为 2.5    (目前无法做到)
}
```

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

形式：

```
类名(一个形参)
{
   函数实现
}
```

★ 只能带一个参数，非该类的数据类型，将该参数转换为对应的对象

★ 系统无缺省定义，根据需要自行定义并使用

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

例：将double转换为Complex的规则，成为对应实部

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(){ real=0; imag=0;}
    Complex(double r, double i){ real=r; imag=i;}
    Complex operator+(const Complex &b)  //const引用
      { return  Complex(real+b.real, imag+b.imag);}
    Complex(double r) { real = r; imag = 0; } //转换构造
    void display()
      { cout << real <<"+"<< imag <<"i" << endl; }
};

int main()
{
  Complex c1(3,5), c2;
  c2=c1+2.5;            5.5+5i
  c2.display();
  c2=c1+Complex(2.5);   5.5+5i
  c2.display();

  return 0;
}
```

隐式调用转换构造函数，建立无名Complex对象，再与c1相加

显式调用转换构造函数，建立无名Complex对象，再与c1相加

重载函数调用时的匹配查找顺序：

（1）寻找参数个数、类型完全一致的定义（严格匹配）

（2）通过系统定义的内部转换寻找匹配函数

（3）通过用户定义的内部转换寻找匹配函数

★ 若每一步中发现两个以上的匹配则出错

	c1+2.5
(1)是否有Complex+double?	无
(2)是否有系统内部定义的转换?	无
(3)是否有用户定义的内部转换?	有　d => C

	c1+Complex(2.5)
(1)是否有Complex+Compelx?	有

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

例：将double转换为Complex的规则，成为对应实部

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(){ real=0; imag=0;}
    Complex(double r, double i){ real=r; imag=i;}
    Complex operator+(const Complex &b)   //const引用
      { return  Complex(real+b.real, imag+b.imag);}
    Complex(double r) { real = r; imag = 0; } //转换构造
    void display()
      { cout << real <<"+"<< imag <<"i" << endl; }
};

int main()
{
  Complex c1(3,5), c2;
  c2=c1+2.5;            5.5+5i
  c2.display();
  c2=c1+Complex(2.5);   5.5+5i
  c2.display();

  return 0;
}
```

问：为什么不能合并为
Complex(double r=0, double i=0)
{ real = r; imag = i; }
答：若缺省为一个参数时，与转换构造存在二义性

问：
1、为什么不用 Complex b
答：每次调用复制构造，效率低

2、为什么不用 Complex &b
答：实参不能是常量/表达式

隐式调用转换构造函数，建立无名Complex对象，再与c1相加

显式调用转换构造函数，建立无名Complex对象，再与c1相加

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

例：整数n(0-86399)表示当天的秒，转换为Time对象

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time() {hour=0; minute=0; sec=0;}
    Time(int second);
    Time(int h, int m, int s) { hour = h; minute = m; sec = s; }
    Time operator-(Time &t);
    void display(){ cout << hour << ":" << minute << ":" << sec <<endl; }
};
Time Time::operator-(Time &t)
{  int s;   Time tt;
   s = (hour*3600+minute*60+sec) - (t.hour*3600+t.minute*60+t.sec);
   tt.hour=s/3600;
   tt.minute=s%3600/60;
   tt.sec=s%60;
   return tt;
}
Time::Time(int second)
{  hour = second/3600; minute = second%3600/60; sec = second % 60;
}
int main()
{ Time t1, t2(3,0,0);
  t1 = Time(65234)-t2;
  t1.display();
}
```

```
65234 = 18:7:34
65234-3600*3 = 15:7:34
```

```
15:7:34
```

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

★ 单一个参数的构造函数也可以不做为转换构造函数只有完成转换功能才称为转换构造函数

例：Time类

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int second);
   };

Time::Time(int second)     //转换构造
{
    hour   = second / 3600;
    minute = second % 3600 / 60;
    sec    = second % 60;
}
```

例：Time类

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int second);
   };

Time::Time(int second)  //带一个参数
{                       //的普通构造
    hour   = 0;
    minute = 0;
    sec    = second % 60;
}
```

两者无法重载，因为函数名、参数个数、参数类型完全相同

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

★ 两种使用方式

类名(要转换类型的数据)　显式调用

对象名 = 要转换的数据　隐式调用

假设已实现无参/两参构造函数及两个复数+及转换构造

```
Complex c1(3,5), c2;
c2 = 3.5;                  //正确(隐式)
c2 = c1 + Complex(3.5);    //正确(显式)
c2 = c1 + 3.5;             //正确(当复数+的参数是const引用时)
```

§11. 运算符重载

7. 不同类型数据间的转换

7.2. 用转换构造函数进行类型转换

★ 也可以将一个类的对象转换为另一个类的对象

类名(要转换类型的对象)

对象名 = 要转换的另一个对象

例：假设两个类表示身高，单位分别是厘米和米

```
class B;
class A {
  private:
    int height;
  public:
    A(int h) { height = h; }
    friend B;
};
class B {
  private:
    double height;
  public:
    B() { height = 0; }
    B(A ha) { height = ha.height/100.0; }
    void display() { cout << height << endl; }
};
```

```
int main()
{
  A ha1(178), ha2(183);
  B hb1, hb2;

  hb1 = B(ha1);
  hb1.display();

  hb2 = ha2;
  hb2.display();

  return 0;
}
```

```
1.78
1.83
```

§11. 运算符重载

7. 不同类型数据间的转换

7.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

例：将Time类转换为秒

```
#include <iostream>
using namespace std;

class Time {
  private:
    int  hour;
    int  minute;
    int  sec;
  public:
    Time(int h=0, int m=0, int s=0) {
        hour = h;
        minute = m;
        sec = s;
        }
    operator int();    //类型转换函数
};

Time::operator int()  //类型转换函数的体外实现
{
    return hour*3600 + minute*60 + sec;
}

int main()
{
    Time t1(14,15,23);
    int k;
    k = 10 + t1;
    cout << k << endl;
}
```

14*3600 + 15*60 + 23 = 51323

51333

10 + t1，因为没有定义 int + Time 的重载，因此t1被转换为int(隐式调用类型转换函数)，再int相加

§11. 运算符重载

7. 不同类型数据间的转换

7.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

重载函数的匹配顺序：
(1)是否有double+Complex?
无
(2)是否有系统内部转换?
无
(3)是否有用户定义转换?
有 C => d

2.5 + c1，因为没有定义double+复数的重载，因此c1被转换为double
(隐式调用类型转换函数)，再double相加

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {
        real = 0;
        imag = 0;
        }
    Complex(double r, double i) {
        real = r;
        imag = i;
        }
    operator double() {
        return real;
        }
};

int main()
{
   Complex c1(3,4);
   double d1;
   d1 = 2.5 + c1;          5.5
   cout << d1 << endl;
}
```

§11. 运算符重载

7. 不同类型数据间的转换

7.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

编译错

1、无 double+复数 的重载

2、无 复数转double 的类型转换函数，也无法理解为 double+

3、无 double转复数 的转换构造函数及 复数+复数 的重载，也无法理解为复数+

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {
        real = 0;
        imag = 0;
        }
    Complex(double r, double i) {
        real = r;
        imag = i;
        }
//  operator double() {
//      return real;
//      }
};

int main()
{
   Complex c1(3,4);
   double d1;
   d1 = 2.5 + c1;
   cout << d1 << endl;
}
```

§11. 运算符重载

7. 不同类型数据间的转换

7.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {  real = 0;   imag = 0;  }
    Complex(double r, double i){
      real = r;
      imag = i;
      }
    operator double() {  return real;  }
    Complex operator+(const Complex &b) {
      return Complex(real+b.real, imag+b.imag);
      }
};

int main()
{
   Complex c1(3,4),c2(5,-10);
   double d1;
   d1 = 2.5 + c1;
   cout << d1 << endl;     5.5
   d1 = c1 + c2;
   cout << d1 << endl;     8
}
```

2.5 + c1，
因为没有定义double+复数的重载，
因此c1被转换为double
(隐式调用类型转换函数)，
再double相加，得5.5

c1 + c2，
复数的+重载得到8-6i，
赋值给d1时被转换为double
(隐式调用类型转换函数)，
d1的值为8

§11. 运算符重载

7. 不同类型数据间的转换

7.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

假设已实现类型转换及两个复数的+

```
d1 = 2.3 + c1; //c1     被隐式转换
d1 = c1  + c2; //c1+c2  被隐式转换
```

★ 定义转换函数后，注意与转换构造函数的区别

§11. 运算符重载

7. 不同类型数据间的转换

★ 拓展讨论：转换构造函数及类型转换函数的使用

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    Complex(double r) { real = r; imag = 0; }    //转换构造函数
    operator double() { return real; }           //类型转换函数
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    //注意：下面两个用的时候只能选择一个，另一个要注释掉!!!!!!!!!!
    Complex operator+(const Complex &c2);
    friend Complex operator+(const Complex &c1, const Complex &c2);
};
```

```
Complex Complex::operator+(const Complex &c2)
{   return Complex(real+c2.real, imag+c2.imag);
}
Complex operator+(const Complex &c1, const Complex &c2)
{   return Complex(c1.real+c2.real, c1.imag+c2.imag);
}
```

//注意：这两个用的时候
//只能选择一个，另一个
//要注释掉!!!!!!!!!!

```
int main()
{  Complex c1(3,4), c3;
   c3 = c1 + Complex(2.5);
   c3.display();
   c3 = c1 + 2.5;
   c3.display();
   c3 = 2.5 + c1;
   c3.display();
}
```

	+的实现	c3=c1+Complex(2.5)	c3=c1+2.5	c3=2.5+c1
无转换构造 无类型转换	友元			
	成员			
无转换构造 有类型转换	友元			
	成员			
有转换构造 无类型转换	友元			
	成员			
有转换构造 有类型转换	友元			
	成员			

§12. 继承和派生

1. 基本概念

重用：事物不加修改或稍作修改即可重复使用

软件代码的可重用性：软件代码可重复利用

★ 一些经典算法及经典结构可以重用

★ 用户界面部分一般超过70%的代码可重用

继承：C++中实现重用的机制

类的继承：在一个已有类的基础上建立一个新类，新类可以从已有类那里获得已有特征，这种现象称为类的继承

类的派生：对继承而言，可以理解为已有类派生出新类

教务处：

```
class student {
    private:
        int  sno;        //学号
        char sname[20];  //姓名
        char ssex;       //性别
        int  sage;       //年龄
        char sdept[16];  //系部
        char saddr[64];  //地址

  public:
   各种函数
};
```

学生处：

```
class student_2 {
    private:
        int  sno;
        char sname[20];
        char ssex;
        int  sage;
        char sdept[16];
        char saddr[64];
        char ssc;//奖惩情况

    public:
    各种函数
};
```

目前是两个独立的类
但有很多成员相同
能否在student的基础上
定义student_2
这样可以少做很多工作

§12. 继承和派生

1. 基本概念

基类与派生类：基类(父类)，指已有的类

派生类(子类)，新建立的类

★ 派生类是基类的细化，基类是派生类的抽象

单继承与多继承：一个派生类从一个基类派生，则称为单继承，否则称为多继承

§12. 继承和派生

2. 派生类的声明方式

形式：

```
class 派生类名:private/public 基类名1, private/public 基类名2, ..., private/public 基类名n {
    private:
        私有成员;
    public:
        公有成员;
};
```

当只有一个基类名时，单继承
当多于一个基类名时，多继承

★ 基类名前的private/public称为基类存取限定符根据限定符的不同分别称为私有继承和公有继承

★ 若基类前不加限定符，缺省是private

★ 根据需要加入派生类自己特有的成员

```
class 派生类名 : 基类名 {
   private:
     私有成员;
   public:
     公有成员;
};
```

§12. 继承和派生

2. 派生类的声明方式

★ 基类名前的private/public称为基类存取限定符根据限定符的不同分别称为私有继承和公有继承

★ 若基类前不加限定符，缺省是private

★ 根据需要加入派生类自己特有的成员

教务处：

```
class student {
  private:
    int  sno;          //学号
    ...
    char saddr[64]; //地址
  public:
    各种函数
};
```

基类的成员表示为私有不太妥当，先这样表示

学生处：

```
class student_2:public student {
  private:
     char ssc;//奖惩情况
  public:
     ...
};
```

student类：

① 可定义对象/指针/数组/引用

② 可做函数参数、返回值

③ 可通过student类的对象访问student类的数据成员及成员函数(按访问规则)

特别强调：student做为一个类，可以正常使用，不要误解为被student_2继承后，只能用student_2

student_2类

① 可定义对象/指针/数组/引用

② 可做函数参数、返回值

③ 可通过student_2类的对象访问student_2类的数据成员及成员函数(按访问规则)

④ 可通过student_2类的对象访问student类的数据成员及成员函数(具体待讨论)

§12. 继承和派生

2. 派生类的声明方式

形式：

```
class 派生类名:private/public 基类名1, private/public 基类名2, ..., private/public 基类名n {
    private:
        私有成员;
    public:
        公有成员;
};
```

★ 对于多继承，派生类对每个基类的继承方式可不同

```
class A {
    ...
};
class B {
    ...
};
class C:public A, private B {
    ...
};
```

★ 暂不讨论基类的数据成员与成员函数与派生类同名的问题

§12. 继承和派生

3. 派生类的构成

派生类对象所占的空间：

基类数据成员所占空间总和 + 派生类数据成员所占空间的总和

派生类可访问的成员函数：

基类成员函数 + 派生类成员函数

class A {
priavte:
int a;
short b;
char c;
public:
void f1();
int f2();
};
	class B:public A {
	priavte:
	int d;
	short e;
	char f;
	public:
	void f3();
	int f4();
	};
		B b1;
		
		b1.f1();
		b1.f2();
		b1.f3();
		b1.f4();
			a
			b
			c
			///
			d
			e
			f
			///

★ 继承基类的全部数据成员(不一定都可以访问)

★ 继承基类除构造函数和析构函数外的全部成员函数(不一定都可以访问)

★ 友元不能继承

★ 派生类的数据成员/成员函数允许和基类的同名，不同的继承方式访问方法不同(暂不讨论)

§12. 继承和派生

4. 派生类的成员访问属性

4.1. 派生类和基类对成员的访问

普通类的访问规则：

类的成员函数访问类的成员 ：全部

作用域外访问类的成员 ：public

派生类与基类的访问规则：

基类的成员函数访问基类的成员 ：全部

基类的成员函数访问派生类的成员 ：不允许

作用域外通过基类访问基类的成员 ：public

作用域外通过基类访问派生类的成员 ：不允许

派生类的成员函数访问基类的成员 ：分析讨论

派生类的成员函数访问派生类的成员 ：全部

作用域外通过派生类访问基类的成员 ：分析讨论

作用域外通过派生类访问派生类的成员 ：public

§12. 继承和派生

4. 派生类的成员访问属性

4.1. 派生类和基类对成员的访问

派生类与基类的访问规则：

派生类的成员函数访问基类的成员　　：分析讨论

作用域外通过派生类访问基类的成员　：分析讨论

```
class A {
   private:
     int a;
     void f1();
   public:
     int b;
     void f2();
};

class B:private/public A {
   private:
     int c;
     void f3();
   public:
     int d;
     void f4();
};
```

基类A的成员函数
没有讨论价值

```
void A::f2()
{
    a=10;   ✓
    f1();   ✓
    b=15;   ✓
    c=10;   ×
    d=15;   ×
    f3();   ×
    f4();   ×
}
```

派生类B的成员函数
部分需要讨论

```
void B::f4()
{
    a=10;   讨论
    f1();   讨论
    b=15;   讨论
    f2();   讨论
    c=10;   ✓
    d=15;   ✓
    f3();   ✓
}
```

基类的作用域外，
没有讨论价值
派生类的作用域外，
部分需要讨论

```
int main()
{
   A a1;
   B b1;

   a1.a=10;  ×
   a1.f1();  ×
   a1.b=10;  ✓
   a1.f2();  ✓
   b1.a=10;  讨论
   b1.f1();  讨论
   b1.b=10;  讨论
   b1.f2();  讨论
   b1.c=10;  ×
   b1.f3();  ×
   b1.d=10;  ✓
   b1.f4();  ✓
}
```

§12. 继承和派生

4. 派生类的成员访问属性

4.2. 公有派生

形式： class 派生类名 : public 基类名 {

```
        private:
            ...
        public:
            ...
    };
```

★ 基类的private成员被继承，但不可访问

★ 基类的public成员被继承为派生类的public

4.3. 私有派生

形式： class 派生类名 : private 基类名 {

```
        private:
            ...
        public:
            ...
    };
```

★ 基类的private成员被继承，但不可访问

★ 基类的public成员被继承为派生类的private

不可访问：不能被直接访问，但仍然可以通过公有函数等形式进行间接访问

（下同）

§12. 继承和派生

4. 派生类的成员访问属性

4.1. 派生类和基类对成员的访问

派生类与基类的访问规则：

派生类的成员函数访问基类的成员　　：分析讨论

作用域外通过派生类访问基类的成员　：分析讨论

```
class A {
   private:
     int a;
     void f1();
   public:
     int b;
     void f2();
};

class B:private/public A {
   private:
     int c;
     void f3();
   public:
     int d;
     void f4();
};
```

B公有继承A

```
void B::f4()
{
    a=10;    讨论 ×
    f1();    讨论 ×
    b=15;    讨论 ✓
    f2();    讨论 ✓
    c=10;    ✓
    f3();    ✓
    d=15;    ✓
}
```

```
int main()
{
    A a1;
    B b1;

    a1.a=10;  ×
    a1.f1();  ×
    a1.b=10;  ✓
    a1.f2();  ✓
    b1.a=10;  讨论 ×
    b1.f1();  讨论 ×
    b1.b=10;  讨论 ✓
    b1.f2();  讨论 ✓
    b1.c=10;  ×
    b1.f3();  ×
    b1.d=10;  ✓
    b1.f4();  ✓
}
```

B私有继承A

```
void B::f4()
{
    a=10;    讨论 ×
    f1();    讨论 ×
    b=15;    讨论 ✓
    f2();    讨论 ✓
    c=10;    ✓
    f3();    ✓
    d=15;    ✓
}
```

```
int main()
{
    A a1;
    B b1;

    a1.a=10;  ×
    a1.f1();  ×
    a1.b=10;  ✓
    a1.f2();  ✓
    b1.a=10;  讨论 ×
    b1.f1();  讨论 ×
    b1.b=10;  讨论 ×
    b1.f2();  讨论 ×
    b1.c=10;  ×
    b1.f3();  ×
    b1.d=10;  ✓
    b1.f4();  ✓
}
```

§12. 继承和派生

4. 派生类的成员访问属性

4.4. 派生类的多级派生

```
class A {
   private:
     int a;
     void f1();
   public:
     int b;
     void f2();
};

class B:publicA {
   private:
     int c;
     void f3();
   public:
     int d;
     void f4();
};

class C:public B {
   private:
     int e;
     void f5();
   public:
     int f;
     void f6();
};
```

★ 基类分为直接基类和间接基类

C的直接基类是B，间接基类是A

B的直接基类是A

★ 派生类包含所有基类的数据成员(不一定可访问)

★ 基类的成员的被访问关系逐级确定

§12. 继承和派生

4. 派生类的成员访问属性

4.4. 派生类的多级派生

★ 基类的成员的被访问关系逐级确定

```
class A {
   private:
     int a;
     void f1();
   public:
     int b;
     void f2();
};
class B : 继承方式 A {
   private:
     int c;
     void f3();
   public:
     int d;
     void f4();
};
class C : 继承方式 B {
   private:
     int e;
     void f5();
   public:
     int f;
     void f6();
};
```

B公有继承A，C公有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ×	c1.a=10; ×
f1(); ×	c1.f1(); ×
b=10;	c1.b=10;
f2();	c1.f2();
c=10; ×	c1.c=10; ×
f3(); ×	c1.f3(); ×
d=10;	c1.d=10;
f4();	c1.f4();
e=10;	c1.e=10; ×
f5();	c1.f5(); ×
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B私有继承A，C公有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ×	c1.a=10; ×
f1(); ×	c1.f1(); ×
b=10; ×	c1.b=10; ×
f2(); ×	c1.f2(); ×
c=10; ×	c1.c=10; ×
f3(); ×	c1.f3(); ×
d=10;	c1.d=10;
f4();	c1.f4();
e=10;	c1.e=10; ×
f5();	c1.f5(); ×
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B公有继承A，C私有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ×	c1.a=10; ×
f1(); ×	c1.f1(); ×
b=10;	c1.b=10; ×
f2();	c1.f2(); ×
c=10; ×	c1.c=10; ×
f3(); ×	c1.f3(); ×
d=10;	c1.d=10; ×
f4();	c1.f4(); ×
e=10;	c1.e=10; ×
f5();	c1.f5(); ×
f=10;	c1.f=10;
f6();	c1.f6();
}	}

B私有继承A，C私有继承B

void C::f6()	int main()
{	{ C c1;
a=10; ×	c1.a=10; ×
f1(); ×	c1.f1(); ×
b=10; ×	c1.b=10; ×
f2(); ×	c1.f2(); ×
c=10; ×	c1.c=10; ×
f3(); ×	c1.f3(); ×
d=10;	c1.d=10; ×
f4();	c1.f4(); ×
e=10;	c1.e=10; ×
f5();	c1.f5(); ×
f=10;	c1.f=10;
f6();	c1.f6();
}	}

§12. 继承和派生

4. 派生类的成员访问属性

4.5. 保护段与保护继承

引入：在多级继承中，基类的private不可访问，public部分被私有继承则不可再传递，若公有继承则对于整个继承序列的外部均可访问，为了加强灵活性，引入保护段及保护继承

保护段的定义：

```
class 类名 {
    private:
      ...
    protected:
      ...
    public:
      ...
};
```

问题：(1) 父类的成员在派生类的继承序列中内部能够访问

(2) .....外部不能....

保护段的使用：

★ protected段的成员对外不可访问，对内可被任意访问，其成员函数可任意访问类的其它成员(等同于private属性)

★ protected段的成员继承后可被派生类访问(protected被private继承后当做派生类的private，protected被public继承后当做派生类的protected)

★ 若该类不被其它类所继承，则声明保护段无意义(不被继承的情况下与声明为private等价)

§12. 继承和派生

4. 派生类的成员访问属性

4.5. 保护段与保护继承

保护继承的定义：

```
class 派生类名 : protected 基类名 {
    private:
        ...
    protected:
        ...
    public:
        ...
};
```

保护继承的使用：

★ 基类的private成员被继承，但不可访问

★ 基类的protected/public成员被继承为派生类的protected

★ 若该类不被其它类所再次传递继承，则声明保护继承无意义(不传递继承的情况下与private继承等价)

§12. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

派生类的最终定义形式：

```
class 派生类名: private/protected/public 基类名{
    private:
        ...
    protected:
        ...
    public:
        ...
};
```

继承与访问关系表：

继承 / 基类	private	protected	public
private	继承，但不可访问	继承，但不可访问	继承，但不可访问
protected	private	protected	protected
public	private	protected	public

继承与派生类成员函数/派生类作用域外的访问关系表：

继承 / 基类	private		protected		public	
	成员	域外	成员	域外	成员	域外
private	×	×	×	×	×	×
protected	✓	×	✓	×	✓	×
public	✓	×	✓	×	✓	✓

§12. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

继承方式:private

B::fun()
{
a=10; ×
f1(); ×
b=10;
f2();
c=10;
f3();
d=10;
f4();
e=10;
f5();
f=10;
f6();
}
	int main()
	{ B b1;
	b1.a=10; ×
	b1.f1(); ×
	b1.b=10; ×
	b1.f2(); ×
	b1.c=10; ×
	b1.f3(); ×
	b1.d=10; ×
	b1.f4(); ×
	b1.e=10; ×
	b1.f5(); ×
	b1.f=10;
	b1.f6();
	}

继承方式protected

B::fun()
{
a=10; ×
f1(); ×
b=10;
f2();
c=10;
f3();
d=10;
f4();
e=10;
f5();
f=10;
f6();
}
	int main()
	{ B b1;
	b1.a=10; ×
	b1.f1(); ×
	b1.b=10; ×
	b1.f2(); ×
	b1.c=10; ×
	b1.f3(); ×
	b1.d=10; ×
	b1.f4(); ×
	b1.e=10; ×
	b1.f5(); ×
	b1.f=10;
	b1.f6();
	}

继承方式:public

B::fun()
{
a=10; ×
f1(); ×
b=10;
f2();
c=10;
f3();
d=10;
f4();
e=10;
f5();
f=10;
f6();
}
	int main()
	{ B b1;
	b1.a=10; ×
	b1.f1(); ×
	b1.b=10; ×
	b1.f2(); ×
	b1.c=10;
	b1.f3();
	b1.d=10; ×
	b1.f4(); ×
	b1.e=10; ×
	b1.f5(); ×
	b1.f=10;
	b1.f6();
	}

```
class A {
 priavte:
   int a;
   void f1();
 protected:
   int b;
   void f2();
 public:
   int c;
   void f3();
};

class B:继承方式 A{
 priavte:
   int d;
   void f4();
 protected:
   int e:
   void f5();
 public:
   int f;
   void f6();
   void fun();
};
```

§12. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

继承传递与成员函数/派生类作用域外的访问关系表：

（假设基本类A，类B继承A，类C继承B）（共9种情况）

列解释：B私 =>B私有继承A
成员=>在C的成员函数内访问
域外=>在C的作用域外访问
行解释：A:私=>A的私有成员在各种
方式下的可访问性

继承 / 基类	B私 C私		B私 C保		B私 C公		B保 C私		B保 C保		B保 C公		B公 C私		B公 C保		B公 C公	
	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外	成员	域外
A：私																		
A：保																		
A：公																		
B：私																		
B：保																		
B：公																		
C：私																		
C：保																		
C：公																		

请完成此表，填×或✓，表示不可访问/可访问

§12. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

继承传递与成员函数/派生类作用域外的访问关系表：

（假设基本类A，类B继承A，类C继承B）（共9种情况）

上页表格九种情况之一的实例表示，其它请自行理解

B私有继承A, C私有继承B

void C::fun()	int main()
{	{ C c;
A私 a1=10;× fa1();×	A私 c.a1=10;× c.fa1();×
A保 a2=10;× fa2();×	A保 c.a2=10;× c.fa2();×
A公 a3=10;× fa3();×	A公 c.a3=10;× c.fa3();×
B私 b1=10;× fb1();×	B私 c.b1=10;× c.fb1();×
B保 b2=10; fb2();	B保 c.b2=10;× c.fb2();×
B公 b3=10; fb3();	B公 c.b3=10;× c.fb3();×
C私 c1=10; fc1();	C私 c.c1=10;× c.fc1();×
C保 c2=10; fc2();	C保 c.c2=10;× c.fc2();×
C公 c3=10; fc3();	C公 c.c3=10; c.fc3();
}	}

```
class A {
    priavte:
        int a1;
        void fa1();
    protected:
        int a2;
        void fa2();
    public:
        int a3;
        void fa3();
};
class B : private A {
     priavte:
         int b1;
         void fb1();
     protected:
         int b2;
         void fb2();
     public:
         int b3;
         void fb3();
};
class C : public B {
    priavte:
        int c1;
        void fc1();
    protected:
       int c2;
       void fc2();
    public:
       int c3;
       void fc3();
       void fun();
};
```

§12. 继承和派生

4. 派生类的成员访问属性

4.6. 继承方式与访问属性

★ 多级派生总结

- 如果在多级派生时都采用私有继承方式，则传递一次后基类的所有成员均不可访问
- 如果在多级派生时都采用保护继承方式，则传递一次后基类的所有成员均只能在类的作用域内访问/不可访问
- 如果在多级派生时都采用公有继承方式，则传递多次后基类的公有及保护成员仍能访问且访问属性不变

=> 实际使用中：成员属性：protected / public
继承方式：public

§12. 继承和派生

4. 派生类的成员访问属性

4.7. 派生类与基类的成员同名

数据成员同名：

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

成员函数同名：

参数的个数、类型完全相同

★ 与数据成员同名的处理方式相同

参数的个数、类型不同

★ 与数据成员同名的处理方式相同

三种情况规则一样

派生类优先于基类，称为支配规则

§12. 继承和派生

4. 派生类的成员访问属性

4.7. 派生类与基类的成员同名

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

```
class A {
   public:
      int a;
   ...
};
class B:public A {
   public:
      short a;
      void fun();
   ...
};
```

```
void B::fun()
{
   a=10;
   A::a=15;
}
int main()
{
   B b1;
   b1.a=10;
   b1.A::a=15;
}
```

变量同名

★ 成员的可访问性符合条件
（内部任意，外部公有）

```
class A {
   public:
      void f1();
   ...
};
class B:public A {
   public:
      void f1();
      void fun();
   ...
};
```

```
void B::fun()
{
   f1();
   A::f1();
}
int main()
{
   B b1;
   b1.f1();
   b1.A::f1();
}
```

函数同名

★ 成员的可访问性符合条件

```
int a;
class A {
   public:
      int a;
   ...
};
class B:public A {
   public:
      short a;
      void fun();
   ...
};
```

讨论：4个不同级别变量同名

```
void B::fun()
{  int a;

   a=10;
   A::a=15;
   B::a=20;
   ::a=30;
}
```

★ 成员的可访问性符合条件

§12. 继承和派生

4. 派生类的成员访问属性

4.7. 派生类与基类的成员同名

★ 派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

函数同名，非重载，适用支配规则

```
class A {
   public:
      void f1(int x);
   ...
};
class B:public A {
   public:
      void f1();
      void fun();
   ...
};
```

```
void B::fun()
{
   f1();
   f1(10);      //编译错误
}
int main()
{
   B b1;
   b1.f1();
   b1.f1(10); //编译错误
}
```

★ 成员的可访问性符合条件

函数同名，非重载，适用支配规则

```
class A {
   public:
      void f1(int x);
   ...
};
class B:public A {
   public:
      void f1();
      void fun();
   ...
};
```

```
void B::fun()
{
   f1();
   A::f1(10);
}
int main()
{
   B b1;
   b1.f1();
   b1.A::f1(10);
}
```

★ 成员的可访问性符合条件

§12. 继承和派生

4. 派生类的成员访问属性

4.8. 基类对象不可访问部分的强制访问

问题引入：

任何继承方式派生类均不能访问基类的private

派生类的对象中包含了基类对象(包括private部分)

=> 不可访问，又占空间？

解决方法：

继承后，基类的私有部分不可被派生类直接访问，但是可以通过基类的可访问的成员函数进行间接访问

1、那句话编译错？

2、删除错误语句后的运行结果

```
#include <iostream>
using namespace std;

class A {
  private:
    int a;
  public:
    void set(int x) {
       a=x;
       }
    void show() {
       cout << a << endl;
       };
};
class B:public A {
  public:
    void fun() {
      a=10;
      set(10);
      }
};
int main()
{
   B b1;
   b1.set(15);
   b1.show();
   b1.fun();
   b1.show();
}
```

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.1. 简单的派生类构造函数

含义：在派生类的数据成员中不包含基类的对象

构造函数的实现方法：

★ 构造函数不能继承，同时派生中又含有基类的成员，需要能同时初始化基类和派生类的数据成员，因此在派生类的构造函数中激活基类的构造函数

形式：

```
派生类名(参数):基类名(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

初始化表方式

构造函数的调用顺序：

★ 先基类，再派生类

(在派生类的构造函数中先自动激活基类的构造函数)

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.1. 简单的派生类构造函数

```
class Time {
   protected:
     int hour, minute, second;
   public:
     Time(int h, int m, int s) {
        hour = h; minute = m; second = s;
        }
};
class Date:public Time {
   protected:
      int year, month, day;
   public:
      Date(int y, int mo, int d, int h, int m, int s):Time(h,m,s) {
        year  = y;
        month = mo;
        day   = d;;
        }
};
```

只有参数名，没有类型，从派生类的形参表中取

体内实现

```
class Date:public Time {
   ...
   public:
      Date(int y, int mo, int d, int h, int m, int s);
};
Date::Date(int y, int mo, int d, int h, int m, int s):Time(h,m,s)
{    year  = y;
     month = mo;
     day   = d;;
}
```

对体外实现，声明时不要，实现时给出即可

体外实现

```
int main()
{
    Date d1 (2021 5, 14, 15, 16, 17);
    ...
}
```

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.2. 含有子对象的派生类构造函数

含义：在派生类定义中包含了基类的对象

形式：

```
派生类名(参数):基类名(参数),子对象名(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

构造函数的调用顺序：

★ 先基类，次子对象，再派生类

★ 不再深入讨论

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.2. 含有子对象的派生类构造函数

```
class student {
    protected:
        int num;
        char name[10];
    public:
        student(int n,char *nam)
        {
            num = n;
            strcpy(name, nam);
        }
  ...
};

class stu:public student {
    protected:
        int  age;
        char addr[30];
        student monitor;
    public:
        stu(int n, char *nam, int n1, char *nam1,int a, char *ad): student(n,nam), monitor(n1,nam1)
        {
            age = a;
            strcpy(addr, ad);
        }
};
```

```
int main()
{
    stu s1(10001,"张三",10009,"李四",20,"上海");
    ...
}
```

班长

对于stu类的对象

stu s1;

所占空间为16+36+16=68字节，即双份student，其中1份继承，1份是monitor

也可体外实现(略)

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.3. 多层派生时的构造函数

形式：

```
派生类名(参数):直接基类名(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

★ 间接基类不需要出现在初始化表中

构造函数的调用顺序：

★ 按继承顺序依次调用，派生类放在最后

(派生类中先激活直接基类，直接基类再先激活其直接基类)

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.3. 多层派生时的构造函数

```
#include <iostream>
using namespace std;

class A {
   public:
      A(int x) { cout << "A" << x << endl; }
};
class B : public A {
   public:
      B(int x, int y):A(x) { cout << "B" << y << endl; }
};
class C : public B {
   public:
      C(int x, int y, int z):B(x, y){ cout << "C" << z << endl;}
};

int main()
{
    B b1(4,5);
    cout << endl;
    C c1(1,2,3);
}
```

A4
B5

A1
B2
C3

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.4. 派生类构造函数的特殊形式

★ 若派生类的初始化表中不出现基类，则系统自动调用基类的无参构造函数

（允许基类不定义构造函数，此时调用系统缺省的无参空体构造函数）

调用A的无参构造函数

```
#include <iostream>
using namespace std;

class A {
   public:
      A()       { cout << "A()"  << endl; }
      A(int x)  { cout << "A(x)" << endl; }
};
class B:public A {
   public:
      B()       { cout << "B()"  << endl; }
      B(int x)  { cout << "B(x)" << endl; }
};

int main()
{   B b1;
    cout << endl;
    B b2(100);
}
```

A()
B()

A()
B(x)

Microsoft Visual Studio 调试控制台

```
A()
B()

A()
B(x)
```

调用A的有参构造函数

```
#include <iostream>
using namespace std;

class A {
   public:
      A()       { cout << "A()"  << endl; }
      A(int x)  { cout << "A(x)" << endl; }
};
class B:public A {
   public:
      B(int x):A(x)          { cout << "B(x)"   << endl; }
      B(int x, int y):A(x)   { cout << "B(x,y)" << endl; };
};

int main()
{   B b1(10);
    cout << endl;
    B b2(100, 200);
}
```

A(x)
B(x)

A(x)
B(x, y)

Microsoft Visual Studio 调试控制台

```
A(x)
B(x)

A(x)
B(x,y)
```

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.4. 派生类构造函数的特殊形式

★ 若派生类的初始化表中不出现基类，则系统自动调用基类的无参构造函数

（允许基类不定义构造函数，此时调用系统缺省的无参空体构造函数）

```
#include <iostream>
using namespace std;

class A {
    public:
        A()      { cout << "A()"  << endl; }
        A(int x) { cout << "A(x)" << endl; }
};

class B:public A {
    public:
        B()                 { cout << "B()"    << endl; }
        B(int x)            { cout << "B(x1)"  << endl; }
        B(int x):A(x)       { cout << "B(x2)"  << endl; }
        B(int x, int y):A(x) { cout << "B(x,y)" << endl; }
};

int main()
{
    B b1;
    cout << endl;
    B b2(100);
    cout << endl;
    B b3(4, 5);
}
```

编译时会报函数重载错误

注释掉两个函数中的第1个，给出程序的运行结果

注释掉两个函数中的第2个，给出程序的运行结果

编译时这两句报函数重载错误，为什么？

§12. 继承和派生

5. 派生类的构造函数和析构函数

5.5. 派生类的析构函数

析构函数的调用：

在派生类对象出作用域时，自动调用派生类的析构函数，在其中再自动调用基类的析构函数

析构函数的调用顺序：

★ 先派生类，再基类(与构造函数的顺序相反)

析构函数的使用：

★ 派生类的数据成员无动态内存申请的情况下一般不需要定义派生类的析构函数

★ 若基类有动态内存申请，派生类无，则基类需要定义，派生类不需要

§12. 继承和派生

6. 多重继承

6.1. 多重继承的声明及使用

```
class 派生类名: private/protected/public 基类名1, ... , private/protected/public 基类名n {
    private:
        私有成员;
    protected:
        保护成员;
    public:
        公有成员;
};
```

★ 每个基类的基类存取限定符允许不同

★ 每个基类被继承后的可访问性由其基类存取限定符确定

★ 派生类继承所有基类的数据成员

★ 派生类继承所有基类除构造函数和析构函数外的所有成员函数

★ 派生类对象所占的空间 = 所有基类的数据成员之和 + 派生类的数据成员之和

★ 多重继承允许多层派生

```
class A {          class B:public A {   class C {
  private:           private:             private:
    int a1;            int b1;              int c1;
  protected:         protected:           protected:
    int a2;            int b2;              int c2;
  public:            public:              public:
    int a3;            int b3;              int c3;
};                 };                   };
class D:private B, public C {
  ...
};
```

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的形式：

派生类名(参数):基类名1(参数),...， 基类名n(参数)

{

函数体(一般是对派生类新增成员的初始化)

}

构造函数的使用：

★ 基类名出现的顺序无限制

★ 若调用基类的无参构造函数，则该基类可不出现在初始化参数表中

```
class A {
   public:
      A(int x) { ... }
};
class B {
   public:
      B(int y) { ... }
};
class C:public A, private B {
   public:
      C(int a, int b, int c):A(a), B(b) { ... }
};
```

B(b), A(a)

```
class A {
   public:
      A() { ... }
};
class B {
   public:
      B(int y) { ... }
};
class C:public A, private B {
   public:
      C(int a, int b, int c):B(b) { ... }
};
```

激活A的无参构造

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的使用：

★ 若是多层派生，只出现直接基类

```
class A {
   public:
      A(int x) { ... }
};
class B {
   public:
      B(int y) { ... }
};
class C:public B {
   public:
      C(int z):B(z) { ... };
};
class D:public A, private C {
   public:
      D(int a, int c, int d):A(a),C(c) { ... }
};
```

B不需要出现

★ 若派生中含有基类的实例对象(子对象)，则子对象也可以出现在初始化参数表中

```
class A {
   public:
      A(int x) { ... }
};
class B {
   public:
      B(int y) { ... }
};
class C:public A, private B {
   public:
      B bb;
      C(int a, int b, int b1):A(a), B(b), bb(b1) { ... }
};
```

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

```
与构造函数中基类出现的顺序无关
class D: ... {
   public:
      D():A(),B(),C() {...}
          B(),A(),C()
          C(),B(),A()
};
构造函数的调用顺序都是A、B、C、D
```

```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};

class B {
  public:
    B() {cout << "B()" << endl;}
};

class C {
  public:
    C() {cout << "C()" << endl;}
};

class D:public A, protected B, private C {
  public:
    D() {cout << "D()" << endl;}
};

int main()
{
    D d1;
}
```

构造函数的调用顺序：

A()
B()
C()
D()

两者一致

D对象的内存映像：

A的数据成员
B的数据成员
C的数据成员
D的数据成员

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    A() { a=1; }
};

class B {
  public:
    int b;
    B() { b=2;}
};

class C {
  public:
    int c;
    C() { c=3; }
};
```

```
class D:public A, protected B, private C {
  public:
    int d;
    D() { d=4;}
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;
    int *p = (int *)&d1;

    cout << *p     << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;

    return 0;
}
```

证明D对象的内存映象与D对象的构造函数调用顺序一致的测试程序

D对象的内存映像：

A的数据成员
B的数据成员
C的数据成员
D的数据成员

两者一致

```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};

class B {
  public:
    B() {cout << "B()" << endl;}
};

class C {
  public:
    C() {cout << "C()" << endl;}
};

class D:public A, protected B, private C {
  public:
    D() {cout << "D()" << endl;}
};

int main()
{
    D d1;
}
```

构造函数的调用顺序：

A()
B()
C()
D()

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

```
#include <iostream>
using namespace std;

class A {
  public:
    A() {cout << "A()" << endl;}
};
class B {
  public:
    B() {cout << "B()" << endl;}
};
class C:public B, public A {
  public:
    C() {cout << "C()" << endl;}
};
class D {
  public:
    D() {cout << "D()" << endl;}
};
class E:public D {
  public:
    E() {cout << "E()" << endl;}
};
class F:public E, public C {
  public:
    F() {cout << "F()" << endl;}
};

int main()
{
    F f1;
}
```

```
D()
E()
B()
A()
C()
F()
```

F对象的内存映像：

D的数据成员
E的数据成员
B的数据成员
A的数据成员
C的数据成员
F的数据成员

两者一致

§12. 继承和派生

6. 多重继承

6.2. 多重继承的派生类构造函数和析构函数

构造函数的调用顺序：

★ 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数

析构函数的使用：

★ 若数据成员没有动态内存申请，一般不需要定义

析构函数的调用顺序：

★ 先派生类的析构函数，再按声明顺序的反序依次调用基类的析构函数

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

```
int C::fun()
{   a=10;
    f1();
    A::a=15;
    A::f1();
}
```

```
int main()
{   C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
}
```

★ 以不产生二义性为基本准则

下面测试程序编译通过，
则证明上面的例子是正确的

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    void f1() { return; }
};
class B {
  public:
    int b;
};
class C:public A, public B {
  public:
    int a;
    void f1() { return; }
    void fun();
};
```

```
void C::fun()
{
    a = 10;
    f1();
    A::a = 15;
    A::f1();
}
int main()
{
    C c1;

    c1.a = 10;
    c1.f1();
    c1.A::a = 15;
    c1.A::f1();

    return 0;
}
```

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

```
int C::fun()
{  A::a=10;
   A::f1();
   B::a=15;
   B::f1();
}
```

```
int main()
{  C c1;
   c1.A::a=10;
   c1.A::f1();
   c1.B::a=15;
   c1.B::f1();
}
```

可自行构造测试程序来验证

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

```
int C::fun()
{   a=10;
    f1();
    A::a=15;
    A::f1();
    B::a=20;
    B::f1();
}
```

```
int main()
{   C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
    c1.B::a=20;
    c1.B::f1();
}
```

可自行构造测试程序来验证

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

```
int D::fun()
{   a=10;
    f1();
    A::a=15;可以B
    A::f1();可以B
    C::a=20;
    C::f1();
}
```

```
int main()
{   D d1;
    d1.a=10;
    d1.f1();
    d1.A::a=15;可以B
    d1.A::f1();可以B
    d1.C::a=20;
    d1.C::f1();
}
```

可自行构造测试程序来验证

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

例：写出右侧两个程序的运行结果（左侧代码相同）

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    void f1() { return; }
};

class B:public A {
  public:
    int b;
};

class C {
 public:
    long a;
    int f1() { return 0; }
};

class D:public B, public C {
  public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

```
void D::fun()
{
    a = 10;
    f1();
    A::a = 15; // B::a = 15;
    A::f1();   // B::f1();
    C::a = 20;
    C::f1();
    int *p = (int *)this;
    cout << sizeof(*this) <<endl;
    cout << *p     << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
}
int main()
{
    D d1;
    d1.fun();
    return 0;
}
```

?

```
void D::fun()
{
    return;
}
int main()
{
    D d1;
    d1.a = 10;
    d1.f1();
    d1.A::a = 15; // B::a = 15;
    d1.A::f1();   // B::f1();
    d1.C::a = 20;
    d1.C::f1();
    int *p = (int *)&d1;
    cout << sizeof(d1) <<endl;
    cout << *p     << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

?

§12. 继承和派生

6. 多重继承

6.3. 多重继承引起的二义性问题(成员同名)

某一个基类与派生类的成员同名：

按单继承的方式进行处理

(支配规则:派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

★ 以不产生二义性为基本准则

下面这两个例子中：A类中的成员在间接基类D中可能多次继承而产生重复问题

问1：在派生类中是否有重复的间接基类数据成员？

问2：如何处理多重继承引起的二义性问题(成员同名)

§12. 继承和派生

6. 多重继承

6.4. 虚基类

引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？ -> 答：有

问2：如何处理多重继承引起的二义性问题(成员同名)

D对象的内存映象(两份A)

针对上面的继承序列，运行右侧的测试程序

1、先通过sizeof(d1)的结果确定A中的int a 在d1对象中有2份

2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
#include <iostream>
using namespace std;
static int x=0;
class A {
  public:
    int a;
    A() { a = ++x;
          cout << "A(" << a << ")" << endl; }
};

class B :public A {
  public:
    int b;
    B() { b = 20;
          cout << "B()" << endl; }
};

class C :public A {
  public:
    int c;
    C() { c = 30;
          cout << "C()" << endl; }
};
```

```
class D :public B, public C {
  public:
    int d;
    D()
    {    d = 40;
         cout << "D()" << endl;
    }
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    cout << *(p+4) << endl;

    return 0;
}
```

§12. 继承和派生

6. 多重继承

6.4. 虚基类

引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？

问2：如何处理多重继承引起的二义性问题(成员同名)

答：通过可区分的类的作用域符来进行区分

```
int D::fun()
{
   B::a=10;
   B::f1();
   C::a=15;
   C::f1();
   return 0;
}  //不能/建议不用A::
```

```
int main()
{
   D d1;
   d1.B::a=10;
   d1.B::f1();
   d1.C::a=15;
   d1.C::f1();
   return 0;
}  //不能/建议不用A::
```

针对上面的继承序列，运行右侧的测试程序

1、先通过sizeof(d1)的结果确定A中的int a 在d1对象中有2份

2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
#include <iostream>
using namespace std;

static int x=0;
class A {
  public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl;}
};

class B :public A {
  public:
    int b;
};

class C :public A {
  public:
    int c;
};

class D :public B, public C {
  public:
    int d;
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;
    d1.A::f1();
    d1.B::f1();
    d1.C::f1();
    d1.f1();
    return 0;
}
```

四编译器分别编译哪些语句报错？

§12. 继承和派生

6. 多重继承

6.4. 虚基类

引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？ -> 答：有

问2：如何处理多重继承引起的二义性问题(成员同名)

D对象的内存映象(两份A)

A的数据成员	B
B的数据成员	
A的数据成员	C
D的数据成员	D

针对上面的继承序列，运行右侧的测试程序

1、先通过sizeof(d1)的结果确定A中的int a 在d1对象中有2份

2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
#include <iostream>
using namespace std;

static int x = 0;

class A {
  public:
    int a;
    A()
    { a = ++x;
      cout << "A(" << a << ")" << endl;
    }
};

class B :public A {
  public:
    int b;
    B()
    { b = 20;
      cout << "B()" << endl;
    }
};

class D :public B, public A {
  public:
    int d;
    D()
    {    d = 40;
         cout << "D()" << endl;
    }
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;

    return 0;
}
```

[Warning] direct base 'A' inaccessible in 'D' due to ambiguity

warning C4584: "D" : 基类"A"已是"B"的基类

Microsoft

```
A(1)
B()
A(2)
D()
16
1
20
2
40
```

§12. 继承和派生

6. 多重继承

6.4. 虚基类

引入：间接基类被多次继承而产生的重复问题

问1：在派生类中是否有重复的间接基类数据成员？

问2：如何处理多重继承引起的二义性问题(成员同名)

答：通过可区分的类的作用域符来进行区分，部分无法区分

```
int D::fun()
{
    A::a=10;  ×/不建议
    A::f1();  ×/不建议
    B::a=15;
    B::f1();
    a=15;     ×
    f1();     ×
}
```

```
int main()
{
    D d1;
    d1.A::a=10;×/不建议
    d1.A::f1();×/不建议
    d1.B::a=15;
    d1.B::f1();
    d1.a=15;    ×
    d1.f1();    ×
}
```

针对上面的继承序列，运行右侧的测试程序

1、先通过sizeof(d1)的结果确定A中的int a 在d1对象中有2份

2、再通过指针访问技巧(红色)确定d1中各数据成员的排列顺序(2个int a在哪)

```
#include <iostream>
using namespace std;

static int x=0;

class A {
  public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl;}
};

class B :public A {
  public:
    int b;
};

class D :public B, public A {
  public:
    int d;
};

int main()
{
    D d1;
    cout << sizeof(d1) << endl;

    d1.A::f1();
    d1.B::f1();
    d1.f1();

    return 0;
}
```

四编译器分别编译哪些语句报错？

§12. 继承和派生

6. 多重继承

6.4. 虚基类

含义：针对某个间接基类被多次继承而产生的多个无名对象，从而导致派生类中有多份相同的数据成员拷贝的情况，引入虚基类，使相同基类只保留一份数据成员

=> 目前D中有两份A，引入虚基类后，D中只有一份A

D对象内存映像(两份A)

声明：

```
class 派生类名: virtual 存取限定符 基类名 {
    ...
};
```

§12. 继承和派生

6. 多重继承

6.4. 虚基类

```
#include <iostream>
using namespace std;

class A {
  public:
    A() {cout << "A()" << endl;}
};
class B:public A {
  public:
    B() {cout << "B()" << endl;}
};
class C:public A {
  public:
    C() {cout << "C()" << endl;}
};
class D:public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};

int main()
{
    E e1;
}
```

```
A() ：第1次
B()
A() ：第2次
C()
A() ：第3次
D()
E()
```

若定义：E e1 ，则E中只有三份A的拷贝

```
#include <iostream>
using namespace std;

class A {
  public:
    A() {cout << "A()" << endl;}
};
class B:virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C:virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
class D:virtual public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};

int main()
{
    E e1;
}
```

```
A()
B()
C()
D()
E()
```

若定义：E e1 ，则E中只有一份A的拷贝

§12. 继承和派生

6. 多重继承

6.4. 虚基类

声明：

```
class 派生类名:virtual 存取限定符 基类名 {
    ...
    };
```

★ 所有继承该基类的直接派生类都应声明为虚基类才能保证只有一份数据拷贝

```
#include <iostream>
using namespace std;

class A {
  public:
    A() {cout << "A()" << endl;}
};
class B:virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C:virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
class D:public A {  //此处无virtual
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};

int main()
{
    E e1;
}
```

```
A()
B()
C()
A()
D()
E()
```

若定义：E e1 ，则e1中有两份A的拷贝

§12. 继承和派生

6. 多重继承

6.4. 虚基类

虚基类的构造函数：

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出，则缺省调用无参构造

```
class A {
  public:
    A()       { cout << "A()"          << endl; }  //无参构造
    A(int x)  { cout << "A(" << x << ")" << endl; }  //一参构造
};
class B:virtual public A {
  public:
    B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
  public:
    C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
  public:
    D(int x, int y, int z):A(x),B(y),C(z) { cout << "D()" << endl;}
};

int main()
{
   D d1(1,2,3);
   cout << endl;
   C c1(4);
   cout << endl;
   B b1(5);
}
```

```
A(1)
B()
C()
D()

A(4)
C()

A()
B()
```

如果通过D激活，应该A(1)
如果通过B激活，应该A()
如果通过C激活，应该A(3)

§12. 继承和派生

6. 多重继承

6.4. 虚基类

虚基类的构造函数：

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出，则缺省调用无参构造

```
class A {
  public:
    A()       { cout << "A()"          << endl; }  //无参构造
    A(int x)  { cout << "A(" << x << ")" << endl; }  //一参构造
};
class B:virtual public A {
  public:
    B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
  public:
    C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
  public:
    D(int x, int y, int z):B(y),C(z) { cout << "D()" << endl;}
};

int main()
{
   D d1(1,2,3);
   cout << endl;
   C c1(4);
   cout << endl;
   B b1(5);
}
```

此处无A(x)，则调A的无参构造

```
A()
B()
C()
D()

A(4)
C()

A()
B()
```

§12. 继承和派生

6. 多重继承

6.4. 虚基类

虚基类的构造函数：

★ 由派生类直接负责给出间接虚基类的初始化参数若派生类中未给出，则缺省调用无参构造

★ 调用时，由派生类直接激活间接虚基类的构造函数，其直接基类不再自动激活虚基类的构造

```
class A {
  public:
    A()       { cout << "A()"            << endl; }  //无参构造
    A(int x) { cout << "A(" << x << ")" << endl; }  //一参构造
};
class B:virtual public A {
  public:
    B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
  public:
    C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
  public:
    D(int x, int y, int z):A(x),B(y),C(z) { cout << "D()" << endl;}
};
```

假设 B b1(1) / C c1(1):
(直接基类是虚基类的情况)

与非虚基类一样，b1/c1生成时会自动激活A的无参/有参构造函数

假设 D d1(1,2,3):
① d1对象生成时会自动激活A、B、C的构造函数(顺序不讨论)，再调用D的构造函数
② B/C的构造函数被调用时，不再自动激活A的构造函数

★ 包含虚基类的构造函数及析构函数的调用顺序与普通的继承不同，不再讨论
(可自行查阅有关资料)

§12. 继承和派生

6. 多重继承

6.4. 虚基类

不再讨论的内容(只给出现象，原因自行研究)：

★ 包含虚基类的派生类对象所占空间与普通继承不同，不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同，不再讨论

无虚基类

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    A() {cout << "A()" << endl; }
};
class B {
  public:
    int b;
    B() {cout << "B()" << endl; }
};
class C : public B {
  public:
    int c;
    C() {cout << "C()" << endl; }
};
class D : public B {
  public:
    int d;
    D() {cout << "D()" << endl; }
};
class E : public B {
  public:
    int e;
    E() {cout << "E()" << endl; }
};
class F : public A, public C,
            public D, public E {
  public:
    int f;
    F() {cout << "F()" << endl; }
};
int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

无虚基类

§12. 继承和派生

6. 多重继承

6.4. 虚基类

不再讨论的内容(只给出现象，原因自行研究)：

★ 包含虚基类的派生类对象所占空间与普通继承不同，不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同，不再讨论

部分虚基类

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    A() {cout << "A()" << endl; }
};
class B {
  public:
    int b;
    B() {cout << "B()" << endl; }
};
class C : virtual public B {
  public:
    int c;
    C() {cout << "C()" << endl; }
};
class D : virtual public B {
  public:
    int d;
    D() {cout << "D()" << endl; }
};
class E : public B {
  public:
    int e;
    E() {cout << "E()" << endl; }
};
class F : public A, public C,
            public D, public E {
  public:
    int f;
    F() {cout << "F()" << endl; }
};
int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

深度思考：

1、如何理解有virtual后sizeof的大小多了4字节？4字节存了什么？

2、如何理解总大小的变化(32=>36)

无虚基类

部分虚基类

§12. 继承和派生

6. 多重继承

6.4. 虚基类

不再讨论的内容(只给出现象，原因自行研究)：

★ 包含虚基类的派生类对象所占空间与普通继承不同，不再讨论

★ 包含虚基类的派生类构造函数及析构函数的调用顺序与普通继承不同，不再讨论

全部虚基类

```
#include <iostream>
using namespace std;

class A {
  public:
    int a;
    A() {cout << "A()" << endl; }
};
class B {
  public:
    int b;
    B() {cout << "B()" << endl; }
};
class C : virtual public B {
  public:
    int c;
    C() {cout << "C()" << endl; }
};
class D : virtual public B {
  public:
    int d;
    D() {cout << "D()" << endl; }
};
class E : virtual public B {
  public:
    int e;
    E() {cout << "E()" << endl; }
};
class F : public A, public C,
            public D, public E {
  public:
    int f;
    F() {cout << "F()" << endl; }
};
int main()
{
    cout << sizeof(A) << endl;
    cout << sizeof(B) << endl;
    cout << sizeof(C) << endl;
    cout << sizeof(D) << endl;
    cout << sizeof(E) << endl;
    cout << sizeof(F) << endl;

    F f1;
    return 0;
}
```

深度思考：

1、如何理解有virtual后sizeof的大小多了4字节？4字节存了什么？

2、如何理解总大小的变化(32=>36=>36)

无虚基类

```
Microsoft
4
4
8
8
8
32
A()
B()
C()
B()
D()
B()
E()
F()
```

部分虚基类

```
Microsoft
4
4
12
12
8
36
B()
A()
C()
D()
B()
E()
F()
```

全部虚基类

```
Microsoft
4
4
12
12
12
36
B()
A()
C()
D()
E()
F()
```

§12. 继承和派生

7. 基类与派生类的转换

例：

```
#include <iostream>
using namespace std;

class A {
    public:
        int a;
};

class B:public A {
    public:
        int b;
};

int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;   10
    a1 = b1;
    cout << a1.a << endl;   15
}
```

问题：

1、运算符重载：系统会缺省对=进行重载，但要求两侧都是相同类对象，即 A=A / B=B

2、既未定义 A=B 的重载也未定义B转A的转换构造函数，为什么此句不错？

3、换成 B=A 会怎样？即：b1=a1;

```
#include <iostream>
using namespace std;

class A {
    public:
        int a;
};

class B:public A {
    public:
        int b;
};

int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;
    b1 = a1;   //编译错!!!
    cout << a1.a << endl;
}
```

```
(21,12): error C2679: 二元“=”: 没有找到接受“A”类型的右操作数的运算符(或没有可接受的转换)
(12,1): message : 可能是“B &B::operator =(B &&)”
(12,1): message : 或       “B &B::operator =(const B &)”
(21,12): message : 尝试匹配参数列表“(B, A)”时
```

§12. 继承和派生

7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃

§12. 继承和派生

7. 基类与派生类的转换

例：单继承下的赋值兼容规则

```
#include <iostream>
using namespace std;

class A {
    public:
        int a;
};

class B:public A {
    public:
        int b;
};

int main()
{  A a1;
   B b1;
   a1.a = 10;
   b1.a = 15;
   cout << a1.a << endl;   10
   a1 = b1;                赋值兼容规则
   cout << a1.a << endl;   15
}
```

§12. 继承和派生

7. 基类与派生类的转换

例：多继承下的赋值兼容规则

```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
};
class B {
  public:
    int b;
};
class C:public A, public B {
  public:
    int c;
};
int main()
{  A a1;   B b1;   C c1;
   a1.a = 10;
   b1.b = 20;
   c1.a = 15;
   c1.b = 25;
```

```
   cout << a1.a << endl;    10
   a1 = c1;                 赋值兼容规则
   cout << a1.a << endl;    15
```

```
   cout << b1.b << endl;    20
   b1 = c1;                 赋值兼容规则
   cout << b1.b << endl;    25
}
```

§ 12. 继承和派生

7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃

- 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数
- 当基类中包含动态申请内存时，赋值兼容规则可能出错（请参考=重载及复制构造函数）

```cpp
#include <iostream>
using namespace std;

class A {
  public:
    int a;
};
class B:public A {
  public:
    int b;
};
int main()
{  B b1;
   b1.a = 15;
   A a1(b1);    //复制构造
   A a2;
   a2 = b1;     //=重载
   cout << a1.a << endl;    //15
   cout << a2.a << endl;    //15
}
```

1、由赋值兼容规则可得到输出为15
2、如果希望输出为b1.a的2倍（30），则赋值兼容规则不适用，怎么办？

```cpp
class B; //提前声明
class A {
  public:
    int a;
    A() {}   //无参空体
    A(B &b);//不能体内实现
    A& operator=(B &b);//不能体内实现
};
class B:public A {
  public:
    int b;
};
A::A(B &b) //一参构造
{   a = b.a*2;
}
A& A::operator=(B &b) //=重载
{   a = b.a*2;
    return (*this);
}
int main()
{  B b1;
   b1.a = 15;
   A a1(b1), a2;    //一参构造，无参构造
   a2 = b1;     //=重载
   cout << a1.a << endl;   //30
   cout << a2.a << endl;   //30
}
```

定义A的构造函数，参数为B &
定义A的=重载，右值为B &

```cpp
#include <iostream>
using namespace std;
class A {
  private:
    char *s;
  public:
    int a;
    A() { s = new char[20]; }
    ~A() {
        cout << "A析构" << endl;
        delete s;
        }
};
class B:public A {
  public:
    int b;
    ~B() { cout << "B析构" << endl; }
};
int main()
{  B b1;
   b1.a = 15;
   A a1(b1);    //复制构造
}
```

运行出错，具体请参考=重载及复制构造函数
带动态申请时的错误分析(一定要自行弄懂!!!)

§ 12. 继承和派生

7. 基类与派生类的转换

赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象

★ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问，因此不可用

★ 使用时，将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃

- 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数
- 当基类中包含动态申请内存时，赋值兼容规则可能出错（请参考=重载及赋值构造函数）

★ 派生类对象可初始化基类的对象或引用

★ 派生类对象可出现在函数参数/返回值为基类的地方

★ 派生类对象可赋值给基类

★ 派生类对象的指针可出现在基类指针出现的位置

下面这两个例子，编译全部正确

```
class A { ... };
class B:public A { ... };
int main()
{
   B b1, *pb = &b1;
   A a1(b1);    //需要A对象，用B对象替代
   A &a2 = b1; //需要A对象，用B对象替代
   A *pa;

   a1 = b1;     //需要A对象，用B对象替代
   pa = pb;     //需要A对象的地址，用B对象的地址替代
   pa = &b1;    //需要A对象的地址，用B对象的地址替代

   return 0;
}
```

```
class A { ... };
class B:public A { ... };
void f1(A  a1) { ... }  //形参为A对象
void f2(A &a1) { ... }  //形参为A对象的引用
void f3(A *pa) { ... }  //形参为A对象的指针
A f4()
{ static B b1;
  return b1; //返回值为A对象
}
int main()
{  B b1;
   f1(b1);  //B对象出现在要求A对象的位置
   f2(b1);  //B对象出现在要求A对象引用的位置
   f3(&b1); //B对象出现在要求A对象指针的位置
}
```

§12. 继承和派生

8. 继承与组合

类的组合：一个类的对象是另一个类的数据成员

（两个类相互独立，无继承关系）

```
class Date {
   private:
      int year, month, day;
   ...
};

class student {
   private:
      Date birthday;
};
```

★ 请注意两个类声明的顺序

★ 可能有提前声明

§13. 多态性与虚函数

1. 多态性的基本概念

多态的广义定义：一个事物有多种形态

面向对象方法学的定义：不同的对象能接受同一消息，并自主地对消息进行处理

★ 程序设计语言+类 = 基于对象的程序设计语言

★ 程序设计语言+类+多态 = 面向对象的程序设计语言

C++程序设计中多态的含义：同一作用域内有多个不同功能的函数可以具有相同的函数名

分类：

静态多态：在程序编译时已确定调用函数(函数重载)

动态多态：在程序运行时才确定操作的对象(虚函数)

§13. 多态性与虚函数

2. 多态性的引入

例：假设存在如下的类继承层次：

	数据成员	成员函数	
Point	坐标 x 坐标 y	构造 setPoint(x,y)设置x,y getX,getY，取x,y的值 << 重载	
Circle	半径 radius	构造 setRadius(r) 设置半径 getRadius() 取半径 area() 求面积 << 重载	设圆心、取圆心的函数继承
Cylinder	高度 height	构造 setHeight(h) 设置高度 getHeight() 取高度 area() 求表面积 volumn() 求体积 << 重载	设圆心、取圆心设半径、取半径的函数继承

§13. 多态性与虚函数

2. 多态性的引入

例：Point类的定义、实现及测试函数

```
#include <iostream>
using namespace std;

const double pi=3.14159;

class Point {
    protected: //希望被子类继承但外部无法访问
       double x, y;
    public:
        Point(double a=0, double b=0);
        void setPoint(double a, double b);
        double getX() const;//常成员函数,不能修改值
        double getY() const;//常成员函数,不能修改值
        friend ostream &operator<< (ostream &out, const Point &p);
};

Point::Point(double a, double b) //初始化时用
{
    x=a;
    y=b;
}

void Point::setPoint(double a, double b)  //执行中用
{
    x=a;
    y=b;
}
```

```
double Point::getX() const
{
    return x;
}

double Point::getY() const
{
    return y;
}

ostream &operator<< (ostream &out, const Point &p)
{
    out << "[" << p.x << "," << p.y << "]" << endl;
    return out;
}

int main()
{
    Point p(3.5, 6.4);
    cout << "p:[x=" << p.getX() << ",y=" << p.getY() << "]" << endl;

    p.setPoint(8.5, 6.8);
    cout << "p(new):" << p << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
p:[x=3.5,y=6.4]
p(new):[8.5,6.8]
```

§13. 多态性与虚函数

2. 多态性的引入

例：Circle类的定义、实现及测试函数

```
#include <iostream>
using namespace std;

... class Point  的定义与实现 ...

class Circle : public Point {
    protected:  //准备被volumn继承
        double radius;
    public:
        Circle(double x=0, double y=0, double r=0);
        void  setRadius(double r);
        double getRadius() const;
        double area() const;
        friend ostream &operator<<(ostream &out, const Circle &c);
};

Circle::Circle(double x, double y, double r) : Point(x,y),radius(r)
{  //初始化表形式，函数体为即可
}

void Circle::setRadius(double r)
{
    radius = r;
}

double Circle::getRadius() const
{
    return radius;
}
```

Microsoft Visual Studio 调试控制台

```
c:圆心[x=3.5,y=6.4]
圆半径=5.2
圆面积=84.9486
point:[3.5,6.4]

c(new):圆心[x=5,y=5]
圆半径=7.5
圆面积=176.714
point(new):[5,5]
```

```
double Circle::area() const
{
    return pi * radius * radius;
}

ostream &operator<<(ostream &out, const Circle &c)
{
    out << "圆心[x=" << c.x << ",y=" <<c.y << "]" << endl
        << "圆半径=" << c.radius << endl
        << "圆面积=" << c.area() << endl;
    return out;
}

int main()
{
    Circle c(3.5, 6.4, 5.2);
    cout << "c:圆心[x=" << c.getX() << ",y=" << c.getY() << "]" << endl
         << "圆半径=" << c.getRadius() << endl
         << "圆面积=" << c.area() << endl;
    Point *p=&c;  //派生类出现在需要基类的位置
    cout << "point:" << *p << endl;  //输出Point的信息+多空一行

    c.setRadius(7.5);
    c.setPoint(5.0, 5.0);
    cout << "c(new):" << c; //<<重载最后已有endl
    Point &pRef=c;  //派生类出现在需要基类的位置
    cout << "point(new):" << pRef << endl;  //输出Point的信息
    return 0;
}
```

§13. 多态性与虚函数

2. 多态性的引入

例：Cylinder类的定义、实现及测试函数

```
#include <iostream>
using namespace std;

... class Point  的定义与实现 ...
... class Circle 的定义与实现 ...

class Cylinder : public Circle {
    protected:
        double height;
    public:
        Cylinder(double x=0, double y=0, double r=0, double h=0);
        void setHeight(double h);
        double getHeight() const;
        double area() const;
        double volume() const;
        friend ostream &operator<<(ostream &out, const Cylinder &cy);
};
Cylinder::Cylinder(double x, double y, double r,double h) :
                                        Circle(x, y, r), height(h)
{   //初始化表形式，空函数体即可
}
void  Cylinder::setHeight(double h)
{   height = h;
}
double Cylinder::getHeight() const
{   return height;
}
double Cylinder::area() const
{   return 2 * Circle::area() + 2 * pi * radius * height;
}
double Cylinder::volume() const
{   return Circle::area() * height;
}
```

```
ostream &operator<< (ostream &out, const Cylinder &cy)
{   out << "柱圆心[x=" << cy.x << ",y=" << cy.y << "]" << endl
        << "柱半径=" << cy.radius << endl
        << "柱高=" << cy.height << endl
        << "柱表面积=" << cy.area() << endl
        << "柱体积=" << cy.volume() << endl;
    return out;
}
int main()
{  Cylinder cy1(3.5, 6.4, 5.2, 10.0);
   cout << "cy1:柱圆心[x=" << cy1.getX() <<",y=" << cy1.getY() << "]"<<endl
        << "柱半径=" << cy1.getRadius() << endl
        << "柱高度=" << cy1.getHeight() << endl
        << "柱表面积=" << cy1.area() << endl
        << "柱体积=" << cy1.volume() << endl;

   Point *p = &cy1;  //赋值兼容规则
   cout << "point:" << *p;
   Circle *pc = &cy1; //赋值兼容规则
   cout << "circle:" << *pc;
   cout << endl; //多空一行

   cy1.setHeight(15.0);
   cy1.setRadius(7.5);
   cy1.setPoint(5.0, 5.0);
   cout << "cy1(new):" << cy1;
   Point &pRef = cy1;  //赋值兼容规则
   cout << "point(new):" << pRef;
   Circle &cRef = cy1; //赋值兼容规则
   cout << "circle(new):" << cRef;

   return 0;
}
```

Microsoft Visual Studio 调试控制台

```
cy1:柱圆心[x=3.5,y=6.4]
柱半径=5.2
柱高度=10
柱表面积=496.623
柱体积=849.486
point:[3.5,6.4]
circle:圆心[x=3.5,y=6.4]
圆半径=5.2
圆面积=84.9486

cy1(new):柱圆心[x=5,y=5]
柱半径=7.5
柱高=15
柱表面积=1060.29
柱体积=2650.72
point(new):[5,5]
circle(new):圆心[x=5,y=5]
圆半径=7.5
圆面积=176.714
```

§13. 多态性与虚函数

2. 多态性的引入

在 Point -> Circle -> Cylinder 的继承层次中：

★ setPoint(),getX(),getY(),setRadius(),getRadius() 是继承

★ area()是支配规则

★ <<运算符是重载

★ 是静态多态，在编译时已确定应调用哪个函数

	数据成员	成员函数	
Point	坐标 x 坐标 y	构造 setPoint(x,y)设置x,y getX,getY，取x,y的值 << 重载	
Circle	半径 radius	构造 setRadius(r) 设置半径 getRadius() 取半径 area() 求面积 << 重载	设圆心、取圆心的函数继承
Cylinder	高度 height	构造 setHeight(h) 设置高度 getHeight() 取高度 area() 求表面积 volumn() 求体积 << 重载	设圆心、取圆心设半径、取半径的函数继承

§13. 多态性与虚函数

3. 虚函数

3.1. 多个同名函数的使用

参数个数、参数类型完全相同：

同一类　　：不允许

不同独立类　：允许，被不同对象调用而区分

类的继承层次：允许，支配规则，用类作用域符区分

参数个数、参数类型不完全相同：

同一类　　：允许，重载

不同独立类　：允许，被不同对象调用而区分

类的继承层次：允许，支配规则，用类作用域符区分（再次强调，不是重载）

§13. 多态性与虚函数

3. 虚函数

3.2. 虚函数的引入

在类的继承层次中，对于参数个数、参数类型完全相同的同名函数，采用支配规则进行访问，要通过不同的对象来访问不同的同名函数(调用形式不同)

```
int main()
{
    Circle c(5.0, 5.0, 7.5);
    cout << c.area() << endl;    //圆面积

    Cylinder cyl(5.0, 5.0, 7.5, 10.0);
    cout << cyl.area() << endl;          //圆柱体表面积
    cout << cyl.Circle::area() << endl; //圆柱体底面积
}
```

```
Micros
176.714
824.667
176.714
```

调用形式不同
圆对象.area()
柱对象.area()
柱对象.圆::area()

通过赋值兼容规则，可使调用形式相同，但只能访问派生类中的基类部分

```
int main()
{
    Circle   c1 (3.5, 6.3, 5.0);
    Cylinder cyl(3.5, 6.3, 5.0, 10.0);
    Circle  *p;
    p = &c1;
    cout << p->area() << endl; //调用形式相同

    p = &cyl;
    cout << p->area() << endl; //调用形式相同

    return 0;
}
```

```
Micros
78.5397
78.5397
```

调用形式相同
圆对象.area()
柱对象.无名圆.area()

为了能采用同一调用形式来访问类继承层次中的同名函数，引入虚函数(调用形式相同)

右例，期望：
78.5397
471.238
但目前做不到

不满足期望的原因，采用了静态多态，在编译时已确定访问基类的area函数

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

改动Circle的定义

```
class Circle : public Point {
    protected:  //准备被volumn继承
        double radius;
    public:
        Circle(double x=0, double y=0, double r=0);
        void  setRadius(double r);
        double getRadius() const;
        virtual double area() const;
        friend ostream &operator<<(ostream &out, const Circle &c);
};
```

改动Circle的定义
但右侧main同上页，无任何变化

```
int main()
{
   Circle   c1 (3.5, 6.3, 5.0);
   Cylinder cyl(3.5, 6.3, 5.0, 10.0);
   Circle  *p;
   p = &c1;
   cout << p->area() << endl; //调用形式相同

   p = &cyl;
   cout << p->area() << endl; //调用形式相同

   return 0;
}
```

```
Micros
78.5397
471.238
```

调用形式相同
圆对象.area()
柱对象.area()

满足期望的原因，采用了动态多态，在运行时
才确定访问哪个类的area函数

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

```
#include <iostream>
usning namespace std;

class A {
  public:
    virtual void display() {
       cout << "A::display()" << endl;
       }
};

class B:public A {
  public:
    void display() {
       cout << "B::display()" << endl;
       }
};

class C:public B {
  public:
    void display() {
       cout << "C::display()" << endl;
       }
};
```

```
//A类不加 virtual 的执行结果
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();    A::
    b.display();    B::
    c.display();    C::
    ra.display();   A::
    rb.display();   B::
    rc.display();   C::
    pa->display();  A::
    pb->display();  B::
    pc->display();  C::
    p=&a;
    p->display();   A::
    p=&b;
    p->display();   A::
    p=&c;
    p->display();   A::
}
```

```
//A类加 virtual 的执行结果
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();    A::
    b.display();    B::
    c.display();    C::
    ra.display();   A::
    rb.display();   B::
    rc.display();   C::
    pa->display();  A::
    pb->display();  B::
    pc->display();  C::
    p=&a;
    p->display();   A::
    p=&b;
    p->display();   B::
    p=&c;
    p->display();   C::
}
```

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

★ 未加virtual前，“基类指针=&派生类对象”/“基类引用=派生类对象”，适用赋值兼容规则，访问的是派生类中的基类部分

★ 加virtual后，突破此限制，访问派生类的同名函数

使用：

★ virtual在类定义时出现，函数体外实现部分不能加

```
class A {
  public:
    virtual void display();
};
virtual void A::display()
{
    cout << "A::display()" << endl;
}
```

error C2723: “A::display”:“virtual”说明符在函数定义上非法

```
class A {
  public:
    virtual void display();
};

class B:public A {
  public:
    virtual void display();
};

class C:public B {
  public:
    virtual void display();
};
```

必须

建议

建议

★ 在类的继承序列中，只需要在最开始的基类中加virtual声明，后续派生类可以不加(建议加)

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 类的继承序列中该同名函数的参数个数、参数类型必须完全相同

★ 若类的继承层次中同名虚函数仅返回类型不同，则象重载一样，认为是错误

编译错误

```
class A {
  public:
    virtual void display();
};
class B:public A {
  public:
    virtual int display();
};
class C:public B {
  public:
    virtual double display();
};
```

```
(10,17): error C2555: “B::display”: 重写虚函数返回类型有差异，且不是来自“A::display”的协变
(6): message : 参见“A::display”的声明
(14,20): error C2555: “C::display”: 重写虚函数返回类型有差异，且不是来自“B::display”的协变
(10): message : 参见“B::display”的声明
```

★ 若派生类中无同名函数，则自动继承基类

```
class A {
  public:
    virtual void display();
};
class B:public A {
    无 display 函数
};
class C:public B {
  public:
    virtual void display();
};
int main()
{ A a;
  B b;
  C c;
  A *p;
  p=&a;
  p->display(); A::
  p=&b;
  p->display(); A::
  p=&c;
  p->display(); C::
}
```

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 若派生类中有同名函数，其参数个数、参数类型与基类的虚函数不同，则失去多态性，按支配规则及赋值兼容规则处理

```
class A {
  public:
    virtual void display() {
        cout << "A::" << endl;
        }
};
class B:public A {
  public:
    void display(int x) {
        cout << "B::x" << endl;
        }
};
class C:public B {
  public:
    void display() {
        cout << "C::" << endl;
        }
};
```

问：三句错的语句如何改正确？
答：第一句可改，后两句无法改
因为基类无法访问派生类成员

```
int main()
{ A a, *p;
  B b;
  C c;
  b.display();      错
  b.display(1);     B::x
  p=&a;
  p->display();     A::
  p=&b;
  p->display();     A::
  p->display(1);    错
  p=&c;
  p->display();     C::
  p->display(1);    错
}
```

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 对于派生类中的其它非virtual仍适用赋值兼容规则

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void display();
    void show();
};
class B:public A {
    void display();
    void show();
};
class C:public B {
  public:
    void display();
    void show();
};

int main()
{ A a, *p;
  B b;
  C c;
  p=&a;
  p->display(); A::
  p->show();    A::
  p=&b;
  p->display(); B::
  p->show();    A::
  p=&c;
  p->display(); C::
  p->show();    A::
}
```

★ 只有通过基类指针/引用方式访问时才适用虚函数规则，
其它形式(对象/自身指针/引用)仍用原来的规则

```
#include <iostream>
usning namespace std;

class A {
  public:
    virtual void display() {
       cout << "A::display()" << endl;
       }
   };
class B:public A {
  public:
    void display() {
       cout << "B::display()" << endl;
       }
   };
class C:public B {
  public:
    void display() {
       cout << "C::display()" << endl;
       }
   };
```

```
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();
    b.display();
    c.display();
    ra.display();
    rb.display();
    rc.display();
    pa->display();
    pb->display();
    pc->display();
    p=&a;
    p->display();
    p=&b;
    p->display();
    p=&c;
    p->display();
}
```

```
A::
B::
C::
A::
B::
C::
A::
B::
C::
A::

B::

C::
```

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

思考题：

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual void f1() { cout << "A::f1()" << endl; }
    virtual void f2() { cout << "A::f2()" << endl; }
    virtual void f3() { cout << "A::f3()" << endl; }
    virtual void f4() { cout << "A::f4()" << endl; }
            void f5() { cout << "A::f5()" << endl; }
};

class B:public A {
  public:
    void f1() { cout << "B::f1()" << endl; }
    void f3(double f) { cout << "B::f3()" << endl;}
    int  f4() { cout << "B::f4()" << endl; }
    void f5(double f) { cout << "B::f5()" << endl; }
};
```

```
int main()
{  B b;
   A *pa=&b;
   pa->f1();
   pa->f2();
   pa->f3();
   pa->f3(10);
   pa->f4();
   pa->f5();
   pa->f5(10);
   b.f1();
   b.f2();
   b.f3();
   b.f3(10);
   b.f4();
   b.f5();
   b.f5(10);
}
```

1、A、B类的定义语句中哪些会编译错？若A/B冲突，删除B中定义

2、保证A、B类定义不冲突的情况下，main中哪些语句会编译出错？

3、去除所有错误的语句，其余正确语句的执行结果？

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 若把函数重载理解为横向重载（同一类中），则虚函数可理解为纵向重载（类的继承层次中）

★ 非类的成员函数不能声明为多态

★ 类的静态成员函数不能声明为多态

支配规则、赋值兼容规则、虚函数的区别：

支配规则　　：通过自身对象、指针、引用访问(自身的)虚函数、普通函数

赋值兼容规则：通过基类指针、对象、引用访问(派生类中基类部分的)普通函数

虚函数　　：通过基类指针、引用访问(基类和派生类的同名)虚函数

§13. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

支配规则、赋值兼容规则、虚函数的区别：

```
class A {
  public:
    virtual f1(int x) {...}
    f2(int x) {...}
};
class B:public A {
  public:
    virtual f1(int x) {...}
    f2(int x) {...}
};
```

```
void fun1(A *pa)
{  pa->f1(10); //虚函数
   pa->f2(15); //赋值兼容
}
void fun2(A &ra)
{  ra.f1(10);  //虚函数
   ra.f2(15);  //赋值兼容
}
int main()
{
   A a;
   B b;
   fun1(&a);
   fun1(&b);
   fun2(a);
   fun2(b);
   a.f1(10);  //支配
   a.f1(10);  //支配
   b.f1(10);  //支配
   b.f2(15);  //支配
}
```

```
int main()
{  A a, *pa;
   B b;
   pa = &a;
   pa->f1(10); //支配
   pa->f2(15); //支配

   pa = &b;
   pa->f1(10); //虚函数
   pa->f2(10); //赋值兼容

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

```
int main()
{  B  b;
   A &ra = b;

   ra.f1(10);  //虚函数
   ra.f2(10);  //赋值兼容

   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

引用一般不用在此处，因为只能始终指向b

```
int main()
{  A a, *pa = &a;
   B b, *pb = &b;

   pa->f1(10); //支配
   pa->f2(15); //支配

   pb->f1(10); //支配
   pb->f2(10); //支配

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

```
int main()
{  A a, &ra = a;
   B b, &rb = b;

   ra.f1(10);  //支配
   ra.f2(15);  //支配

   rb.f1(10);  //支配
   rb.f2(10);  //支配

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

§13. 多态性与虚函数

3. 虚函数

3.4. 静态关联与动态关联

关联：在编译系统中，确定标识符和存储地址的对应关系的过程称为关联

★ 包含了确定对象及所调用的函数的关系

★ 又称为联编、编联、束定、绑定(binding)

分类：

静态关联：在编译时确定对应关系(早期关联)

动态关联：在运行时确定对应关系(滞后关联)

§13. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

```
#include <iostream>
using namespace std;

class A {
   public:
     ~A() { cout << "A析构" << endl; }
};

class B : public A {
  private:
    char *s;
  public:
    B() {  s = new char[10];  }
    ~B()
     {  delete s;
        cout << "B析构"  << endl; }
};
```

```
int main()
{   B b;
}  //运行结束后，系统自动调用
   //B的析构函数，再激活A析构
```

✓

```
int main()
{   B *pb = new B;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   B *pb = new B;
    A *pa = pb;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   A *pa = new B;
    delete pa; //仅调用A析构，s无法释放
}              //运行不错，丢内存
```

✗

§13.多态性与虚函数

3.虚函数

3.5.虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

解决：将基类的析构函数声明为虚函数

```
virtual ~ 类名()
{
    函数体
}
```

§13. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

```
#include <iostream>
using namespace std;

class A {
   virtual public:
      ~A() { cout << "A析构" << endl; }
};

class B : public A {
  private:
    char *s;
  public:
    B() {  s = new char[10];  }
    ~B()
     {  delete s;
        cout << "B析构"  << endl; }
};
```

```
int main()
{   B b;
}   //运行结束后，系统自动调用
    //B的析构函数，再激活A析构
```

```
int main()
{   B *pb = new B;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

```
int main()
{   B *pb = new B;
    A *pa = pb;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

```
int main()
{   A *pa = new B;
    delete pa; //调用B析构
}              //再激活A的析构
```

§13. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

解决：将基类的析构函数声明为虚函数

```
virtual ~ 类名()
{
    函数体
}
```

★ 使用于派生类中有动态申请空间的情况，虽然类的继承序列中析构函数名不同，系统会自动当作虚函数处理

★ 虚析构函数调用时，先派生类，再基类

（普通虚函数　：只调派生类，不调基类

普通析构函数：先调派生类，再调基类）

★ 析构函数声明为虚函数后，通过基类、派生类自己生成的对象在释放时也不会出错，因此一般在类的继承序列中，建议将析构函数声明为虚析构函数

★ 构造函数不能声明为虚函数（虚函数只有和对象结合才能呈现多态，构造函数时对象正在生成）

§13. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

虚析构函数

```
#include <iostream>
using namespace std;

class A {
  public:
    virtual ~A() { cout << "A析构" << endl; }
};
class B:public A {
  private:
    char *s;
  public:
    B() { s=new char[10]; }
    ~B() { delete s; cout << "B析构" << endl; }
};
```

基类指针方式

```
int main()
{
   A *pa = new B;
   delete pa;      B析构
   return 0;       A析构
}
```

派生类对象/指针方式

```
int main()
{
   B *pb = new B;
   delete pb;      B析构
   return 0;       A析构
}
```

普通析构函数

```
#include <iostream>
using namespace std;

class A {
  public:
    ~A() { cout << "A析构" << endl; }
};
class B:public A {
  private:
    char *s;
  public:
    B() { s=new char[10]; }
    ~B() { delete s; cout << "B析构" << endl; }
};
```

基类指针方式

```
int main()
{
   A *pa = new B;
   delete pa;      A析构
   return 0;       丢内存!
}
```

派生类对象/指针方式

```
int main()
{
   B *pb = new B;
   delete pb;      B析构
   return 0;       A析构
}
```

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

引入：在类的继承层次中，派生类都有同名函数，而基类没有，为使用虚函数机制，需要建立一条从基类到派生类的虚函数路径，因此在基类中定义一个同名空虚函数

形式：函数名、参数个数、参数类型、返回值与派生类相同，函数体为空即可

```
class A {
  有实际意义的数据成员及函数
  virtual void display() { }
  virtual int show(){ return 0; }
};
class B:public A {
  有实际意义的数据成员及函数
  void display(){ 具体实现 }
  int show() { 具体实现 }
};

//CDEFG等相同
  1.有各自有意义的数据成员及函数
  2.各自独立的display及show函数
    参数个数、参数类型、返回值同
    但实现过程各不相同
```

```
A中定义display及show后
可用统一方法调用
例：F f;  B b;
    A *pa = &f;
    pa->display();
    pa->show(10);
    pa = &b;
    pa->display();
    pa->show(15);
```

★ 基类的该函数虽然无意义，但基类的其它部分仍有意义，可定义对象、引用、指针等并进行正常操作

```
class A {
  有实际意义的数据成员及函数
  virtual void display() { }
  virtual int show(){ return 0; }
};
//BCDEFG的定义

void main()
{   C c1;
    A a1, *pa=&c1;
    a1.****;          //正常操作
    pa->show(20);     //虚函数形式
}
```

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

面向对象方法学的含义：为了对各类进行归纳，在更高的层次、更抽象的级别上考虑问题，简化复杂性，引入抽象类

下中国象棋
下国际象棋
下围棋
下跳棋
} 下棋

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

```
假设A-G每个类都有display
且参数个数、参数类型、返回值都相同
class A {
  其它成员及函数
  void display(){ 具体实现 }
};
...
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```

```
假设A-G每个类都有display
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
};
class A:public V { //继承V
  其它成员及函数
  void display(){ 具体实现 }
};
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```

```
A a1;
G g1;
V *p;
p = &a1;
p->display()
p=&g1;
p->display();
```

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

★ 也可以用于统一几个独立的继承层次

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

```
假设A-G每个类都有display
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
};
class A:public V { //继承V
  其它成员及函数
  void display(){ 具体实现 }
};
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```

```
A a1;
G g1;
V *p;
p = &a1;
p->display()
p=&g1;
p->display();
```

存在的问题：

V还能定义对象 V v1;

但V的对象实际无意义

解决方法：

将V定义为抽象类

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.2. 纯虚函数与抽象类

抽象类的定义：C++中无明确的关键字定义，只要声明某一成员函数为纯虚函数即可

纯虚函数的声明：

```
virtual 返回类型 函数名(参数表) = 0;
```

★ 表示该函数没有实际意义，也不被调用

例：

```
class V {
  public:
     virtual void display()=0;
  };
```

假设A-G每个类都有display

```
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
  };
```

```
class V {
  public:
    virtual void display()=0;
  };
```

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.2. 纯虚函数与抽象类

抽象类的使用：

★ 抽象类不能有实例对象，但可用于声明指针或引用

```
class V {
    public:
        virtual void display()=0;
    };
V v1;                //错误
V *p;                //正确
V &p=派生类对象;     //正确(一般用于函数形参)
```

★ 抽象类中定义数据成员及有实际意义的成员函数都是无意义的，但为了简化继承序列，可以进行定义，供派生类使用

(会导致理解混乱，初学者不推荐)

例：假设 A,B,C中都有int a,b成员

```
class A {
   protected:
      int a,b;
};
class B {
   protected:
      int a,b;
};
class C {
   protected:
      int a,b;
};
```

```
class V {
   protected:
      int a, b;
};
class A:public V {
   protected:
      ...
};
class B:public V {
   protected:
      ...
};
class C:public V {
   protected:
     ...
};
```

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.2. 纯虚函数与抽象类

抽象类的使用：

★ 抽象类的直接派生类的同名虚函数必须定义，否则继承抽象类的纯虚函数，也成为抽象类(若不需要，可定义为空虚函数)

```
若：class V {
      public:
        virtual void display()=0;
      };

则：ABC中的display()必须定义
    即使B中不需要display，也要定义

    class B:public V {
       public:
         void display() { }
       };
```

空虚函数与纯虚函数的区别：

空虚函数：类的其它成员有实际含义，可生成对象

纯虚函数：无实例对象，无实际含义，仅为了在更高的层次上统一类

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.3. 应用实例

例：将各种几何形状的求面积、表面积、...等做成一个继承序列

三个独立继承序列

加Shape抽象类

§13. 多态性与虚函数

4. 纯虚函数与抽象类

4.3. 应用实例

```
class Shape {
    public:
        virtual double area() const { return 0.0; }
        virtual double volumn() const { return 0.0; }
        virtual void shapeName() const = 0;
    };
```

以此继承序列为例

★ area()为空虚函数，在Point中可不再定义

★ volumn()为空函数，在Point、Circle中可不再定义

★ 选择shapeName()为纯虚函数，为了声明抽象类，且shapeName()每个类中必须再次定义

area 为空虚函数
volumn为空虚函数

不需要定义area
不需要定义volumn

定义area
不需要定义volumn

定义area
定义volumn

```
Point p1;
Circle c;
Cylinder cy1;
shape *p;
p=&p1;
p->area();    //0.0无意义
p->volumns();//0.0无意义
p=&c;
p->area()     //有意义
p->volumn(); //0.0无意义
p=&cy1;
p->area();    //有意义
p->volumn(); //有意义
```

§. 基础知识题 – 进制转换

1. 十进制转二进制

A. 2023

2	2023	
2	1011	1
2	505	1
2	252	1
2	126	0
2	63	0
2	31	1
2	15	1
2	7	1
2	3	1
2	1	1
	0	

$$(2023)_{10} = (11111100111)_2$$

B. 50758

2	50758	
2	25379	0
2	12689	1
2	6344	1
2	3172	0
2	1586	0
2	793	0
2	396	1
2	198	0
2	99	0
2	49	1
2	24	1
2	12	0
2	6	0
2	3	0
2	1	1
	0	

$$(50758)_{10} = (1100011001000110)_2$$

C. 0.375

$$\begin{array}{r} 0.375 \\ \times \quad 2 \\ \hline 0.750 \cdots 0 \end{array}$$

小数部分：

$$\begin{array}{r} 0.75 \\ \times \quad 2 \\ \hline 1.50 \cdots 1 \end{array}$$

$$\begin{array}{r} 0.5 \\ \times \quad 2 \\ \hline 1.0 \cdots 1 \end{array}$$

$$(0.375)_{10} = (0.011)_2$$

D. 8.0625

整数部分：

2	8	
2	4	0
2	2	0
2	1	0
	0	1

小数部分：

$$\begin{array}{r} 0.0625 \\ \times \quad 2 \\ \hline 0.1250 \cdots 0 \end{array}$$

$$\begin{array}{r} 0.125 \\ \times \quad 2 \\ \hline 0.250 \cdots 0 \end{array}$$

$$\begin{array}{r} 0.25 \\ \times \quad 2 \\ \hline 0.50 \cdots 0 \end{array}$$

$$\begin{array}{r} 0.5 \\ \times \quad 2 \\ \hline 1.0 \cdots 1 \end{array}$$

$$(8.0625)_{10} = (1000.0001)_2$$

§. 基础知识题 – 进制转换

2. 二进制转十进制

A. 10101110110

$$= 1\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + 0\times 2^{7} + 1\times 2^{6} + 1\times 2^{5} + 1\times 2^{4} + 0\times 2^{3} + 1\times 2^{2} + 1\times 2^{1} + 0\times 2^{0}$$
$$= 1024 + 0 + 256 + 0 + 64 + 32 + 16 + 0 + 4 + 2 + 0$$
$$= 1398$$

$$(10101110110)_2 = (1398)_{10}$$

B. 1100011001000110

$$= 1\times 2^{15} + 1\times 2^{14} + 0\times 2^{13} + 0\times 2^{12} + 0\times 2^{11} + 1\times 2^{10} + 1\times 2^{9} + 0\times 2^{8} + 0\times 2^{7} + 1\times 2^{6} + 0\times 2^{5} + 0\times 2^{4} + 0\times 2^{3} + 1\times 2^{2} + 1\times 2^{1} + 0\times 2^{0}$$
$$= 32768 + 16384 + 0 + 0 + 0 + 1024 + 512 + 0 + 0 + 64 + 0 + 0 + 0 + 4 + 2 + 0$$
$$= 50758$$

$$(1100011001000110)_2 = (50758)_{10}$$

C. 1011.1101

$$= 1\times 2^{3} + 0\times 2^{2} + 1\times 2^{1} + 1\times 2^{0} + 1\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4}$$
$$= 8 + 0 + 2 + 1 + 0.5 + 0.25 + 0 + 0.0625$$
$$= 11.8125$$

$$(1011.1101)_2 = (11.8125)_{10}$$

D. 0.10111101

$$= 0\times 2^{0} + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4} + 1\times 2^{-5} + 1\times 2^{-6} + 0\times 2^{-7} + 1\times 2^{-8}$$
$$= 0 + 0.5 + 0 + 0.125 + 0.0625 + 0.03125 + 0.015625 + 0 + 0.00390625$$
$$= 0.73828125$$

$$(0.1011101)_2 = (0.73828125)_{10}$$

§. 基础知识题 – 进制转换

3. 十进制与八进制的相互转换

（1） 十进制转八进制

A. 3905

$$
\begin{array}{r|rl}
8 & 3905 & \\
8 & 488 & 1 \\
8 & 61 & 0 \\
8 & 7 & 5 \\
 & 0 &
\end{array}
$$

$$(3905)_{10} = (7501)_8$$

B. 50758

$$
\begin{array}{r|rl}
8 & 50758 & \\
8 & 6344 & 6 \\
8 & 793 & 0 \\
8 & 99 & 1 \\
8 & 12 & 3 \\
8 & 1 & 4 \\
 & 0 & 1
\end{array}
$$

$$(50758)_{10} = (143106)_8$$

（2） 八进制转十进制

A. 617032

$$
\begin{aligned}
& 617032 \\
&= 6 \times 8^5 + 1 \times 8^4 + 7 \times 8^3 + 0 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 \\
&= 196608 + 4096 + 3584 + 0 + 24 + 2 \\
&= 204314
\end{aligned}
$$

$$(617032)_8 = (204314)_{10}$$

B. 143106

$$
\begin{aligned}
& 143106 \\
&= 1 \times 8^5 + 4 \times 8^4 + 3 \times 8^3 + 1 \times 8^2 + 0 \times 8^1 + 6 \times 8^0 \\
&= 32768 + 16384 + 1536 + 64 + 0 + 6 \\
&= 50758
\end{aligned}
$$

$$(143106)_8 = (50758)_{10}$$

§. 基础知识题 – 进制转换

4. 十进制与十六进制的相互转换

（1） 十进制转十六进制

A. 3905

$$
\begin{array}{r|rl}
16 & 3905 & \\
16 & 244 & 1 \\
16 & 15 & 4 \\
 & 0 &
\end{array}
$$

$$(3905)_{10} = (F41)_{16}$$

B. 50758

$$
\begin{array}{r|rl}
16 & 50758 & \\
16 & 31\ 2 & 6 \\
16 & 19 & 4 \\
16 & 12 & 6 \\
 & 0 & 1
\end{array}
$$

$$(50758)_{10} = (C646)_{16}$$

（2） 十六进制转十进制

A. C35B8

$$
\begin{aligned}
& C35B8 \\
&= 12 \times 16^4 + 3 \times 16^3 + 5 \times 16^2 + 11 \times 16^1 + 8 \times 16^0 \\
&= 786432 + 12288 + 1280 + 176 + 8 \\
&= 800184
\end{aligned}
$$

$$(C35B8)_{16} = (800184)_{10}$$

B. C646

$$
\begin{aligned}
& C646 \\
&= 12 \times 16^3 + 6 \times 16^2 + 4 \times 16^1 + 6 \times 16^0 \\
&= 49152 + 1536 + 64 + 6 \\
&= 50758
\end{aligned}
$$

$$(C646)_{16} = (50758)_{10}$$

§. 基础知识题 – 进制转换

5. 其他进制的相互转换

（1） 二进制转八进制

A. 10101110110

$(10101110110)_2 = 10\ 101\ 110\ 110 = (2566)_8$

B. 1100011001000110

$(1100011001000110)_2 = 1\ 100\ 011\ 001\ 000\ 110 = (143106)_8$

（2） 八进制转二进制

A. 617032

$(617032)_8 = 110\ 001\ 111\ 000\ 011\ 010 = (110001111000011010)_2$

B. 143106

$(143106)_8 = 001\ 100\ 011\ 001\ 000\ 110 = (1100011001000110)_2$

（3） 二进制转十六进制

A. 10101110110

$(10101110110)_2 = 101\ 0111\ 0110 = (576)_{16}$

B. 1100011001000110

$(1100011001000110)_2 = 1100\ 0110\ 0100\ 0110 = (C646)_{16}$

§. 基础知识题 – 进制转换

5. 其他进制的相互转换

（4） 十六进制转二进制

A. C35B8

$(C35B8)_{16} = 1100\ 0011\ 0101\ 1011\ 1000 = (11000011010110111000)_2$

B. C646

$(C646)_{16} = 1100\ 0110\ 0100\ 0110 = (1100011001000110)_2$

（5） 八进制转十六进制

A. 617032

$(617032)_8 = 110\ 001\ 111\ 000\ 011\ 010 = 0011\ 0001\ 1110\ 0001\ 1010 = (31E1A)_{16}$

B. 143106

$(143106)_8 = 001\ 100\ 011\ 001\ 000\ 110 = 1100\ 0110\ 0100\ 0110 = (C646)_{16}$

（6） 十六进制转八进制

A. C35B8

$(C35B8)_{16} = 1100\ 0011\ 0101\ 1011\ 1000 = 011\ 000\ 011\ 010\ 110\ 111\ 000 = (3032670)_8$

B. C646

$(C646)_{16} = 1100\ 0110\ 0100\ 0110 = 001\ 100\ 011\ 001\ 000\ 110 = (143106)_8$

§.基础知识题 - 二进制补码

1.十进制整数转二进制补码（仿照课件PDF的P.19，写出具体步骤，包括绝对值、取反、+1）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

A.-122　（假设为1字节整数）

数值	-122
绝对值的二进制表示	1111010
原码	01111010
取反+1，得到补码	10000101 +) 1 ---------- 10000110

§.基础知识题 - 二进制补码

1.十进制整数转二进制补码（仿照课件PDF的P.19，写出具体步骤，包括绝对值、取反、+1）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

B.-244　（假设为2字节整数）

数值	-244
绝对值的二进制表示	11110100
原码	00000000 11110100
取反+1，得到补码	11111111 00001011 +) 1 ——————————— 11111111 00001100

§.基础知识题 - 二进制补码

1.十进制整数转二进制补码（仿照课件PDF的P.19，写出具体步骤，包括绝对值、取反、+1）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

C.-244　（假设为4字节整数）

数值	-244
绝对值的二进制表示	11110100
原码	00000000 00000000 00000000 11110100
取反+1，得到补码	11111111 11111111 11111111 00001011 +) 1 ---------- 11111111 11111111 11111111 00001100

§.基础知识题 - 二进制补码

1.十进制整数转二进制补码（仿照课件PDF的P.19，写出具体步骤，包括绝对值、取反、+1）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

D.本人学号逆序后取最多五位对应的int型十进制负数

数值	-85705
绝对值的二进制表示	10100111011001001
原码	00000000 00000001 01001110 11001001
取反+1，得到补码	11111111 11111110 10110001 00110110 +) 1 ---------- 11111111 11111110 10110001 00110111

§.基础知识题 - 二进制补码

2.二进制补码转十进制整数（只考虑有符号数，写出具体步骤，包括-1、取反、绝对值、加负号）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

A.1011 0101

补码-1	10110101 -) 1 ---------- 10110100
取反，得到原码	01001011
绝对值的二进制表示	1001011
绝对值的十进制表示	75
加负号，得到数值	-75

§.基础知识题 - 二进制补码

2.二进制补码转十进制整数（只考虑有符号数，写出具体步骤，包括-1、取反、绝对值、加负号）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

B.1011 1001 1101 1010

补码-1	10111001 11011010 -) 1 ---------- 10111001 11011001
取反，得到原码	01000110 00100110
绝对值的二进制表示	100011000100110
绝对值的十进制表示	17958
加负号，得到数值	-17958

§.基础知识题 - 二进制补码

2.二进制补码转十进制整数（只考虑有符号数，写出具体步骤，包括-1、取反、绝对值、加负号）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

C.1101 1101 0110 0000 0110 1011 1001 0000

补码-1	11011101 01100000 01101011 10010000 -) 1 ———————————————— 11011101 01100000 01101011 10001111
取反，得到原码	00100010 10011111 10010100 01110000
绝对值的二进制表示	100010100111111001010001110000
绝对值的十进制表示	580883568
加负号，得到数值	-580883568

§.基础知识题 - 二进制补码

2.二进制补码转十进制整数（只考虑有符号数，写出具体步骤，包括-1、取反、绝对值、加负号）

格式要求：多字节时，每8bit中间加一个空格或-(例："11010100 00110001" 或 "11010100-00110001")

D.本人学号逆序后取最多五位对应的int型十进制负数的二进制补码形式

补码-1	11111111 11111110 10110001 00110111 -) 1 ---------- 11111111 11111110 10110001 00110110
取反，得到原码	00000000 00000001 01001110 11001001
绝对值的二进制表示	10100111011001001
绝对值的十进制表示	85705
加负号，得到数值	-85705

§.基础知识题 - 字符串长度

1.求出下列字符串的长度

A. "\bvt\\tnc\4921\x3fr\2a\'\r\\v\a\f" =21

B. "\19\x2f\43\x8x\383\x65\012\xd5\231\xe3\1325\x6a" =17

§.基础知识题 – 字符串长度

1.求出下列字符串的长度

```
#include <iostream>
using namespace std;

int main()
{
    cout << "2250758" << endl;
    cout << strlen("\23456f") << endl;
    cout << strlen("\43456f") << endl;
    return 0;
}
```

C.运行上面的程序，贴含本人学号的源程序+编译器的错误信息截图

观察编译信息，得到结论如下：

1、转义符\后的合法8进制数>3个，则转义符\与其之后的3个合法8进制数组成1个转义字符。

2、转义符\后的合法8进制数≤3个但超出上限377，则出现error C2022的提示，转义符\与其之后的3个合法8进制数组成的转义字符不能表示出某一个字符。

编译提示中的那个数字是怎么来的？

$(434)_8=(284)_{10}$

§.基础知识题 – 字符串长度

1.求出下列字符串的长度

```
#include <iostream>
using namespace std;

int main()
{
    cout << "2250758" << endl;
    cout << strlen("\x23") << endl;
    cout << strlen("\x234") << endl;
    return 0;
}
```

D.运行上面的程序，贴含本人学号的源程序+编译器的错误信息截图

观察编译信息，得到结论如下：

1、转义符\x后的合法16进制数>2个，则转义符\x与其之后的合法16进制数组成1个转义字符。
编译提示中的那个数值是怎么来的？
$(234)_{16}=(564)_{10}$

2、综合CD，在用转义符表示8/16进制时，超过限定的长度的错误处理是不一致的。

§.基础知识题 – 字符串长度

1.求出下列字符串的长度

```
#include <iostream>
using namespace std;

int main()
{
    cout << "2250758" << endl;
    cout << strlen("\9876") << endl;
    cout << strlen("\*321") << endl;
    return 0;
}
```

E.运行上面的程序，贴含本人学号的源程序+编译器的错误信息截图

观察编译信息，得到结论如下：

1、转义符\后直接跟非法的8进制，则'\非法的8进制'是不可识别的字符转义序列，转义符\被忽略，其含义等价于'非法的8进制'。

2、对两个strlen的输出结果进行分析（合理猜测）

'\9'是不可识别的字符转义序列，转义符\被忽略，其含义等价于'9'，故字符串"\9876"的长度为4；

'*'是不可识别的字符转义序列，转义符\被忽略，其含义等价于'*'，故字符串"*321"的长度为4。

§.基础知识题 – 字符串长度

1.求出下列字符串的长度

```
#include <iostream>
using namespace std;

int main()
{
    cout << "2250758" << endl;
    cout << strlen("\xg231") << endl;
    cout << strlen("\x*231") << endl;
    return 0;
}
```

[Error] \x used with no following hex digits
[Error] \x used with no following hex digits

F.运行上面的程序，贴含本人学号的源程序+编译器的错误信息截图

观察编译信息，得到结论如下：

1、转义符\x后直接跟非法的16进制，则出现error C2153的提示，十六进制常量必须至少有一个十六进制数字，\x后面必须至少有一个十六进制数字。

2、综合EF，在用转义符表示8/16进制时，直接跟非法字符的错误处理是不一致的。

§.基础知识题 – 变量及表达式求值

1.给出下列程序段中变量b的值

```
A. short a=32750;
   short b=a+24;

Step1：b=a+24，得b二进制补码形式
       a = 00000000 00000000 01111111 11101110 -> a（红色表示整型提升的填充位）
  +) 24 = 00000000 00000000 00000000 00011000 -> 24
  ----------------------------------------
           00000000 00000000 10000000 00000110 -> a+24（int型）
       b = ~~00000000 00000000~~ 10000000 00000110 -> b=a+24（二进制补码形式，删除线表示丢弃的位数）

Step2：求b的十进制表示
   （1）减一      10000000 00000110
             -) 00000000 00000001
             --------------------
                10000000 00000101
   （2）取反      01111111 11111010
   （3）绝对值   32762  （十进制表示形式）
   （4）加负号   -32762（十进制表示形式）
```

§.基础知识题 – 变量及表达式求值

1.给出下列程序段中变量b的值

B. unsigned short a=65520;
short b=a;

Step1：b=a，得b二进制补码形式

a = 00000000 00000000 11111111 11110000 -> a（红色表示整型提升的填充位）

a = 00000000 00000000 11111111 11110000 -> a（int型）

b = ~~00000000 00000000~~ 11111111 11110000 -> b=a（二进制补码形式，删除线表示丢弃的位数）

Step2：求b的十进制表示

（1）减一 11111111 11110000
-) 00000000 00000001

11111111 11101111

（2）取反 00000000 00010000

（3）绝对值 16 （十进制表示形式）

（4）加负号 -16（十进制表示形式）

§.基础知识题 - 变量及表达式求值

1.给出下列程序段中变量b的值

```
C. short a=-4095;
   int b=a;

Step1：b=a，得b二进制补码形式
   a = 11111111 11111111 11110000 00000001 -> a（红色表示整型提升的填充位）
   a = 11111111 11111111 11110000 00000001 -> a（int型）
   b = 11111111 11111111 11110000 00000001 -> b=a（二进制补码形式）

Step2：求b的十进制表示
   （1）减一     11111111 11111111 11110000 00000001
              -) 00000000 00000000 00000000 00000001
              ---------------------------------------
                 11111111 11111111 11110000 00000000
   （2）取反     00000000 00000000 00001111 11111111
   （3）绝对值   4095  （十进制表示形式）
   （4）加负号   -4095（十进制表示形式）
```

§.基础知识题 – 变量及表达式求值

1.给出下列程序段中变量b的值

```
D.unsigned short a=65520;
  long long int b=a;
```

Step1：b=a，得b二进制补码形式

a = 00000000 00000000 11111111 11110000 -> a

（红色表示整型提升的填充位）

a = 00000000 00000000 00000000 00000000 00000000 00000000 11111111 11110000 -> a

（long long型）

b = 00000000 00000000 00000000 00000000 00000000 00000000 11111111 11110000 -> b=a

（二进制补码形式）

Step2：求b的十进制表示

$(1111111111110000)_2=(65520)_{10}$

§.基础知识题 – 变量及表达式求值

1.给出下列程序段中变量b的值

E. long long int a=4207654321;
int b=a;

Step1：b=a，得b二进制补码形式

a = 00000000 00000000 00000000 00000000 11111010 11001011 10110101 10110001 -> a

a = ~~00000000 00000000 00000000 00000000~~ 11111010 11001011 10110101 10110001 -> a

（int型，删除线表示丢弃的位数）

b = 11111010 11001011 10110101 10110001 -> b=a

（二进制补码形式）

Step2：求b的十进制表示

（1）减一 11111010 11001011 10110101 10110001

-) 00000000 00000000 00000000 00000001

11111010 11001011 10110101 10110000

（2）取反 00000101 00110100 01001010 01001111

（3）绝对值 87312975 （十进制表示形式）

（4）加负号 -87312975（十进制表示形式）

§.基础知识题 – 变量及表达式求值

1.给出下列程序段中变量b的值

F. long a=-4207654321;
  unsigned short b=a;

Step1：确定-4207654321的类型

-4207654321为有符号数，long型的数值范围为-2147483648~+2147483647，long long型的数值范围为-9223372036854775808~+9223372036854775807，-4207654321超出了long型数值范围的下限-2147483648，故-4207654321的类型为long long型。通过“cout << typeid(-4207654321).name() << endl;”的编译结果为“__int64”对-4207654321的类型为long long型进行了验证。

Step2：确定a的值

-4207654321 = 11111111 11111111 11111111 11111111 00000101 00110100 01001010 01001111

a = ~~11111111 11111111 11111111 11111111~~ 00000101 00110100 01001010 01001111

（long型，删除线表示丢弃的位数）

a = 00000101 00110100 01001010 01001111

（二进制补码形式）

Step3：b=a，得b二进制补码形式

a = 00000101 00110100 01001010 01001111 -> a

a = ~~00000101 00110100~~ 01001010 01001111 -> a（unsigned short型，删除线表示丢弃的位数）

b = 01001010 01001111 -> b（二进制补码形式）

Step4：求b的十进制表示

$(100101001001111)_2=(19023)_{10}$

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

A. 21 / 2 + 47 % 3 - 1.3 + 3.5 * 2

表达式一共有6个运算符，因此计算的6个步骤分别是（本页不需要画栈）：

步骤①：21 / 2 => 式1

步骤②：47 % 3 => 式2

步骤③：式1 + 式2 => 式3

步骤④：式3 - 1.3 => 式4

步骤⑤：3.5 * 2 => 式5

步骤⑥：式4 + 式5

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

A. 21 / 2 + 47 % 3 - 1.3 + 3.5 * 2

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

要进栈的(+)低于栈顶的(/)，先计算

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

A. 21 / 2 + 47 % 3 – 1.3 + 3.5 * 2

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

运算数栈	运算符栈
3	%
47	+
21 / 2	

要进栈的(-)低于栈顶的(%)，先计算

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

A. 21 / 2 + 47 % 3 - 1.3 + 3.5 * 2

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

式1：21 / 2
式2：47 % 3
式3：式1 + 式2
式4：式3 - 1.3

3.5 式4	* +

*进栈　要进栈的(*)高于栈顶的(+)

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

B. a = 3 * 5 , a = b = 6 * 4 （假设所有变量均为int型）

表达式一共有6个运算符，因此计算的6个步骤分别是：

步骤①：3 * 5 => 式1

步骤②：a = 式1 => 式2

步骤③：6 * 4 => 式3

步骤④：b = 式3 => 式4

步骤⑤：a = 式4 => 式5

步骤⑥：式2 , 式5

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

B. a = 3 * 5 , a = b = 6 * 4 （假设所有变量均为int型）

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

要进栈的(,)低于栈顶的(*)，先计算

§. 基础知识题 – 变量及表达式求值

2. 用栈方式给出下列表达式的求解过程

B. a = 3 * 5 , a = b = 6 * 4 （假设所有变量均为int型）

目前已分析到整个表达式的尾部，画出从当前栈的状态到整个表达式分析完成的整个过程

步骤①：3 * 5 => 式1
步骤②：a = 式1 => 式2
步骤③：6 * 4 => 式3
步骤④：b = 式3 => 式4
步骤⑤：a = 式4 => 式5
步骤⑥：式2 , 式5

计算 6 * 4		计算 b = 式3		计算 a = 式4		计算 式2 , 式5		求值完成	
4									
6	*	式3							
b	=	b	=	式4					
a	=	a	=	a	=	式5			
式2	,	式2	,	式2	,	式2	,	式5	

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

C. a + (b - 3 * (a + c) - 2) % 3　(假设所有变量均为int型)

(本题提示：将左右小括号分开处理，

1、"("进栈前优先级最高，进栈后优先级最低；

2、")"优先级最低，因此要将栈中压在"("之上的全部运算符都计算完成，随后和"("成对消除即可)

表达式一共有 10 个运算符，因此计算的 6 个步骤分别是：

步骤①：a + c => 式1

步骤②：3 * 式1 => 式2

步骤③：b - 式2 => 式3

步骤④：式3 - 2 => 式4

步骤⑤：式4 % 3 => 式5

步骤⑥：a + 式5

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

C. a + (b - 3 * (a + c) - 2) % 3　(假设所有变量均为int型)

(本题提示：将左右小括号分开处理，

1、"("进栈前优先级最高，进栈后优先级最低；

2、")"优先级最低，因此要将栈中压在"("之上的全部运算符都计算完成，随后和"("成对消除即可)

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

+进栈　要进栈的(+)高于栈顶的(()

§.基础知识题 – 变量及表达式求值

2.用栈方式给出下列表达式的求解过程

C. a + (b – 3 * (a + c) – 2) % 3　（假设所有变量均为int型）

（本题提示：将左右小括号分开处理，

1、"("进栈前优先级最高，进栈后优先级最低；

2、")"优先级最低，因此要将栈中压在"("之上的全部运算符都计算完成，随后和"("成对消除即可）

目前准备进栈的运算符如箭头所示，画出当前运算数栈和运算符栈的状态（本页需要画栈）

运算数栈	运算符栈
2	–
b-3*(a+c)	(
a	+

要进栈的())低于栈顶的(-)，先计算

§. 基础知识题 – 变量及表达式求值

2. 用栈方式给出下列表达式的求解过程

C. a + (b – 3 * (a + c) – 2) % 3 （假设所有变量均为int型）

（本题提示：将左右小括号分开处理，

1、"("进栈前优先级最高，进栈后优先级最低；

2、")"优先级最低，因此要将栈中压在"("之上的全部运算符都计算完成，随后和"("成对消除即可）

目前已分析到整个表达式的尾部，画出从当前栈的状态到整个表达式分析完成的整个过程

步骤①：a + c => 式1
步骤②：3 * 式1 => 式2
步骤③：b – 式2 => 式3
步骤④：式3 – 2 => 式4
步骤⑤：式4 % 3 => 式5
步骤⑥：a + 式5

计算 式4 % 3　　计算 a + 式5　　求值完成

§.基础知识题 – 变量及表达式求值

3. 求表达式的值

A. a = 2 * 3 , a = b = 5 * 7（写验证程序时，假设所有变量均为int型）

(1)	2 * 3	=> 6	int型
(2)	a = 2 * 3	=> 6	int型
(3)	5 * 7	=> 35	int型
(4)	b = 5 * 7	=> 35	int型
(5)	a = b = 5 * 7	=> 35	int型
(6)	a = 2 * 3 , a = b = 5 * 7	=> 35	int型

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << (a = 2 * 3, a = b = 5 * 7) << endl;
    cout << typeid(a = 2 * 3, a = b = 5 * 7).name() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
35
int
```

§.基础知识题 – 变量及表达式求值

3. 求表达式的值

B. a - (b + 4 * (b + c) / 3) % 5（写验证程序时，假设所有变量均为int型，int a=1,b=2,c=3;）

(1)	b + c	=> 5	int型
(2)	(b + c)	=> 5	int型
(3)	4 * (b + c)	=> 20	int型
(4)	4 * (b + c) / 3	=> 6	int型
(5)	b + 4 * (b + c) / 3	=> 8	int型
(6)	(b + 4 * (b + c) / 3)	=> 8	int型
(7)	(b + 4 * (b + c) / 3) % 5	=> 3	int型
(8)	a - (b + 4 * (b + c) / 3) % 5	=> -2	int型

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    cout << (a - (b + 4 * (b + c) / 3) % 5) << endl;
    cout << typeid(a - (b + 4 * (b + c) / 3) % 5).name() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
-2
int
```

§.基础知识题 – 变量及表达式求值

3.求表达式的值

C. 2.5 * 4UL + 7U * 5ULL – 'x'

(1) 2.5 * 4UL => 10 double型
(2) 7U * 5ULL => 35 unsigned long long型
(3) 2.5 * 4UL + 7U * 5ULL => 45 double型
(4) 2.5 * 4UL + 7U * 5ULL – 'x' => -75 double型

My Project.cpp

My Project (全局范围)

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << (2.5 * 4UL + 7U * 5ULL - 'x') << endl;
7      cout << typeid(2.5 * 4UL + 7U * 5ULL - 'x').name() << endl;
8      return 0;
9  }
10
```

Microsoft Visual Studio 调试控制台

```
-75
double
```

§.基础知识题 – 变量及表达式求值

3.求表达式的值

D. 3LU % 7 + 13LL % 3 + 3.5F

(1) 3LU % 7 => 3 unsigned long型
(2) 13LL % 3 => 1 long long型
(3) 3LU % 7 + 13LL % 3 => 4 long long型
(4) 3LU % 7 + 13LL % 3 + 3.5F => 7.5 float型

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << (3LU % 7 + 13LL % 3 + 3.5F) << endl;
    cout << typeid(3LU % 7 + 13LL % 3 + 3.5F).name() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
7.5
float
```

§.基础知识题 – 变量及表达式求值

3.求表达式的值

E. 3.2 + 11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2 * 3.2F

(1)	11 % 5	=> 1	int型
(2)	1.8F + 2LL	=> 3.8	float型
(3)	static_cast<unsigned long>(1.8F + 2LL)	=> 3	unsigned long型
(4)	11 % 5 * static_cast<unsigned long>(1.8F + 2LL)	=> 3	unsigned long型
(5)	11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2	=> 1	unsigned long型
(6)	11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2 * 3.2F	=> 3.2	float型
(7)	3.2 + 11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2 * 3.2F	=> 6.4	double型

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << (3.2 + 11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2 * 3.2F) << endl;
    cout << typeid(3.2 + 11 % 5 * static_cast<unsigned long>(1.8F + 2LL) % 2 * 3.2F).name() << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
6.4
double
```

§.基础知识题 – 变量及表达式求值

3.求表达式的值

F. long(3.8 + 1.3) / 2 + (int)3.9 % 7LU – 'G' * 3L

(1)	3.8 + 1.3	=> 5.1	double型
(2)	long(3.8 + 1.3)	=> 5	long型
(3)	long(3.8 + 1.3) / 2	=> 2	long型
(4)	(int)3.9	=> 3	int型
(5)	(int)3.9 % 7LU	=> 3	unsigned long型
(6)	long(3.8 + 1.3) / 2 + (int)3.9 % 7LU	=> 5	unsigned long型
(7)	'G' * 3L	=> 213	long型
(8)	long(3.8 + 1.3) / 2 + (int)3.9 % 7LU – 'G' * 3L	=> 4294967088	unsigned long型

-208(long) = 11111111 11111111 11111111 00110000 (二进制补码形式) = 4294967088 (unsigned long型)

My Project.cpp

My Project (全局范围)

```
#include <iostream>
using namespace std;

int main()
{
    cout << (long(3.8 + 1.3) / 2 + (int)3.9 % 7LU – 'G' * 3L) << endl;
    cout << typeid(long(3.8 + 1.3) / 2 + (int)3.9 % 7LU – 'G' * 3L).name() << endl;
    return 0;
}
```

C26454 算术溢出: "-"操作在编译时生成负的无符号结果(io.5)。

Microsoft Visual Studio 调试控制台

```
4294967088
unsigned long
```

§.基础知识题 – 变量及表达式求值

4. 求复合赋值表达式的值

假设int a = 3, n = 7;

A. a += a - n

(1) a - n (记作①)　　a = 3　　n = 7
=> a = a + ①
(2) a + ①　　a = 3　　n = 7　　和-1存放在中间变量中
(3) a = 和　　a = -1　　n = 7

My Project.cpp

My Project

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3, n = 7;
    a += a - n;
    cout << a << " " << n << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
-1 7
```

§. 基础知识题 – 变量及表达式求值

4. 求复合赋值表达式的值

假设int a = 3, n = 7;

B. n += a += 5

a += 5（记作①）
=> a = a + 5

(1) a + 5	a = 3	n = 7	第一个和8存放在中间变量中
(2) a = 第一个和	a = 8	n = 7	

n += ①
=> n = n + ①

(3) n + ①	a = 8	n = 7	第二个和15存放在中间变量中
(4) n = 第二个和	a = 8	n = 15	

My Project.cpp

My Project

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3, n = 7;
    n += a += 5;
    cout << a << " " << n << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
8 15
```

§.基础知识题 – 变量及表达式求值

4. 求复合赋值表达式的值

假设int a = 3;

C. a += a %= a -= a

为什么编译不报错，但运行无输出、返回代码为负值、且运行时间更长？

程序运行至除0取余步骤时，无法进行除0取余运算，C++的标准规定整型运算中除以0是“未定义行为（Undefined Behavior）”，程序异常退出，无法运行至return 0语句，返回代码-1073741676为负值代表整数除0的异常情况。所以编译不报错，但运行无输出、返回代码为负值、且运行时间更长。

a -= a（记作①）
=> a = a - a
(1) a - a　　a = 3　　差0存放在中间变量中
(2) a = 差　　a = 0
a %= ①
=> a = a % ①
(3) a % ①　　a = 0（程序运行至此，无法进行除0取余运算，程序无法运行至return 0语句）

My Project.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3;
    a += a %= a -= a;
    cout << a << endl;
    return 0;
}
```

D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe

```
--------------------------------
Process exited after 0.7677 seconds with return value 3221225620
请按任意键继续. . .
```

My Project.cpp　My Project　(全局范围)

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3;
    a += a %= a -= a;
    cout << a << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
D:\Learning\高级语言程序设计\My Project\Debug\My Project.exe (进程 3728)已退出，代码为 -1073741676。
按任意键关闭此窗口. . .
```

§.基础知识题 – 变量及表达式求值

4. 求复合赋值表达式的值

假设int a = 3, n = 7;

D. n %= a %= 3

为什么编译不报错，但运行无输出、返回代码为负值、且运行时间更长？

程序运行至除0取余步骤时，无法进行除0取余运算，C++的标准规定整型运算中除以0是“未定义行为（Undefined Behavior）”，程序异常退出，无法运行至return 0语句，返回代码-1073741676为负值代表整数除0的异常情况。所以编译不报错，但运行无输出、返回代码为负值、且运行时间更长。

a %= 3 (记作①)
=> a = a % 3

(1) a % 3	a = 3	n = 7	模0存放在中间变量中
(2) a = 模	a = 0	n = 7	

n %= ①
=> n = n % ①

(3) n % ①	a = 0	n = 7

（程序运行至此，无法进行除0取余运算，程序无法运行至return 0语句）

My Project.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3, n = 7;
    n %= a %= 3;
    cout << a << " " << n << endl;
    return 0;
}
```

D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe

```
--------------------------------
Process exited after 0.7434 seconds with return value 3221225620
请按任意键继续. . .
```

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 3, n = 7;
    n %= a %= 3;
    cout << a << " " << n << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
D:\Learning\高级语言程序设计\My Project\Debug\My Project.exe (进程 21784)已退出，代码为 -1073741676。
按任意键关闭此窗口. . .
```

§.基础知识题 - IEEE754的理解

float型数据的内部存储格式

```
#include <iostream>
using namespace std;
int main()
{
    float f = 123.456f;
    unsigned char* p = (unsigned char*)&f;
    cout << hex << (int)(*p)      << endl;
    cout << hex << (int)(*(p+1)) << endl;
    cout << hex << (int)(*(p+2)) << endl;
    cout << hex << (int)(*(p+3)) << endl;
    return 0;
}
```

Microsoft

79
e9
f6
42

d:低位在前存放	
2000	0x79
2001	0xe9
2002	0xf6
2003	0x42

上例解读：单精度浮点数123.456，在内存中占四个字节，四个字节的值依次为0x42 0xf6 0xe9 0x79（按打印顺序逆向取）

转换为32bit则为：0100 0010 | 1111 0110 | 1110 1001 | 0111 1001

8位指数　　23位尾数

§.基础知识题 - IEEE754的理解

double型数据的内部存储格式

```
#include <iostream>
using namespace std;
int main()
{
    double d = 1.23e4;
    unsigned char* p = (unsigned char*)&d;
    cout << hex << (int)(*p)      << endl;
    cout << hex << (int)(*(p+1)) << endl;
    cout << hex << (int)(*(p+2)) << endl;
    cout << hex << (int)(*(p+3)) << endl;
    cout << hex << (int)(*(p+4)) << endl;
    cout << hex << (int)(*(p+5)) << endl;
    cout << hex << (int)(*(p+6)) << endl;
    cout << hex << (int)(*(p+7)) << endl;
    return 0;
}
```

Microsoft

0
0
0
0
0
6
c8
40

d:低位在前存放	
2000	0x00
2001	0x00
2002	0x00
2003	0x00
2004	0x00
2005	0x06
2006	0xc8
2007	0x40

上例解读：双精度浮点数1.23e4，在内存中占八个字节，八个字节的值依次为0x40 0xc8 0x06 0x00 0x00 0x00 0x00 0x00(逆向)

转换为64bit则为：0100 0000 | 1100 1000 | 0000 0100 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000 | 0000 0000

11位指数　　52位尾数

§.基础知识题 - IEEE754的理解

1.float型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

A.2250758.8570522

注：尾数为正、指数为正

(1) 得到的32bit的机内表示是：0100 1010 0000 1001 0110 0000 0001 1011

(2) 其中：符号位是0

指数是1001 0100(填32bit中的原始形式)
指数转换为十进制形式是148(32bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是21(32bit中的原始形式按IEEE754的规则转换)

尾数是000 1001 0110 0000 0001 1011(填32bit中的原始形式)
尾数转换为十进制小数形式是0.07324540615081787109375(32bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.07324540615081787109375(加整数部分的1)

§.基础知识题 - IEEE754的理解

1.float型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

B.-8570522.2250758

注：尾数为负、指数为正

(1) 得到的32bit的机内表示是：1100 1011 0000 0010 1100 0110 1001 1010

(2) 其中：符号位是1

指数是1001 0110(填32bit中的原始形式)
指数转换为十进制形式是150(32bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是23(32bit中的原始形式按IEEE754的规则转换)

尾数是000 0010 1100 0110 1001 1010(填32bit中的原始形式)
尾数转换为十进制小数形式是0.021685838699340820312 5(32bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.021685838699340820312 5(加整数部分的1)

§.基础知识题 - IEEE754的理解

1.float型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

C.0.002250758

注：尾数为正、指数为负

(1) 得到的32bit的机内表示是：0011 1011 0001 0011 1000 0001 0111 0100

(2) 其中：符号位是0

指数是0111 0110(填32bit中的原始形式)
指数转换为十进制形式是118(32bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是-9(32bit中的原始形式按IEEE754的规则转换)

尾数是001 0011 1000 0001 0111 0100(填32bit中的原始形式)
尾数转换为十进制小数形式是0.15238809585571289062 5(32bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.15238809585571289062 5(加整数部分的1)

§.基础知识题 - IEEE754的理解

1.float型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

D.-0.008570522

注：尾数为负、指数为负

(1) 得到的32bit的机内表示是：1011 1100 0000 1100 0110 1011 0110 0000

(2) 其中：符号位是1

指数是0111 1000(填32bit中的原始形式)
指数转换为十进制形式是120(32bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是-7(32bit中的原始形式按IEEE754的规则转换)

尾数是000 1100 0110 1011 0110 0000(填32bit中的原始形式)
尾数转换为十进制小数形式是0.097026824951171875(32bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.097026824951171875(加整数部分的1)

§.基础知识题 - IEEE754的理解

2.double型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

A.2250758.8570522

注：尾数为正、指数为正

(1) 得到的64bit的机内表示是：0100 0001 0100 0001 0010 1100 0000 0011 0110 1101 1011 0011 1110 0010 1111 0001

(2) 其中：符号位是0

指数是1000 0010 100(填64bit中的原始形式)
指数转换为十进制形式是1044(64bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是21(64bit中的原始形式按IEEE754的规则转换)

尾数是0001 0010 1100 0000 0011 0110 1101 1011 0011 1110 0010 1111 0001(填64bit中的原始形式)
尾数转换为十进制小数形式是0.0732454571972847(64bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.0732454571972847(加整数部分的1)

§.基础知识题 - IEEE754的理解

2.double型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

B.-8570522.2250758

注：尾数为负、指数为正

(1) 得到的64bit的机内表示是：1100 0001 0110 0000 0101 1000 1101 0011 0100 0111 0011 0011 1101 0010 0010 1010

(2) 其中：符号位是1

指数是1000 0010 110(填64bit中的原始形式)
指数转换为十进制形式是1046(64bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是23(64bit中的原始形式按IEEE754的规则转换)

尾数是0000 0101 1000 1101 0011 0100 0111 0011 0011 1101 0010 0010 1010(填64bit中的原始形式)
尾数转换为十进制小数形式是0.02168586553046703(64bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.02168586553046703(加整数部分的1)

§.基础知识题 - IEEE754的理解

2.double型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

C.0.002250758

注：尾数为正、指数为负

(1) 得到的64bit的机内表示是：0011 1111 0110 0010 0111 0000 0010 1110 1000 0000 0000 1001 1110 1010 0101 0011

(2) 其中：符号位是0

指数是0111 1110 110(填64bit中的原始形式)
指数转换为十进制形式是1014(64bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是-9(64bit中的原始形式按IEEE754的规则转换)

尾数是0010 0111 0000 0010 1110 1000 0000 0000 1001 1110 1010 0101 0011(填64bit中的原始形式)
尾数转换为十进制小数形式是0.15238809599999992(64bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.15238809599999992(加整数部分的1)

§.基础知识题 - IEEE754的理解

2.double型数的机内表示

格式要求：多字节时，每4bit中间加一个空格或-(例："1101 0100 0011 0001" 或 "1101-0100-0011-0001")

D.-0.008570522

注：尾数为负、指数为负

(1) 得到的64bit的机内表示是：1011 1111 1000 0001 1000 1101 0110 1011 1111 1101 1001 1000 1110 0001 0101 0010

(2) 其中：符号位是1

指数是0111 1111 000(填64bit中的原始形式)
指数转换为十进制形式是1016(64bit中的原始形式按二进制原码形式转换)
指数表示的十进制形式是-7(64bit中的原始形式按IEEE754的规则转换)

尾数是0001 1000 1101 0110 1011 1111 1101 1001 1000 1110 0001 0101 0010(填64bit中的原始形式)
尾数转换为十进制小数形式是0.09702681600000007(64bit中的原始形式按二进制原码形式转换)
尾数表示的十进制小数形式是1.09702681600000007(加整数部分的1)

§.基础知识题 - IEEE754的理解

3.总结

(1) ①float型数据的32bit是如何分段来表示一个单精度的浮点数的？给出bit位的分段解释。

float型数据在内存中的存储分为三部分，分别是符号位、指数部分和尾数部分。

②尾数的正负如何表示?

尾数的正负由符号位表示，符号位为0表示该double型数据为正数，为1表示为负数。

③尾数如何表示?

把十进制浮点数表示为二进制形式，把这个二进制数转换为以2为底的指数形式，把小数点放在左起第一位和第二位之间，且第一位需要是个非0数。尾数部分的最高位始终为1，所以隐藏高位的整数部分1，低位补0，补齐23位，如果小数部分无限循环，则四舍五入为23位，这23位二进制数即为尾数在32bit中的原始形式。

④指数的正负如何表示？指数如何表示?

8位二进制数是指数在32bit中的原始形式，按二进制原码形式转换，可以得到十进制形式的256个非负整数(0~255)，每个非负整数按IEEE754的规则转换，减去偏移量(127)，就得到了指数，范围是-127~128，于是指数的正负便表示出来了。同时，以2为底的指数通过这种规则也表示出来了。

§.基础知识题 - IEEE754的理解

3.总结

(2) ①为什么float型数据只有7位十进制有效数字？为什么最大只能是$3.4x10^{38}$?

float型数据在内存中的存储分为三部分，分别是1位符号位、8位指数位和23位尾数位（实际上尾数位为24位，因为在尾数位前还隐藏了一个整数部分1或0）。符号位用于控制正负号，指数位用于控制小数点的移动，尾数位用于控制精度（或者说记录状态）。在24位尾数中能精确记录2^24=16777216种不同的状态，即float型数据最多只能存储16777216种十进制状态。16777216是8位数，所以float型数据的精度最多是7位十进制有效数字（0~9999999），共10000000种状态。如果float型数据的精度是8位十进制（0~99999999），则共100000000种状态，大于了float型数据能存储的状态上限16777216。所以说float型数据的精度不能达到8位十进制有效数字，只有7位十进制有效数字。

根据IEEE754，按照尾数位隐藏的整数部分是1还是0，可以将浮点数划分为normal number和subnormal number，当指数位全部被1填充时，这个浮点数为non-number，即这个数可能是±infinity或NaN。

当float型数据的指数位为00000000时，对应的数为subnormal number；当float型数据的指数位在00000001到11111110之间（转换为十进制为1~254，减去偏移量127为-126~127）时，对应的数为normal number；当float型数据的指数位为11111111时，对应的数为non-number。讨论浮点数的取值范围实际上讨论的是normal number的取值范围。float型数据的指数部分无法取到-127和128，因为用于表示-127的00000000被用来表示subnormal number了，用于表示128的11111111被用来表示non-number了。所以float型数据的指数部分的取值范围为[-126,127]。

尾数（含隐藏的整数部分）所表示的值的范围其实是[1.00...00, 1.11...11]，即十进制形式的[1,2)。

float型数据的取值范围=±尾数×2^指数=±[1,2)×$2^{[-126,127]}$，把以2为底替换为以10为底，得到float型数据的取值范围为$(-3.402823669...\times10^{38},-1.17549435...\times10^{-38}]\cup[1.17549435...\times10^{-38},3.402823669...\times10^{38})$，从上面这个集合中取一个更容易表示的子集$[-3.4\times10^{38},-1.18\times10^{-38}]\cup[1.18\times10^{-38},3.4\times10^{38}]$就是float型数据的取值范围。之所以能写成闭区间的形式是因为它是真正取值范围的一个子集。所以float型数据最大只能是$3.4x10^{38}$。

§.基础知识题 - IEEE754的理解

3.总结

(2) ②有些资料上说有效位数是6~7位，能找出6位/7位不同的例子吗?

float型数据的尾数部分是23bit，2^23=8388608，故float型数据能表示的最大数为8388608，而大于8388608的7位整数，如9999999，float型数据便不能表示，这就决定了float型数据的有效位数不能是7位，但是绝对能保证精度的有效位数是6位，所以说float型数据的有效位数是6~7位。不超过8388608的float型数据是精度保证的，大于8388608而小于9999999的float型数据则是精度不保证的。

6位/7位不同的例子如下：

My Project.cpp

My Project

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float f = 9.111111;
    cout << setprecision(8) << f << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
9.1111107
```

§.基础知识题 - IEEE754的理解

3.总结

(3) ① double型数据的64bit是如何分段来表示一个双精度的浮点数的？给出bit位的分段解释。

double型数据在内存中的存储分为三部分，分别是符号位、指数部分和尾数部分。

②尾数的正负如何表示?

尾数的正负由符号位表示，符号位为0表示该double型数据为正数，为1表示为负数。

③尾数如何表示?

把十进制浮点数表示为二进制形式，把这个二进制数转换为以2为底的指数形式，把小数点放在左起第一位和第二位之间，且第一位需要是个非0数。尾数部分的最高位始终为1，所以隐藏高位的整数部分1，低位补0，补齐52位，如果小数部分无限循环，则四舍五入为52位，这52位二进制数即为尾数在64bit中的原始形式。

④指数的正负如何表示？指数如何表示?

11位二进制数是指数在64bit中的原始形式，按二进制原码形式转换，可以得到十进制形式的2048个非负整数(0~2047)，每个非负整数按IEEE754的规则转换，减去偏移量(1023)，就得到了指数，范围是-1023~1024，于是指数的正负便表示出来了。同时，以2为底的指数通过这种规则也表示出来了。

§.基础知识题 - IEEE754的理解

3.总结

(4) ①为什么double型数据只有15位十进制有效数字？为什么最大只能是$1.7x10^{308}$？

double型数据在内存中的存储分为三部分，分别是1位符号位、11位指数位和52位尾数位（实际上尾数位为53位，因为在尾数位前还隐藏了一个整数部分1或0）。符号位用于控制正负号，指数位用于控制小数点的移动，尾数位用于控制精度（或者说记录状态）。在53位尾数中能精确记录2^53=9007199254740992种不同的状态，即double型数据最多只能存储9007199254740992种十进制状态。9007199254740992是16位数，所以double型数据的精度最多是15位十进制有效数字（0~999999999999999），共1000000000000000种状态。如果double型数据的精度是16位十进制（0~9999999999999999），则共10000000000000000种状态，大于了double型数据能存储的状态上限9007199254740992。所以说double型数据的精度不能达到16位十进制有效数字，只有15位十进制有效数字。

根据IEEE754，按照尾数位隐藏的整数部分是1还是0，可以将浮点数划分为normal number和subnormal number，当指数位全部被1填充时，这个浮点数为non-number，即这个数可能是±infinity或NaN。

当double型数据的指数位为00000000000时，对应的数为subnormal number；当double型数据的指数位在00000000001到11111111110之间（转换为十进制为1~2046，减去偏移量1023为-1022~1023）时，对应的数为normal number；当double型数据的指数位为11111111111时，对应的数为non-number。讨论浮点数的取值范围实际上讨论的是normal number的取值范围。 double型数据的指数部分无法取到-1023和1024，因为用于表示-1023的00000000000被用来表示subnormal number了，用于表示1024的11111111111被用来表示non-number了。所以double型数据的指数部分的取值范围为[-1022,1023]。

尾数（含隐藏的整数部分）所表示的值的范围其实是[1.00...00, 1.11...11]，即十进制形式的[1,2)。

double型数据的取值范围=±尾数×2^指数=±[1,2)×$2^{[-1022,1023]}$，把以2为底替换为以10为底，得到double型数据的取值范围为$(-1.797693134...\times10^{308},-4.940656458...\times10^{-324}]\cup[4.940656458...\times10^{-324},1.797693134...\times10^{308})$，从上面这个集合中取一个更容易表示的子集$[-1.7\times10^{308},-5.0\times10^{-324}]\cup[5.0\times10^{-324},1.7\times10^{308}]$就是double型数据的取值范围。之所以能写成闭区间的形式是因为它是真正取值范围的一个子集。所以double型数据最大只能是$1.7x10^{308}$。

§.基础知识题 - IEEE754的理解

3.总结

(4) ②有些资料上说有效位数是15~16位，能找出15位/16位不同的例子吗?

double型数据的尾数部分是52bit，2^52=4503599627370496，故double型数据能表示的最大数为4503599627370496，而大于4503599627370496的16位整数，如9999999999999999，double型数据便不能表示，这就决定了double型数据的有效位数不能是16位，但是绝对能保证精度的有效位数是15位，所以说double型数据的有效位数是15~16位。不超过4503599627370496的double型数据是精度保证的，大于4503599627370496而小于9999999999999999的double型数据则是精度不保证的。

15位/16位不同的例子如下：

My Project.cpp

My Project

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double f = 9.111111111111111;
    cout << setprecision(17) << f << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
9.1111111111111107
```

§.基础知识题 - IEEE754的理解

4.思考

(1) 8/11bit的指数的表示形式是2进制补码吗？如果不是，一般称为什么方式表示？

不是。二进制原码。

(2) double赋值给float时，下面两个程序，double型常量不加F的情况下，左侧有warning，右侧无warning，为什么？

总结一下规律。

```
#include <iostream>
using namespace std;
int main()
{
    float f = 1.2;
    unsigned char* p = (unsigned char*)&f;
    cout << hex << (int)(*p)      << endl;
    cout << hex << (int)(*(p+1)) << endl;
    cout << hex << (int)(*(p+2)) << endl;
    cout << hex << (int)(*(p+3)) << endl;
    return 0;
}
```

warning C4305: “初始化”：从“double”到“float”截断

```
#include <iostream>
using namespace std;
int main()
{
    float f = 100.25;
    unsigned char* p = (unsigned char*)&f;
    cout << hex << (int)(*p)      << endl;
    cout << hex << (int)(*(p+1)) << endl;
    cout << hex << (int)(*(p+2)) << endl;
    cout << hex << (int)(*(p+3)) << endl;
    return 0;
}
```

左侧有warning的原因：double型数据1.2在赋值给float进行机内存储时，小数部分0.2在转换为二进制形式时，小数部分无限循环，需要四舍五入为23位，此处体现出误差，所以编译器显示warning C4305: “初始化”：从“double”到“float”截断。

右侧无warning的原因：double型数据100.25在赋值给float进行机内存储时，小数部分0.25在转换为二进制形式时，小数部分需要低位补0，补齐23位，此处未体现出误差，所以无warning。

规律总结：double型数据在赋值给float时，若数据被截断，体现出误差，则有warning；若数据未被截断，未体现出误差，则无warning。

§.基础知识题 – 关系运算、逻辑运算与选择结构

1.关系运算符的求值顺序

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int a=1, b=2, c=3, d;

    d = a > b > c;
    cout << d << endl;

    d = a > b < c;
    cout << d << endl;

    d = a > b < c;
    cout << d << endl;

    return 0;
}
```

1、贴运行结果

2、VS下为什么会有三个warning？说说你的理解

根据运算符优先级，>和<为优先级第8组，左结合，=为优先级第16组，右结合，a > b会返回bool类型，其值为0/1，再与c进行比较，因此连续比较是不安全的。所以会显示warning C4804: “>”: 在操作中使用类型“bool”不安全。

§.基础知识题 – 关系运算、逻辑运算与选择结构

1.关系运算符的求值顺序

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int a=3, b=2, c=1, d;

    d = a > b > c;
    cout << d << endl;

    d = a < b < c;
    cout << d << endl;

    d = b > a < c;
    cout << d << endl;

    return 0;
}
```

1、贴运行结果

2、(1)a > b > c这个式子，按常规理解，3>2>1是正确的，为什么结果是0？

a > b为真，返回bool类型，其值为1，1 > c为假，返回bool类型，其值为0，0赋值给整型变量d，所以结果是0。

(2)a < b < c这个式子，按常规理解，3<2<1是错误的，为什么结果是1？

a < b为假，返回bool类型，其值为0，0 < c为真，返回bool类型，其值为1，1赋值给整型变量d，所以结果是1。

(3)b > a < c这个式子，按常规理解，2>3<1是错误的，为什么结果是1？

b > a为假，返回bool类型，其值为0，0 < c为真，返回bool类型，其值为1，1赋值给整型变量d，所以结果是1。

§.基础知识题 – 关系运算、逻辑运算与选择结构

2.关系运算符与实数

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    float f1 = 100.25;
    cout << (f1 - 100.25) << endl;
    cout << (f1 == 100.25) << endl;
    cout << (fabs(f1-100.25) < 1e-6) << endl;

    float f2 = 1.2;
    cout << (f2 - 1.2) << endl;
    cout << (f2 == 1.2) << endl;
    cout << (fabs(f2-1.2) < 1e-6) << endl;

    return 0;
}
```

1、贴VS+Dev下的运行结果

Microsoft Visual Studio 调试控制台

```
0
1
1
4.76837e-08
0
1
```

D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe

```
0
1
1
4.76837e-08
0
1
```

warning C4305: “初始化”: 从“double”到“float”截断

2、删除第2行的#include<cmath>后，再次贴VS+Dev的运行结果

Microsoft Visual Studio 调试控制台

```
0
1
1
4.76837e-08
0
1
```

warning C4305: “初始化”: 从“double”到“float”截断

[Error] 'fabs' was not declared in this scope; did you mean 'labs'?

3、由本例得出的结论，实数进行相等比较时的通用方法是当<u>两数相减的绝对值小于某个值则认为相等</u>。

§.基础知识题 – 关系运算、逻辑运算与选择结构

2.关系运算符与实数

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
#include <cmath> //VS可不加
using namespace std;

int main()
{
    double d1=123.456789012345678;
    double d2=123.456789123456789;
    cout << (d1==d2) << endl;
    cout << (fabs(d1-d2)<1e-6) << endl;
    cout << (fabs(d1-d2)<1e-7) << endl;

    float f1=123.456789012345678;
    float f2=123.456789123456789;
    cout << (f1==f2) << endl;
    cout << (fabs(f1-f2)<1e-6) << endl;
    cout << (fabs(f1-f2)<1e-7) << endl;

    return 0;
} //VS有两个warning
```

1、贴运行结果

2、观察 fabs(**)<1e-6 和 fabs(**)<1e-7 在float和double下的表现，哪个相同？哪个不同？为什么？

fabs(**)<1e-6 和 fabs(**)<1e-7 在float下的表现相同；

fabs(**)<1e-6 和 fabs(**)<1e-7 在double下的表现不同；

从“double”到“float”截断，超出float型有效位数的数据根据IEEE754在机内存储，小数点后第7位在机内的存储是相同的，所以fabs(**)<1e-6 和 fabs(**)<1e-7 在float下的表现相同；double型的有效位数比float型多，小数点后第7位在机内的存储是不同的，所以fabs(**)<1e-6 和 fabs(**)<1e-7 在double下的表现不同。

§.基础知识题 – 关系运算、逻辑运算与选择结构

3.逻辑常量与逻辑变量

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    cout << true    << endl;
    cout << "true"  << endl;

    cout << endl;

    cout << false   << endl;
    cout << "false" << endl;

    return 0;
}
```

1、贴运行结果

2、解释 true 和 "true" 的区别（false和"false"）

true是逻辑常量，数据类型为逻辑变量bool类型；"true"是字符串常量；false是逻辑常量，数据类型为逻辑变量bool类型；"false"是字符串常量。

3、进阶思考：目前直接输出逻辑常量true和false，在屏幕上的输出是1/0，如果想输出为true/false，应该怎么做？

使用前导格式控制符boolalpha。

§.基础知识题 – 关系运算、逻辑运算与选择结构

3.逻辑常量与逻辑变量

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    bool k1 = true;

    cout << sizeof(true) << endl;
    cout << sizeof(k1) << endl;
    cout << k1 << ' ' << int(k1) << endl;

    cout << endl;

    bool k2 = false;
    cout << sizeof(false) << endl;
    cout << sizeof(k1) << endl;
    cout << k2 << ' ' << int(k2) << endl;

    return 0;
}
```

1、贴运行结果

2、bool型常量/变量在内存中占用1字节，
值是0(false)/1(true)

总结bool型常量/变量在输出时的规则
（限制：在无3.A的前导格式控制符的前提下）
bool型常量/变量在输出时按整型量进行处理，屏幕上的输出是1/0，而不是true/false。

§.基础知识题 – 关系运算、逻辑运算与选择结构

3.逻辑常量与逻辑变量

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    bool k;

    cin >> k;
    cout << k << ' ' << int(k) << endl;

    return 0;
}
```

1、输入0，输出是：

2、输入1，输出是：

3、输入123，输出是：

4、输入true，输出是：

5、输入false，输出是：

总结bool型变量在输入时的规则：
cin时只能输入0/1，否则得到不可信值。

§.基础知识题 – 关系运算、逻辑运算与选择结构

3.逻辑常量与逻辑变量

D.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    bool k;

    k='A';
    cout << k << ' ' << (int)k << endl;

    k=0;
    cout << k << ' ' << (int)k << endl;

    k=256;
    cout << k << ' ' << (int)k << endl;

    return 0;
}
```

1、贴运行结果

```
warning C4305: "=": 从"char"到"bool"截断
warning C4305: "=": 从"int"到"bool"截断
```

```
Microsoft Visual Studio 调试控制台
1 1
0 0
1 1
```

2、解释VS下waring的意思

从多字节变量赋值给取值只有0/1的bool类型少字节变量时会发出此警告，数据被截断，这可能将导致信息丢失。

3、k='A'是1字节赋值给1字节，为什么还有warning?

A的ASCII码为65，k='A'代表将65赋值给值只有0/1的bool类型，数据被截断，所以会显示warning。

4、k=256如果按整型的4字节赋给1字节，k应该是多少？现在实际是多少？为什么？

k=256如果按整型的4字节赋给1字节，k应该是0，现在实际是1，因为bool赋值时不是多字节赋少字节的规则，而是"非0为真0为假"。

5、"非0为真0为假"这句话如何解释？

非0值在赋值给bool类型时值为真(ture)，取值为1，0值在赋值给bool类型时值为假(false)，取值为0

§.基础知识题 – 关系运算、逻辑运算与选择结构

3.逻辑常量与逻辑变量

E.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    bool f=true;
    int  a=10;

    a=a+f;
    cout << a << endl;

    return 0;
}
```

1、贴运行结果

2、当bool参与表达式计算时，当做<u>整型值，即0/1</u>。

§.基础知识题 – 关系运算、逻辑运算与选择结构

4. 逻辑运算符与逻辑运算

A. 完成下列两个表格的填写

a	b	!a	!b	a&&b	a||b
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

a	b	!a	!b	a&&b	a||b
非0	非0	0	0	1	1
非0	0	0	1	0	1
0	非0	1	0	0	1
0	0	1	1	0	0

§.基础知识题 – 关系运算、逻辑运算与选择结构

4.逻辑运算符与逻辑运算

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int a=1, b=2, c=3, d=4, m=1, n=1;

    cout << "m=" << m << " n=" << n << endl;
    (m=a>b)&&(n=c>d);
    cout << "m=" << m << " n=" << n << endl;

    return 0;
}
```

1、贴运行结果

2、解释(m=a>b)&&(n=c>d)的求值过程（标出步骤顺序）

① a>b

② m=a>b

③ (m=a>b)&&(n=c>d)

3、短路运算的意思是：<u>仅当必须执行下一个逻辑运算符才能求出解时，才执行该运算符，否则不执行</u>

§. 基础知识题 – 关系运算、逻辑运算与选择结构

4. 逻辑运算符与逻辑运算

C. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int m = 1;
    cout << (8 < 4 - !0) << endl;
    5 > 3 && 2 || (m = 8 < 4 - !0);
    cout << m << endl;

    return 0;
}
```

有以下逻辑表达式

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

1、构造一个测试程序，在不改变该表达式目前求值顺序的情况下（允许插入新的运算，但目前这几个运算符的顺序不要变），证明 8<4 - !0 存在短路运算

2、用栈方式画包含短路运算的表达式，则从分析到短路运算符进栈开始（本例中为||），忽略比||优先级高的运算符。

初始：两栈均为空

5进栈

>进栈

3进栈

要进栈的(&&)低于栈顶的(>)，先计算

&&进栈

2进栈

要进栈的(||)低于栈顶的(&&)，先计算（5>3 && 2 记作①）

①的值为1，要进栈的运算符为||，此时该表达式的值必为1，后面的运算符不再求解

§.基础知识题 – 关系运算、逻辑运算与选择结构

5.if语句 – 基本使用

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int i;

    cout<<"请输入成绩[0-100]"<<endl;
    cin >> i;

    if (i<60) {
        cout << "不及格" << endl;
        }
    cout << "程序结束" << endl;

    return 0;
}
```

1、输入34，贴运行结果

2、输入74，贴运行结果

3、画出程序对应的流程框图

§.基础知识题 – 关系运算、逻辑运算与选择结构

5.if语句 – 基本使用

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int i;

    cout<<"请输入成绩[0-100]"<<endl;
    cin >> i;

    if (i<60) {
        cout << "不及格" << endl;
    cout << "程序结束" << endl; //未缩进
    }

    return 0;
}
```

1、输入34，贴运行结果

2、输入74，贴运行结果

3、画出程序对应的流程框图

4、程序标注"未缩进"的行，<u>应该</u>缩进

§.基础知识题 – 关系运算、逻辑运算与选择结构

5.if语句 – 基本使用

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
   int i;

   cout<<"请输入成绩[0-100]"<<endl;
   cin >> i;

   if (i<60;) {
      cout << "不及格" << endl;
   cout << "程序结束" << endl; //未缩进
   }

   return 0;
}
```

错误信息截图：

```
warning C4552: "<": 未使用表达式结果
error C2429: 语言功能 "if/switch 中的 init-statement" 需要编译器标志 "/std:c++17"
error C2059: 语法错误: ")"
error C2143: 语法错误: 缺少 ";" (在 "{" 的前面)
```

错误原因：

if语句条件表达式后无分号，若在条件表达式后加分号，则不符合语法规则。整个if语句可以作为一个语句来看待，若在条件表达式后加分号，编译器会认为该语句已结束，所以会出现warning C4552、error C2429、error C2059和error C2143的提示。

§.基础知识题 – 关系运算、逻辑运算与选择结构

5.if语句 – 基本使用

D.观察下列程序的运行结果，回答问题

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    cout << "请输入成绩[0-100]" << endl;
    cin >> i;
    if (i>=90 && i<=100)
        cout << "优" << endl;
    else if (i>=80 && i<90)
        cout << "良" << endl;
    else if (i>=70 && i<80)
        cout << "中" << endl;
    else if (i>=60 && i<70)
        cout << "及格" << endl;
    else if (i>=0 && i<60)
        cout << "不及格" << endl;
    else
        cout << "输入错误" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

1、给出程序的流程框图

2、i<90能否改为i<=89？哪个更好？

因为本程序中的成绩为int型数据，故i<90能改为i<=89。i<90更好。

3、i<90能否改为i<=90？运行是否正确？

能。正确。

§.基础知识题 - 关系运算、逻辑运算与选择结构

6.if语句 - 多重嵌套

A.一个有10行代码的if语句嵌套，回答问题

0：if（表达式）{
1：if（表达式）{
2：}
3：else {
4：}
5：}
6：else {
7：if（表达式）{
8：}
9：}
	第0行的"{"和第 5 行的"}"配对
	
	第1行的"{"和第 2 行的"}"配对
	
	第3行的"{"和第 4 行的"}"配对
	
	第6行的"{"和第 9 行的"}"配对
	
	第7行的"{"和第 8 行的"}"配对
	
	总结：给出大括号配对的基本准则
	自上而下，忽略 { ，以 } 为准向上匹配未配对的 { 。{}的匹配可用栈理解，遇 { 进栈，遇 } 则栈顶 { 出栈并匹配为一对，若到最后仍有 { 或 } 未配对则语法错。

§.基础知识题 – 关系运算、逻辑运算与选择结构

6.if语句 – 多重嵌套

B.一个if语句嵌套如下，回答问题

if (表达式1) {
if (表达式2) {
A;
}
B;
}
	1、当表达式1 真 ，表达式2 真 时，执行语句A
	
	2、当表达式1 真 ，表达式2 任意 时，执行语句B

§.基础知识题 – 关系运算、逻辑运算与选择结构

6.if语句 – 多重嵌套

C.一个if语句嵌套如下，回答问题

```
if (表达式1) {
    if (表达式2) {
        A;
        }
    else {
        B;
        }
    C;
    }
else {
    if (表达式3) {
        D;
        }
    E;
    }
```

1、当表达式1 真 ，表达式2 真 时，执行语句A

2、当表达式1 真 ，表达式2 假 时，执行语句B

3、当表达式1 真 ，表达式2 任意 时，执行语句C

4、当表达式1 假 ，表达式3 真 时，执行语句D

5、当表达式1 假 ，表达式3 任意 时，执行语句E

§.基础知识题 – 关系运算、逻辑运算与选择结构

6.if语句 – 多重嵌套

D.一个if语句嵌套如下，回答问题

```
if (表达式1) {
    if (表达式2) {
        A;
        }
    else {
        B;
        }
    C;
    }
F;
else {
    if (表达式3) {
        D;
        }
    E;
    }
```

在6.C的基础上，在箭头位置插入语句F

1、请构造一个符合此要求的测试程序，并给出该程序的程序及编译错误截图

2、请说明错误原因

整个双支语句可以作为一个语句来看待，中间不允许插入任何的其他语句。因为从编译器的思维角度，从上到下看，单if、并列的cout，因此else非法，会显示 error C2181: 没有匹配if的非法else。

error C2181: 没有匹配 if 的非法 else

My Project.cpp

My Project

```cpp
#include<iostream>
using namespace std;

int main()
{
    int n;
    cin >> n;

    if (n > 0) {
        if (n > 10) {
            cout << "A" << endl;
        }
        else {
            cout << "B" << endl;
        }
        cout << "C" << endl;
    }
    cout << "F" << endl;
    else {
        if (n < -10) {
            cout << "D" << endl;
        }
        cout << "E" << endl;
    }

    return 0;
}
```

§.基础知识题 – 关系运算、逻辑运算与选择结构

6.if语句 – 多重嵌套

E.一个if语句嵌套如下，回答问题

if (表达式1) {
if (表达式2) {
A;
}
B;
}
else {
C;
}
	左侧代码按缩进格式排版
	if (表达式1) {
	  if (表达式2) {
	  A;
	  }
	  B;
	}
	else {
	  C;
	}
		1、当表达式1 真 ，表达式2 真 时，执行语句A
		
		2、当表达式1 真 ，表达式2 任意 时，执行语句B
		
		3、当表达式1 假 ，表达式2 任意 时，执行语句C

if (表达式1) {
if (表达式2) {
A;
}
else {
B;
}
C;
}
	左侧代码按缩进格式排版
	if (表达式1) {
	  if (表达式2) {
	  A;
	  }
	  else {
	  B;
	  }
	  C;
	}
		1、当表达式1 真 ，表达式2 真 时，执行语句A
		
		2、当表达式1 真 ，表达式2 假 时，执行语句B
		
		3、当表达式1 真 ，表达式2 任意 时，执行语句C

§.基础知识题 – 关系运算、逻辑运算与选择结构

7.条件运算符与条件表达式

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int a,b;

    cin >> a >> b;

    if (a>b)
        cout << "max=" << a << endl;
    else
        cout << "max=" << b << endl;

    a > b ? cout << "max=" << a << endl : cout << "max=" << b << endl; //1

    cout << "max=" << (a>b?a:b) << endl; //2

    printf("max=%d", a>b?a:b); //3

   return 0;
}
```

1、输入12 34，给出运行截图

2、输入34 12，给出运行截图

3、//1 //2 //3这三种条件运算符的使用，按你的喜欢程度排序为<u>//2、//1、//3</u>

§.基础知识题 – 关系运算、逻辑运算与选择结构

7.条件运算符与条件表达式

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int main()
{
    int a=1, b=2;

    a==1 ? "Hello" : 123;                //编译报错

    a>b ? cout << a : printf("%d", b);  //编译报错

    a==1 ? 'A' : 123;                    //编译正确

    return 0;
}
```

1、给出编译报错的截图

```
error C2446: ":": 没有从"int"到"const char [6]"的转换
message : 从整型类型转换为指针类型需要 reinterpret_cast、C 样式转换或带圆括号的函数样式强制转换
 error C2678: 二进制"?": 没有找到接受"std::basic_ostream<char,std::char_traits<char>>"类型的左操作数的运算符(或没有可接受的转换)
 message : 可以是"内置 C++ operator?(int, int)"
 message : 尝试匹配参数列表"(std::basic_ostream<char,std::char_traits<char>>, int)"时
```

2、条件表达式使用的三句中，前两句报错，最后一句正确，总结下条件表达式使用时的限制规则

(提示：注意表达式2和表达式3的类型)

表达式1、2、3的类型可以不同，但2、3的类型必须相容。

§.基础知识题 - 关系运算、逻辑运算与选择结构

8. switch-case语句

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }

    return 0;
}
```

程序的期望，是当输入的score在[0..100]时，分段输出"优/良/中/及格/不及格"，否则输出"输入错误"

1、程序不完全正确，找出不符合期望的两个数据区间并给出运行截图
(不需要改对)
[101, 109]和[-9, -1]

§.基础知识题 - 关系运算、逻辑运算与选择结构

8.switch-case语句

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    const int k=5;
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        case k+2:
            cout<<"中"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        }

    return 0;
}
```

在8.A的基础上

1、将6、8、default的位置进行了交换

2、将7写为常变量+常量形式

验证此程序与8.A的功能是否完全一致

（即：8.A中正确的,此程序中同样正确；8.A错误的，此程序中同样错误）

结论：8.A和8.B<u>完全一致</u>

§.基础知识题 – 关系运算、逻辑运算与选择结构

8.switch-case语句

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int k=5;
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        case k+2:
            cout<<"中"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        }

    return 0;
}
```

在8.B的基础上，将k从const int改为int

1、给出编译错误的截图

```
error C2131: 表达式的计算结果不是常数
message : 因读取超过生命周期的变量而失败
message : 请参见“k”的用法
error C2051: case 表达式不是常量
```

2、解释错误原因

case后面应该加整型常量表达式，不能加变量表达式，否则不符合语法规则。

§.基础知识题 – 关系运算、逻辑运算与选择结构

8.switch-case语句

D.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
        case 4+2:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }

    return 0;
}
```

在8.A的基础上，多了一个case 4+2

1、给出编译错误的截图

```
error C2196: case 值“6”已使用
```

2、解释错误原因

各整型常量表达式的值应各不相同。4+2的值为6，与6重复。

§.基础知识题 - 关系运算、逻辑运算与选择结构

8.switch-case语句

E.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    float score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }

    return 0;
}
```

在8.A的基础上，将score从int改为float

1、给出编译错误的截图

```
error C2450: 类型为“float”的 switch 表达式无效
message : 要求整型表达式
```

2、解释错误原因

表达式可以是任何类型，但最终取值应该为整型。score为float型数据，score/10的结果也为float型数据，最终取值不是整型。

§.基础知识题 – 关系运算、逻辑运算与选择结构

8.switch-case语句

F.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;

        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }

    return 0;
}
```

在8.A的基础上，删除case 8后面的break

1、给出与8.A运行结果不一致的测试数据即截图

2、解释break的作用

提前结束循环体的循环。

§.基础知识题 - 关系运算、逻辑运算与选择结构

8.switch-case语句

G.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout<<"请输入成绩[0-100]"<<endl;
    cin >> score;
    switch(score/10) {
        case 10:
        case 9:
            cout<<"优"<<endl;
            break;
        case 8:
            cout<<"良"<<endl;
            break;
        case 7:
            cout<<"中"<<endl;
            break;
        case 6:
            cout<<"及格"<<endl;
            break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0:
            cout<<"不及格"<<endl;
            break;
        default:
            cout<<"输入错误"<<endl;
            break;
        }

    return 0;
}
```

程序同8.A，将其改正确，即所有[0..100]之外的数据均给出"输入错误"即可

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "请输入成绩[0-100]" << endl;
    cin >> score;
    if (score >= 0 && score <= 100) {
        switch (score / 10) {
        case 10:
        case 9:
            cout << "优" << endl;
            break;
        case 8:
            cout << "良" << endl;
            break;
        case 7:
            cout << "中" << endl;
            break;
        case 6:
            cout << "及格" << endl;
            break;
        default:
            cout << "不及格" << endl;
            break;
        }
    }
    else
        cout << "输入错误" << endl;
    cout << "程序结束" << endl;
    return 0;
}
```

§. 基础知识题 – 关系运算、逻辑运算与选择结构

8. switch-case语句

H. 思考

如果将成绩区间对应为：
[85-100] – 优
[70-85) – 良
[60-70) – 及格
[0-60) – 不及格

1、用if-else语句完成该程序并贴图

2、如果用switch语句，该如何实现？

My Project.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "请输入成绩[0-100]" << endl;
    cin >> score;
    if (score >= 85 && score <= 100)
    {
        cout << "优" << endl;
    }
    else if (score >= 70 && score < 85)
    {
        cout << "良" << endl;
    }
    else if (score >= 60 && score < 70)
    {
        cout << "及格" << endl;
    }
    else if (score >= 0 && score < 60)
    {
        cout << "不及格" << endl;
    }
    else
    {
        cout << "输入错误" << endl;
    }
    cout << "程序结束" << endl;
    return 0;
}
```

My Project.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    int score;
    cout << "请输入成绩[0-100]" << endl;
    cin >> score;
    if (score >= 0 && score <= 100) {
        switch (score / 5) {
            case 20:
            case 19:
            case 18:
            case 17:
                cout << "优" << endl;
                break;
            case 16:
            case 15:
            case 14:
                cout << "良" << endl;
                break;
            case 13:
            case 12:
                cout << "及格" << endl;
                break;
            default:
                cout << "不及格" << endl;
                break;
        }
    }
    else {
        cout << "输入错误" << endl;
    }
    return 0;
}
```

§.基础知识题 – 关系运算、逻辑运算与选择结构

8.switch-case语句

H.思考

3、如果学生成绩带小数点，即"xx.5"形式，能用if语句吗？能用switch语句吗？请解释原因

能用if语句，不能用switch语句，因为switch语句表达式可以是任何类型，但最终取值应该为整型，"xx.5"形式为浮点型。

4、总结switch语句使用时的注意事项

(1)表达式可以是任何类型，最终取值为整型即可

(2)当整型表达式的取值与整型常量表达式1-n中的任意一个相等时，执行对应的语句序列；否则执行default后的语句序列（不能是实数，因为无法直接判断相等）

(3)各整型常量表达式的值应各不相同，但顺序无要求

(4)各语句序列的最后一句应是break，否则连续执行下一case语句，最后一个可省

(5)语句序列不必加{}

(6)多个case可以共用一组语句

(7)不能完全替代多重if语句的嵌套

5、switch-case语句能完全取代if-else吗？

不能。

§.基础知识题 – 循环结构

1.循环的嵌套

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    int i, j, k;
    int count1 = 0, count2 = 0, count3 = 0;

    for(i=1; i<=100; i++) {
        ++count1;
        for(j=1; j<=100; j++) {
            ++count2;
            for(k=1; k<=100; k++)
                ++count3;
            }
        }
    cout << "count1=" << count1 << endl;
    cout << "count2=" << count2 << endl;
    cout << "count3=" << count3 << endl;
    return 0;
}
```

1、贴运行结果

2、当循环嵌套时，内层循环的执行次数和外层循环是什么关系？

外层循环每执行一次，内层循环都要执行一遍。因此嵌套循环的总执行次数等于内层循环的执行次数与外层循环的执行次数的乘积。

§. 基础知识题 – 循环结构

1. 循环的嵌套

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    int i, j, k;
    int count1 = 0, count2 = 0, count3 = 0;

    for(i=1; i<=100; i++) {
        ++count1;
        for(j=i; j<=100; j++) {
            ++count2;
            for(k=j; k<=100; k++)
                ++count3;
            }
        }
    cout << "count1=" << count1 << endl;
    cout << "count2=" << count2 << endl;
    cout << "count3=" << count3 << endl;
    return 0;
}
```

1、贴运行结果

2、当循环嵌套时，内层循环的执行次数和外层循环是什么关系？

Microsoft Visual Studio 调试控制台

```
count1=100
count2=5050
count3=171700
```

第一层循环第i次执行	第二层循环的执行次数	第三层循环的执行次数				
1	100	5050	=	1+2+…+100	=	(1+100)*100/2
2	99	4950（比上次的执行次数少100）	=	1+2+…+99	=	(1+99)*99/2
3	98	4851（比上次的执行次数少99）	=	1+2+…+98	=	(1+98)*98/2
4	97	4753（比上次的执行次数少98）	=	1+2+…+97	=	(1+97)*97/2
5	96	4656（比上次的执行次数少97）	=	1+2+…+96	=	(1+96)*96/2
…	…	…	=	…	=	
i	101-i	（比上次的执行次数少100-i）	=	1+2+…+(101-i)	=	(1+101-i)*(101-i)/2
…	…	…	=	…	=	
96	5	15（比上次的执行次数少6）	=	1+2+3+4+5	=	(1+5)*5/2
97	4	10（比上次的执行次数少5）	=	1+2+3+4	=	(1+4)*4/2
98	3	6（比上次的执行次数少4）	=	1+2+3	=	(1+3)*3/2
99	2	3（比上次的执行次数少3）	=	1+2	=	1+2
100	1	1（比上次的执行次数少2）	=	1	=	1

第一层循环总执行次数	第二层循环总执行次数	第三层循环总执行次数
100	(1+100)*100/2 =5050	$\sum_{i=1}^{100}\frac{(1+101-i)(101-i)}{2}=171700$

关系：当第一层循环第i次执行时，第二层循环的执行次数为100-i，第三层循环的执行次数为(1+101-i)*(101-i)/2；当第一层循环第i次执行，第二层循环第j次执行时，第三层循环的执行次数为(101-i)-(j-1)。

§.基础知识题 – 循环结构

1.循环的嵌套

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    int i, j, count = 0;
    for(i=1; i<=100; i++) {
        for(j=1; i<=100; j++) {
            ++count;
            if (count % 1000 == 0) {
                cout << "*";
                _getch();
                }
            }
        }

    cout << "count = " << count << endl;
    return 0;
}
//注意：这个程序无法通过按CTRL+C终止，要关窗口
```

1、贴运行结果

2、按内外for循环的执行步骤依次分析，为什么会得到这个结果？

第 1 步 – 外循环表达式1 – i=1
第 2 步 – 外循环表达式2 – i<=100为真
第 3 步 – 内循环表达式1 – j=1
第 4 步 – 内循环表达式2 – i<=100为真
第 5 步 – ++count – count值为1
第 6 步 – 条件表达式 – count%1000==0为假
第 7 步 – j++ – j的值为2
第 8 步 – 内循环表达式2 – i<=100为真
第 9 步 – ++count – count值为2
第 10 步 – 条件表达式 – count%1000==0为假
…
第 n 步 – ++count – count值为1000
第n+1步 – 条件表达式 – count%1000==0为真
第n+2步 – cout << "*";
第n+3步 – _getch();
第n+3步 – j++ – j的值为1001
第n+4步 – 内循环表达式2 – i<=100为真
…
第 m 步 – ++count – count值为2000
第m+1步 – 条件表达式 – count%1000==0为真
第m+2步 – cout << "*";
第m+3步 – _getch();
…

原因分析：

内层for循环的表达式2为i<=100，在内层for循环中没有能改变表达式2取值的语句，因此内层for循环为永真循环。当count%1000==0为真时，执行if单支语句序列，输出一个 * ， 并 使 用 _getch()函数输入一个字符。但是由于_getch()函数无回显，不需要回车键且_getch()函数的返回值被忽略，因此造成了每次在键盘上输入一个字符，程序都会输出一个*，并且暂停等待输入下一个字符的效果。

§.基础知识题 - 循环结构

2.break与continue

A.已知代码如下，回答问题

```
while(1) {
    ①
    ②
    if (X)
        continue;
    ③
    ④
    }
```

当X为真时，重复执行①②
当X为假时，重复执行①②③④

```
for(1; 1; ④) {
    ①
    ②
    if (X)
        continue;
    ③
    }
```

当X为真时，重复执行①②④
当X为假时，重复执行①②③④

§.基础知识题 – 循环结构

2. break与continue

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    int i=0,sum=0;

    while(i<1000) {
        i++;
        break;
        sum=sum+i;
        }

    cout << "i=" << i << endl;
    cout << " sum=" << sum << endl;

    return 0;
}
//问题1：循环执行了多少次？1次
//问题2：sum=sum+i执行了多少次？0次
```

Microsoft Visual Studio 调试控制台

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

```
#include <iostream>
using namespace std;

int main()
{
    int i=0,sum=0;

    while(i<1000) {
        i++;
        continue;
        sum=sum+i;
        }

    cout << "i=" << i << endl;
    cout << " sum=" << sum << endl;

    return 0;
}
//问题1：循环执行了多少次？1000次
//问题2：sum=sum+i执行了多少次？0次
```

Microsoft Visual Studio 调试控制台

```
i=1000
 sum=0
```

§.基础知识题 – 循环结构

3. 观察程序运行结果

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
#include <iomanip>    //格式输出
#include <cmath>      //fabs
#include <windows.h>  //取系统时间
using namespace std;

int main()
{
    int s=1;
    double n=1, t=1, pi=0;

    LARGE_INTEGER tick, begin, end;
    QueryPerformanceFrequency(&tick);        //取计数器频率
    QueryPerformanceCounter(&begin);         //取初始硬件定时器计数

    while(fabs(t)>1e-6) {
       pi=pi+t;
       n=n+2;
       s=-s;
       t=s/n;
       }

    QueryPerformanceCounter(&end);//获得终止硬件定时器计数

    pi=pi*4;
    cout << "n="  << setprecision(10) << n << endl;
    cout<<"pi="<<setiosflags(ios::fixed)<<setprecision(9)<<pi<< endl;

    cout << "计数器频率：" << tick.QuadPart << "Hz" << endl;
    cout << "时钟计数  ：" << end.QuadPart - begin.QuadPart << endl;
    cout << setprecision(6) << (end.QuadPart - begin.QuadPart)/double(tick.QuadPart) << "秒" <<endl;

    return 0;
}
```

用下面的迭代公式求Pi的值

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$$

(1) n，t，pi为double型

精度为1e-6：n=1000001　　pi=3.141590654　时间=0.002594(秒)

1e-7：n=10000001　　pi=3.141592454　时间=0.020412(秒)

1e-8：n=100000001　　pi=3.141592634　时间=0.203840(秒)

1e-9：n=1000000001　pi=3.141592652　时间=1.934466(秒)

(2) n，t，pi为float型

精度为1e-6：n=1000001　　pi=3.141593933　时间=0.018857(秒)

1e-7：n=10000001　　pi=3.141596556　时间=0.183082(秒)

1e-8：无结果

问：1、7项中哪个没结果？为什么？

n，t，pi为float型，精度为1e-8时没结果。因为精度1e-8已经超出了float型数据的精度，fabs(t)>1e-8永真，循环为永真循环，故无结果。

2、float和double同精度下哪个时间快？

float和double同精度下double时间快。

§. 基础知识题 – 循环结构

3. 观察程序运行结果

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int n = 0, i, m, k;
    bool prime;

    for (m = 103; m <= 200; m += 2) { //偶数没必要判断
        prime = true;                 //对每个数，先认为是素数
        k = int(sqrt(m));

        for (i = 2; i <= k; i++)
            if (m % i == 0) {
                prime = false;
                break;
            }

        if (prime) {
            cout << setw(5) << m;
            n = n + 1;   //计数器，只为了加输出换行

            if (n % 10 == 0) //每10个数输出一行
                cout << endl;
        }
    } //end of for

    return 0;
}
```

(1) 目前输出结果：一共21个，每10个一行

Microsoft Visual Studio 调试控制台

```
  101   103   107   109   113   127   131   137   139   149
  151   157   163   167   173   179   181   191   193   197
  199
```

(2) 将m的初值从101改为103，应该是20个，共2行。
实际呢？为什么？

Microsoft Visual Studio 调试控制台

```
  103   107   109   113   127   131   137   139   149   151

  157   163   167   173   179   181   191   193   197   199
```

实际是20个，共4行。

因为：当m的值为151时，prime的值为true，n的值为10，n%10==0为真，换行；

当m的值为153时，prime的值为false，n的值仍为10，n%10==0为真，换行；

当m的值为155时，prime的值为false，n的值仍为10，n%10==0为真，换行；

当m的值为157时，prime的值为true，n的值为11，n%10==0为假，不换行。

所以最终显示20个数据，共4行。

(3) 将左侧程序改正确

§.基础知识题 – 函数基础

1.C和C++的不写函数返回类型时的差异

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

fun()
{
    /* 注意：输出必须改为自己学号-姓名 */
    printf("2250758-林继申\n");
    return 0;
}

int main()
{
    fun();
    return 0;
}
```

1、将左侧程序贴到.c后缀的源程序中

2、将左侧程序贴到.cpp后缀的源程序中

3、结论：

如果C程序不写函数的返回类型，则编译器不报错，函数可以被正常调用，C程序可以正常运行。

如果C++程序不写函数的返回类型，则编译器报错，函数不能被正常调用，C++程序不能正常运行。

§.基础知识题 – 函数基础

1.C和C++的不写函数返回类型时的差异

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    printf("%lf\n", sqrt(2));
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0.000000
```

```
warning C4013: “sqrt”未定义；假设外部返回 int
warning C4477: “printf”：格式字符串“%lf”需要类型“double”的参数，但可变参数 1 拥有了类型“int”
```

将左侧的两个程序贴到.c后缀的源程序中编译运行，分析为什么结果不同

sqrt()函数是math.h头文件中的函数，前者程序中未加math.h头文件，故sqrt()函数未定义，假设外部返回int型，并返回值为0，所以输出0.000000；后者程序中添加math.h头文件，故sqrt(2)返回值为2的算术平方根，所以输出1.414214。

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

int main()
{
    printf("%lf\n", sqrt(2));
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1.414214
```

§.基础知识题 – 函数基础

2.main函数的返回值差异

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    return 0;
}
```

1、VS下编译

```
error C4430: 缺少类型说明符 - 假定为 int。注意: C++ 不支持默认 int
```

2、在Dev下编译

```
C:\Users\lenovo\Desktop\Dev Demo.exe
2250758-林继申
```

§.基础知识题 – 函数基础

2.main函数的返回值差异

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    return;
}
```

1、VS下编译

2、在Dev下编译

§.基础知识题 – 函数基础

2.main函数的返回值差异

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

long main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    return 0L;
}
```

1、VS下编译

2、在Dev下编译

3、综合2.A/2.B/2.C三题的结论，main函数的返回类型应定义为 int 最合适。

§.基础知识题 – 函数基础

3.函数的单向传值

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void fun(int x)
{
    cout << "x1=" << x << endl;
    x=5;
    cout << "x2=" << x << endl;
}

int main( )
{
    int k=15;
    cout << "k1=" << k << endl;
    fun(k);
    cout << "k2=" << k << endl;

    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
```

1、运行结果

2、为什么x的改变不会影响到k？

因为参数的传递方式是“单向传值”，即将实参的值复制一份到形参中，可以理解为“形参 = 实参”的形式，所以执行后形参的变化不影响实参值，形参在使用时分配空间，函数运行结束后释放空间。

§.基础知识题 – 函数基础

4. 函数的返回值

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(short x)
{
    cout << "x=" << x << endl;
    return 0;
}

int main()
{
    long k=70000;
    fun(k);
    cout << "k=" << k << endl;

    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
```

1、运行结果（含warning）

warning C4244: “参数”: 从“long”转换到“short”，可能丢失数据

2、用第2章的知识分析并解释运行结果

参数的传递方式是“单向传值”，即将实参的值复制一份到形参中，可以理解为“形参 = 实参”的形式，即short x = k，70000的二进制补码为10001000101110000，赋值给short型时按照不同长度的整型数据相互赋值的规则进行赋值，高位丢弃得到二进制补码为1000101110000，转换为十进制数为4464，即x的值。

§. 基础知识题 – 函数基础

4. 函数的返回值

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

short fun3()
{
    long a = 70000;
    return a;
}

int main()
{
    long d;
    d = fun3();
    cout << d << endl;

    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
```

1、运行结果（含warning）

warning C4244: “return”: 从“long”转换到“short”，可能丢失数据

2、用第2章的知识分析并解释运行结果

调用函数fun3()时，long型变量a的值为70000，函数fun3()的返回值为short型的a，因此进行数据类型强制转换，70000的二进制补码为10001000101110000，赋值给short型时按照不同长度的整型数据相互赋值的规则进行赋值，高位丢弃得到二进制补码为1000101110000，转换为十进制数为4464，即fun3()的返回值，之后再赋值给long型变量d，进行整型提升操作后得到二进制补码为00000000000000000001000101110000，转换为十进制数为4464，即d的值。

§.基础知识题 – 函数基础

4.函数的返回值

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void fun()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
}

int main()
{
    fun();
}

//main函数，返回类型为int，无return
```

Microsoft Visual Studio 调试控制台

2250758-林继申

```
#include <iostream>
using namespace std;

int fun()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
}

int main()
{
    fun();
    return 0;
}

//非main函数，返回类型为int，无return
```

error C4716: “fun”: 必须返回一个值

§.基础知识题 – 函数基础

4.函数的返回值

D.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f()
{
    int x=10;
}
int main()
{
    int k=10;

    k = k + f();
    k,f();
    cout << (k, f()) << endl;
    cout << (k, f(), k+2) << endl;

    return 0;
}
```

1、编译结果

```
error C2186: “+”:“void”类型的操作数非法
error C2679: 二元“<<”: 没有找到接受“void”类型的右操作数的运算符(或没有可接受的转换)
```

2、解释报error的两行为什么错

void表示不需要返回类型，f()无返回类型，不能参与运算，所以显示error C2186: “+”:“void”类型的操作数非法；

逗号表达式(k, f())的值为f()，但f()无返回类型，不能参与运算，所以显示error C2679: 二元“<<”: 没有找到接受“void”类型的右操作数的运算符(或没有可接受的转换)。

3、结论：

① 返回类型为void的函数不能出现在除<u>逗号</u>表达式外的任何表达式中

② 若逗号表达式要参与其它运算/输出，则返回类型为void的函数不能做为第<u>n</u>个表达式出现
(假设逗号表达式有n个表达式组成，此处填1~n)

§.基础知识题 – 函数基础

4. 函数的返回值

E. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(int x)
{
    if (x>10) {
        if (x>20)
            return 1;
        }
    else
        return 0;
}
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    cout << fun(5)  << endl;
    cout << fun(15) << endl;
    cout << fun(25) << endl;
    return 0;
}
```

1、运行结果

warning C4715: “fun”: 不是所有的控件路径都返回值

2、解释warning的含义

由于函数中的分支语句条件覆盖不全面，return未覆盖全部分支/出口，指定的函数可能无法返回值。如x的值为15时，x>10为真，x>20为假，无return语句，故这时函数的返回值不可信。

3、后三行输出中哪行不可信？

后三行输出中第二行的输出不可信。

§.基础知识题 – 函数基础

4.函数的返回值

F.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(int x)
{
    if (x>1)
        return 1;
    else if (x<=1)
        return 0;
} //if+else if已覆盖int型的全部表示范围

int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    cout << fun(0)  << endl;
    cout << fun(2) << endl;
    return 0;
}
```

1、运行结果

warning C4715: “fun”: 不是所有的控件路径都返回值

2、解释warning的含义（编译器是如何理解fun的逻辑的）

编译器不会理解到if已覆盖int型的全部表示范围，编译器只会理解函数中的分支语句条件覆盖不全面，return未覆盖全部分支/出口，所以指定的函数可能无法返回值。

§.基础知识题 – 函数基础

5.函数的调用

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

long fun(void)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0L;
}
int main()
{
    long k;
    k = fun();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

2250758-林继申

```
#include <iostream>
using namespace std;

long fun(void)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0L;
}
int main()
{
    long k;
    k = long fun();
    return 0;
}
```

error C2062: 意外的类型“long”

结论：函数调用时，<u>不能</u>写返回类型

§.基础知识题 – 函数基础

5.函数的调用

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(void)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
int main()
{
    int k;
    k = fun();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
2250758-林继申
```

```
#include <iostream>
using namespace std;

int fun(void)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
int main()
{
    int k;
    k = fun(void);
    return 0;
}
```

```
error C2144: 语法错误: "void" 的前面应有 ")"
error C2144: 语法错误: "void" 的前面应有 ";"
error C2059: 语法错误: ")"
warning C4091: "": 没有声明变量时忽略 "void" 的左侧
```

结论：无参函数调用时，参数位置 不能 写void

§. 基础知识题 – 函数基础

5. 函数的调用

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(int x, int y)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
int main()
{
    int k, x=10, y=15;
    k = fun(x, y);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
2250758-林继申
```

```
#include <iostream>
using namespace std;

int fun(int x, int y)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    return 0;
}
int main()
{
    int k, x=10, y=15;
    k = fun(int x, int y);
    return 0;
}
```

```
(13,13): error C2144: 语法错误: "int" 的前面应有 ")"
(13,9): error C2660: "fun": 函数不接受 0 个参数
(4,5): message : 参见 "fun" 的声明
(13,9): message : 尝试匹配参数列表 "()" 时
(13,13): error C2144: 语法错误: "int" 的前面应有 ":"
(13,17): error C2086: "int x": 重定义
(12,12): message : 参见 "x" 的声明
(13,20): error C2062: 意外的类型 "int"
(13,25): error C2059: 语法错误: ")"
```

结论：有参函数调用时，实参 <u>不能</u> 写类型

§.基础知识题 – 函数基础

5.函数的调用

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int fun(int x, int y)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
    return 0;
}
int main()
{
    int x=10, y=15;
    int fun(int x, int y);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

为什么本程序不报错但也无输出？

int fun(int x, int y);是对fun()函数的调用说明，在调用说明后没有对fun()函数的调用，程序直接运行至语句return 0;，所以本程序不报错但也无输出。

§. 基础知识题 – 函数基础

5. 函数的调用

D. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10, y = 15;
    fun(x, y);
    return 0;
}
int fun(int x, int y);
void f()
{
    fun(10, 15);
}
int fun(int x, int y)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
    return 0;
}
```

error C3861: “fun”: 找不到标识符

```
#include <iostream>
using namespace std;
int main()
{
    int fun(int x, int y);
    int x = 10, y = 15;
    fun(x, y);
    return 0;
}
void f()
{
    fun(10, 15);
}
int fun(int x, int y)
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
    return 0;
}
```

error C3861: “fun”: 找不到标识符

结论：1、函数声明如果放在函数外，则对哪些函数有效？

针对后面所有函数都有效，函数声明后面的所有被声明函数都能被调用。

2、函数声明如果放在函数内，则对哪个函数有效？

只对本函数有效，被声明函数只能在本函数内被调用。

§. 基础知识题 – 变量类型

1. 自动变量及形参的分配与释放

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f1(int x)
{
    int y;
    cout << &x << ' ' << &y << endl;//打印地址
}
void f2(long p)
{
    float q;
    cout << &p << ' ' << &q << endl;//打印地址
}
int main()
{
    f1(10);
    f2(15L);
    return 0;
}
```

1、运行结果截图及结论

截图：

结论：

1.1 int型变量x 和 long型变量p 共用了从 010FFA50（每次运行该地址不确定）开始的4个字节空间。

1.2 int型变量y 和 float型变量q 共用了从 010FFA3C（每次运行该地址不确定）开始的4个字节空间。

2、把f2中float q改为short q，运行结果截图及结论

截图：

结论：

2.1 q和 y 共用了从 008FFA84（每次运行该地址不确定）开始的 2 个字节空间。

§. 基础知识题 – 变量类型

1. 自动变量及形参的分配与释放

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f1(int x)
{
    int y;
    cout << &x << ' '<< &y << endl;
}

int main()
{
    f1(10);
    cout << "..." << endl;

    f1(10);
    cout << "..." << endl;

    f1(10);
    cout << "..." << endl;
    return 0;
}
```

1、运行结果截图

2、结论：

2.1 本示例中，三次调用时分配的x占用 相同 空间，三次调用时分配的y占用 相同 空间。

2.2 总结形参x和自动变量y的分配和释放规则

形参在使用时分配空间，函数运行结束后释放空间；自动变量在函数进入后分配空间，函数运行结束后释放空间（重复进行）。

§.基础知识题 – 变量类型

1.自动变量及形参的分配与释放

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f1()
{
    int a = 15;
    cout << &a << ' '<< a << endl;
}

void f2()
{
    long a = 70000;
    cout << &a << ' '<< a << endl;
}

void f3()
{
    short a = 23;
    cout << &a << ' '<< a << endl;
}

int main()
{
    f1();
    f2();
    f3();
    return 0;
}
```

1、运行结果截图

2、结论：

2.1 f1/f2/f3中的三个a占用 相同 空间。

2.2 如果当前正在执行f2函数，则f1中的a 已释放 ，f3中的a 未分配 。

§. 基础知识题 – 变量类型

1. 自动变量及形参的分配与释放

D. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f3()
{
    short a = 23;
    cout << "f3   " << &a << ' '<< a << endl;
}
void f2()
{
    long a = 70000;
    cout << "f2-1 " << &a << ' '<< a << endl;
    f3();
    cout << "f2-2 " << endl;
}
void f1()
{
    int a = 15;
    cout << "f1-1 " << &a << ' '<< a << endl;
    f2();
    cout << "f1-2 " << endl;
}
int main()
{
    f1();
    return 0;
}
```

1、运行结果截图

2、结论：

2.1 f1/f2/f3中的三个a占用 不同 空间

2.2 如果当前正在执行f1函数的cout-1语句，则f2中的a 未分配 ，f3中的a 未分配 ；

2.3 如果当前正在执行f1函数的cout-2语句，则f2中的a 已释放 ，f3中的a 已释放 ；

2.4 如果当前正在执行f2函数的cout-1语句，则f1中的a 已分配 ，f3中的a 未分配 ；

2.5 如果当前正在执行f2函数的cout-2语句，则f1中的a 已分配 ，f3中的a 已释放 ；

2.6 如果当前正在执行f3函数的cout语句， 则f1中的a 已分配 ，f2中的a 已分配 。

2.7 上述2.2~2.6问题中如果某个a是已分配状态，则此时这个a在何处？
a在“现场栈”中。

§.基础知识题 – 变量类型

2.局部变量的作用范围

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
void fun()
{
    int i,a;
    a=15;
    for(i=0;i<10;i++) {
        int y;
        y=11;
        a=16;
        }
     y=12;
     a=17;
}
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    fun();
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

error C2065: “y”: 未声明的标识符

2、解释出现的error/warning的原因

循环语句内的变量，只在循环语句中有效，自动变量在进入循环后分配空间，退出循环后释放空间，所以“y=12”处会显示error C2065: “y”: 未声明的标识符，因为在进入循环后分配空间的int型变量y在退出循环后已经释放空间。

§.基础知识题 – 变量类型

2.局部变量的作用范围

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void fun()
{
    int i,a=15;
    {
        int y;
        y=11;
        a=16;
        {
            int w=10;
            y=12;
            a=13;
            w=14;
        }
        w=15;
    }
    y=12;
    a=17;
}
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    fun();
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

```
error C2065: “w”: 未声明的标识符
error C2065: “y”: 未声明的标识符
```

2、解释出现的error/warning的原因

复合语句内的变量，只在复合语句中有效，自动变量在进入复合语句后分配空间，退出复合语句后释放空间，所以“w=12”处会显示error C2065: “w”: 未声明的标识符，“y=12”处会显示error C2065: “y”: 未声明的标识符，因为在进入复合语句后分配空间的int型变量w和int型变量y在退出复合语句后已经释放空间。

§.基础知识题 – 变量类型

2.局部变量的作用范围

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void fun()
{
    a=14;
}

int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    int a;
    a=15;
    fun();
    a=16;
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

error C2065: “a”: 未声明的标识符

2、结论：在某个函数(main)中定义的自动变量，在它的调用函数(fun)中 不允许 访问。

§.基础知识题 – 变量类型

3.全局变量的作用范围

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int f1()
{
    a=15;
}
int a;
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;
    a=16;
    return 0;
}
int f2()
{
    a=17;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

error C2065: “a”: 未声明的标识符

2、解释出现的error/warning的原因

全局变量从定义点到源文件结束之间的所有函数均可使用。“a=15”处在全局变量定义点之前，不可用，所以“a=15”处显示error C2065: “a”: 未声明的标识符。

§. 基础知识题 – 变量类型

3. 全局变量的作用范围

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;
int a;
void f1()
{
    a=15;
    cout << "fa=" << a << ' '<< &a << endl;
}
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    a=10;
    cout << "ma1=" << a << ' '<< &a << endl;
    f1();
    cout << "ma2=" << a << ' '<< &a << endl;
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

2、由运行结果中的地址可以证明，f1和main中访问的变量a是 相同 的a。

§.基础知识题 – 变量类型

3.全局变量的作用范围

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int f1(int a)
{
    a=15;
    cout << "fa=" << a << ' ' << &a << endl;
    return a;
}
int main()
{
    /* 注意：输出必须改为自己学号-姓名 */
    cout << "2250758-林继申" << endl;

    int a =10;
    cout << "ma1=" << a << ' ' << &a << endl;
    a = f1(a);
    cout << "ma2=" << a << ' ' << &a << endl;
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

2、由运行结果中的地址可以证明，f1和main中访问的变量a是 不同 的a。

3、a不是全局变量，解释为什么ma1和ma2两句cout输出的a值不相同？a是如何被改变的？

因为main()和f1()访问的是不同的a，所以ma1和ma2两句cout输出的a值不相同。实参a是因为赋值语句而改变的：实参a在main()函数中定义并初始化，输出“ma1=10”，之后实参向形参单向传值，输出“fa=15”并用函数返回值改变实参的值，输出“ma2=15”。

§. 基础知识题 – 变量类型

4. 变量同名

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int a=10, b;
void f1()
{
    int a=5, b;
    cout << "a1=" << a << ' ' << &a << endl;
    cout << "b1=" << b << ' ' << &b << endl;
}
void f2()
{
    cout << "a2=" << a << ' ' << &a << endl;
    cout << "b2=" << b << ' ' << &b << endl;
}
int main()
{
    f1();
    f2();
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

2、由b可知，局部变量不初始化，初值为 -858993460(不确定值)；全局变量不初始化，初值为 0 。

3、由截图可知，全局变量a/b的起始地址差308个字节；局部变量a/b之间差12个字节；全局和局部之间差约2610KB，说明这是两个不同的存储区，全局变量在静态存储区，局部变量在动态存储区。

§.基础知识题 – 变量类型

4.变量同名

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

int a=10;
short a;
void f1()
{
    int x=5;
    double x=1.2;

    short p=1, p=2;
}
int main()
{
    f1();
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

```
(5,7): error C2371: “a”: 重定义；不同的基类型
(4,5): message : 参见“a”的声明
(9,12): error C2371: “x”: 重定义；不同的基类型
(8,9): message : 参见“x”的声明
(11,18): error C2374: “p”: 重定义；多次初始化
(11,11): message : 参见“p”的声明
```

2、结合4.A/4.B可以得知：<u>不同</u>级别的变量允许同名；<u>相同</u>级别的变量不允许同名；变量同名是的使用规则是<u>“低层屏蔽高层”的规则。“低层屏蔽高层”的规则适用于全局变量与局部变量同名、局部变量和复合语句内的局部变量同名、在多层次嵌套的情况下不同层次的变量同名的情况。</u>

§.基础知识题 – 变量类型

5.自动变量与静态局部变量

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <iostream>
using namespace std;

void f1()
{
    int a=1;
    a++;
    cout << "a=" << a << ' ' << &a << endl;

    static int b=1;
    b++;
    cout << "b=" << b << ' ' << &b << endl;
}
int main()
{
    f1();
    f1();
    f1();
    return 0;
}
```

注：如果是error，贴error截图
如果是warning，贴warning截图+运行结果
如果正常，贴运行结果

1、截图

2、结合a/b各自的地址和值，得到结论为：

自动变量a多次调用，则 每次 进行初始化，函数运行结束后 会 释放空间，下次进入时 再次分配 ；

静态局部变量a多次调用，则 仅第一次 进行初始化，函数运行结束后 不会 释放空间，下次进入时 继续使用上次的空间 ；

根据上面的分析结果，自动变量应该放在 动态数据存储区 ，静态局部变量应该放在 静态数据存储区 。

§.基础知识题 - cin与cout的基本使用

1.cout的基本理解

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    /* 第1组 */
    cout << "This is a C++ program." << endl;

    /* 第2组 */
    cout << "This is " << "a C++ " << "program." << endl;

    /* 第3组 */
    cout << "This is "
         << "a C++ "
         << "program."
         << endl;

    /* 第4组 */
    cout << "This is ";
    cout << "a C++ ";
    cout << "program.";
    cout << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
This is a C++ program.
This is a C++ program.
This is a C++ program.
This is a C++ program.
```

第3组和第4组在语句上的区别是：
第3组是1个cout语句分4行，前三行无分号；
第4组是4个cout语句，每行有分号。

§. 基础知识题 - cin与cout的基本使用

1. cout的基本理解

B. 观察下列4个程序的运行结果，回答问题并将各程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    int a=10, b=15, c=20;
    cout << a << b << c;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

101520

```
#include <iostream>
using namespace std;

int main()
{
    int a=10, b=15, c=20;
    cout << a, b, c;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

10

```
#include <iostream>
using namespace std;

int main()
{
    int a=10, b=15, c=20;
    cout << (a, b, c) << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

20

```
#include <iostream>
using namespace std;

int main()
{
    int a=10, b=15, c=20;
    cout << a, b, c << endl;
    return 0;
}
```

error C2563: 在形参表中不匹配
error C2568: “<<”: 无法解析函数重载
message : 可能是“std::basic_ostream<_Elem,_Traits> &std::endl(std::basic_ostream<_Elem,_Traits> &)”

解释这3个程序输出不同的原因：

程序1：<<运算符为左结合，优先级相同时先进行输出，所以依次输出a、b、c的值

程序2：<<运算符的优先级高于逗号运算符，先进行输出，所以输出a的值

程序3：()运算符的优先级高于<<运算符，先计算(a, b, c)，得到的值为c，所以输出c的值

解释错误原因：

一个流插入运算符<<只能输出1个数据。另外，逗号运算符的优先级低于流插入运算符<<，所以后面相当于c<<endl，无法解析函数重载。

结论：一个流插入运算符 << 只能输出 1 个数据。

§.基础知识题 - cin与cout的基本使用

1.cout的基本理解

C.观察下列2个程序的运行结果，回答问题并将各程序的运行结果截图贴上

```
#include <iostream>
using namespace std;
int main()
{
    char ch = 65;
    cout << ch << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

A

```
#include <iostream>
using namespace std;
int main()
{
    int ch = 65;
    cout << ch << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

65

解释这两个程序输出不同的原因：

程序1：ch为字符型变量，65代表字符型变量ch的ASCII码，输出结果为ASCII码为65的字符，即输出结果为A

程序2：ch为整型变量，65代表整型变量ch的值，即输出结果为65

§.基础知识题 - cin与cout的基本使用

1.cout的基本理解

D.程序同C，将修改后符合要求的程序及运行结果贴上

```
#include <iostream>
using namespace std;
int main()
{
    char ch = 65;
    cout << ch << endl;
    return 0;
}
```

在char类型不变的情况下，要求输出为65
（不允许添加其它变量）

```
#include <iostream>
using namespace std;
int main()
{
    int ch = 65;
    cout << ch << endl;
    return 0;
}
```

在int类型不变的情况下，要求输出为A
（不允许添加其它变量）

§.基础知识题 - cin与cout的基本使用

1.cout的基本理解

E.程序同C，将修改后符合要求的程序及运行结果贴上

```
#include <iostream>
using namespace std;
int main()
{
    char ch = 65;
    cout << ch << endl;
    return 0;
}
```

在char类型不变的情况下，要求输出为65
（不允许添加其它变量，
不允许使用任何方式的强制类型转换）

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

A.运行下面的程序，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;
int main()
{
    short k;
    cin >> k;
    cout << cin.good();
    cout << " k=" << k << endl;

    return 0;
}
```

基础知识：

short的最小值是：-32768

short的最大值是：32767

1、输入：123↙

2、输入：123 456↙(一个空格)

3、输入：123　　456↙(多个空格)

4、输入：123m↙

5、输入：m↙

6、输入：　　123↙(持续多个空格后，再输入123，按回车)

7、输入：　　↙ (持续多个空格后，按回车)
　　123↙　(再输入123，按回车)

8、输入：↙
　　...
　　↙
　　123↙ (持续多个空回车后，输入123)

分析结果：

1、在前面有正确输入的情况下，回车、空格、(对int型而言是非法的字符)m的作用是？
中止输入
2、直接输入若干空格和回车后，再输入正确，变量是否能得到正确的值？
是
3、直接输入(对int型而言是)非法的数据m，输出是？
0

Microsoft Visual Studio 调试控制台
```
123
1 k=123
```

Microsoft Visual Studio 调试控制台
```
123 456
1 k=123
```

Microsoft Visual Studio 调试控制台
```
123     456
1 k=123
```

Microsoft Visual Studio 调试控制台
```
123m
1 k=123
```

Microsoft Visual Studio 调试控制台
```
m
0 k=0
```

Microsoft Visual Studio 调试控制台
```
     123
1 k=123
```

Microsoft Visual Studio 调试控制台
```
123
1 k=123
```

Microsoft Visual Studio 调试控制台
```
123
1 k=123
```

§.基础知识题 – cin与cout的基本使用

2.cin的基本理解 – 单数据情况

B.运行下面的程序，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    short k;
    cin >> k;
    cout << "k=" << k << endl;
    cout << "cin.good()=" << cin.good() << endl;
    cout << "cin.fail()=" << cin.fail() << endl;
    return 0;
}
```

结论：

多个输入中，编号 4、5、6 输入的k值是不可信的

1、输入：123↙　　（正确+回车）

2、输入：123␣456↙（正确+空格）

3、输入：-123m↙　（正确+非法字符）

4、输入：m↙　　（直接非法字符）

5、输入：54321↙　（超上限）

6、输入：-40000↙　（超下限）

VS与Dev编译结果一致

Microsoft Visual Studio 调试控制台

```
123
k=123
cin.good()=1
cin.fail()=0
```

Microsoft Visual Studio 调试控制台

```
123 456
k=123
cin.good()=1
cin.fail()=0
```

Microsoft Visual Studio 调试控制台

```
-123m
k=-123
cin.good()=1
cin.fail()=0
```

Microsoft Visual Studio 调试控制台

```
m
k=0
cin.good()=0
cin.fail()=1
```

Microsoft Visual Studio 调试控制台

```
54321
k=32767
cin.good()=0
cin.fail()=1
```

Microsoft Visual Studio 调试控制台

```
-40000
k=-32768
cin.good()=0
cin.fail()=1
```

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

B-Compare.运行下面的对比程序（cin输入与赋值），观察运行结果并与B的输出结果进行对比分析

```
#include <iostream>
using namespace std;
int main()
{
    short k1, k2, k3, k4, k5;

    k1 = 12345;
    k2 = 54321;
    k3 = 70000;
    k4 = -12345;
    k5 = -54321;

    cout << k1 << endl;
    cout << k2 << endl;
    cout << k3 << endl;
    cout << k4 << endl;
    cout << k5 << endl;

    return 0;
}
```

B的输入：

1、输入：12345↙ （合理范围）
对应本例的k1=

2、输入：54321↙ （超上限但未超同类型的u_short上限）
对应本例的k2=

3、输入：70000↙ （超上限且超过同类型的u_short上限）
对应本例的k3=

4、输入：-12345↙（合理范围）
对应本例的k4=

5、输入：-54321↙（超下限）
对应本例的k5=

对比分析：输入的值在合理范围内时，输出与输入一致。输入的值超出short型数据的上下限时，输出与输入不一致，输出的值是short型数据的上下限，赋值时得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

C.仿B，自行构造不同测试数据，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    int k;
    cin >> k;
    cout << "k=" << k << endl;
    cout << "cin.good()=" << cin.good() << endl;
    cout << "cin.fail()=" << cin.fail() << endl;
    return 0;
}
```

结论：

多个输入中，编号 2、3、5 输入的k值是不可信的

1、输入：2000000000↙
（合理范围）

```
Microsoft Visual Studio 调试控制台
2000000000
k=2000000000
cin.good()=1
cin.fail()=0
```

2、输入：4000000000↙
（超上限但未超同类型的u_int上限）

```
Microsoft Visual Studio 调试控制台
4000000000
k=2147483647
cin.good()=0
cin.fail()=1
```

3、输入：9000000000↙
（超上限且超过同类型的u_int上限）

```
Microsoft Visual Studio 调试控制台
9000000000
k=2147483647
cin.good()=0
cin.fail()=1
```

4、输入：-2000000000↙
（合理范围）

```
Microsoft Visual Studio 调试控制台
-2000000000
k=-2000000000
cin.good()=1
cin.fail()=0
```

5、输入：-4000000000↙
（超下限）

```
Microsoft Visual Studio 调试控制台
-4000000000
k=-2147483648
cin.good()=0
cin.fail()=1
```

VS与Dev编译结果一致

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

C-Compare.仿B-Compare，构造对比程序（cin输入与赋值，int型），观察运行结果并与C的输出结果进行对比分析

需要回答下列问题：

1、输入/赋值超int上限但未超同类型的u_int上限，两者是否一致？如果有区别，区别是？

不一致。

输入超int上限但未超同类型的u_int上限，得到的结果是int型变量的上限2147483647；

赋值超int上限但未超同类型的u_int上限，按照同长度的signed与unsigned相互赋值的规制进行赋值，得到的结果的二进制补码是原数据的二进制形式。

2、输入/赋值超int上限且超同类型的u_int上限，两者是否一致？如果有区别，区别是？

不一致。

输入超int上限且超同类型的u_int上限，得到的结果是int型变量的上限2147483647；

赋值超int上限且超同类型的u_int上限，按照不同长度的整型数据相互赋值的规则进行赋值，得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

3、输入/赋值超int下限，两者是否一致？如果有区别，区别是？

不一致。

输入超int下限，得到的结果是int型变量的下限-2147483648；

赋值超int下限，得到的结果是原数据的二进制补码进行高位丢弃操作和低位赋值之后对应的十进制数。

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

D.运行下面的程序，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    unsigned short k;
    cin >> k;
    cout << "k=" << k;
    cout << " good=" << cin.good();
    cout << " fail=" << cin.fail() << endl;
    return 0;
}
```

结论：

多个输入中，编号 2、6 输入的k值是不可信的

VS与Dev编译结果一致

1、输入：12345↙
（合理范围）

2、输入：70000↙
（超上限）

3、输入：-12345↙
（负数但未超过short下限）

4、输入：-1↙
（负数且未超过short下限）

5、输入：-65535↙
（负数且未超过u_short上限加负号后的下限）

6、输入：-65536↙
（负数且超过u_short上限加负号后的下限）

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

D-Compare.仿B-Compare构造的对比程序（cin输入与赋值，u_short型），观察运行结果并与D的输出结果进行对比分析

```
#include <iostream>
using namespace std;
int main()
{
    unsigned short k1, k2, k3, k4, k5, k6;

    k1 = 12345;
    k2 = 70000;
    k3 = -12345;
    k4 = -1;
    k5 = -65535;
    k6 = -65536;

    cout << k1 << endl;
    cout << k2 << endl;
    cout << k3 << endl;
    cout << k4 << endl;
    cout << k5 << endl;
    cout << k6 << endl;
    return 0;
}
```

1、输入：12345↙ （合理范围）
对应本例的k1=

2、输入：70000↙ （超上限）
对应本例的k2=

3、输入：-12345↙（负数但未超过short下限）
对应本例的k3=

4、输入：-1↙ （负数且未超过short下限）
对应本例的k4=

5、输入：-65535↙（负数且未超过u_short上限加负号后的下限）
对应本例的k5=

6、输入：-65536↙（负数且超过u_short上限加负号后的下限）
对应本例的k6=

Microsoft Visual Studio 调试控制台

```
12345
4464
53191
65535
1
0
warning C4305: "=": 从"int"到"unsigned short"截断
warning C4309: "=": 截断常量值
warning C4309: "=": 截断常量值
```

D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe

```
12345
4464
53191
65535
1
0
[Warning] unsigned conversion from 'int' to 'short unsigned int' changes value from '70000' to '4464' [-Woverflow]
[Warning] unsigned conversion from 'int' to 'short unsigned int' changes value from '-65535' to '1' [-Woverflow]
[Warning] unsigned conversion from 'int' to 'short unsigned int' changes value from '-65536' to '0' [-Woverflow]
```

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

E.仿D，自行构造不同测试数据，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    unsigned int k;
    cin >> k;
    cout << "k=" << k;
    cout << " good()=" << cin.good();
    cout << " fail()=" << cin.fail() << endl;
    return 0;
}
```

1、输入：4000000000↙
（合理范围）

Microsoft Visual Studio 调试控制台

```
4000000000
k=4000000000 good()=1 fail()=0
```

2、输入：5000000000↙
（超上限）

Microsoft Visual Studio 调试控制台

```
5000000000
k=4294967295 good()=0 fail()=1
```

3、输入：-2000000000↙
（负数但未超int下限）

Microsoft Visual Studio 调试控制台

```
-2000000000
k=2294967296 good()=1 fail()=0
```

4、输入：-4000000000↙
（负数且未超过u_int
上限加负号后的下限）

Microsoft Visual Studio 调试控制台

```
-4000000000
k=294967296 good()=1 fail()=0
```

5、输入：-5000000000↙
（负数且超过u_int上限加负号后的下限）

Microsoft Visual Studio 调试控制台

```
-5000000000
k=4294967295 good()=0 fail()=1
```

结论：

多个输入中，编号 2、5 输入的k值是不可信的

VS与Dev编译结果一致

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

E-Compare.仿B-Compare，构造对比程序（cin输入与赋值，u_int型），观察运行结果并与E的输出结果进行对比分析

需要回答下列问题：

1、输入/赋值超u_int上限，两者是否一致？如果有区别，区别是？
不一致。
输入超u_int上限，得到的结果是u_int型变量的上限4294967295；
赋值超u_int上限，按照不同长度的整型数据相互赋值的规则进行赋值，得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

2、输入/赋值为负数但未超int下限，两者是否一致？如果有区别，区别是？
一致。

3、输入/赋值为负数且未超过u_int上限加负号后的下限，两者是否一致？如果有区别，区别是？
一致。

4、输入/赋值为负数且超过u_int上限加负号后的下限？如果有区别，区别是？
不一致。
输入超int下限，得到的结果是u_int型变量的上限4294967295；
赋值超int下限，得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

B-E.总结

1、signed数据在输入正确且范围合理的情况下
cin输入与赋值的输出一致。
2、signed数据在输入正确但超上限（未超同类型unsigned上限）的情况下
cin输入与赋值的输出不一致，cin输入得到的结果是该signed数据取值的上限，赋值得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。
3、signed数据在输入正确且超上限（超过同类型unsigned上限）的情况下
cin输入与赋值的输出不一致，cin输入得到的结果是该signed数据取值的上限，赋值得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。
4、signed数据在输入正确但超下限范围的情况下
cin输入与赋值的输出不一致，cin输入得到的结果是该signed数据取值的下限，赋值得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。
5、unsigned数据在输入正确且范围合理的情况下
cin输入与赋值的输出一致。
6、unsigned数据在输入正确且超上限的情况下
cin输入与赋值的输出不一致，cin输入得到的结果是该unsigned数据取值的上限，赋值得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。
7、unsigned数据在输入正确但为负数（未超同类型signed下限）的情况下
cin输入与赋值的输出一致。
8、unsigned数据在输入正确且为负数（超过同类型signed下限）的情况下
cin输入与赋值的输出一致。
9、unsigned数据在输入正确且为负数（超过同类型unsigned上限加负号后的下限）的情况下
cin输入与赋值的输出不一致，cin输入得到的结果是该unsigned数据取值的上限，赋值得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

对比：cin输入与变量赋值，在输入/右值超范围的情况下，表现是否相同？总结规律

表现不相同。cin输入在输入超范围的情况下，得到的结果是该数据类型取值的上限或下限；变量赋值在右值超范围的情况下，得到的结果是原数据的二进制补码进行高位丢弃和低位赋值操作之后对应的十进制数。

cin输入与变量赋值，在输入/右值合理范围的情况下，表现是否相同？总结规律

表现相同。

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

F.运行下面的程序，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    cin >> ch;

    cout << "ch=" << int(ch) << endl;
    cout << "ch=" << ch << endl;

    return 0;
}
```

1、键盘输入A（单个图形字符）

2、键盘输入\b（退格键的转义符）

3、键盘输入\101（A的ASCII码的8进制转义表示）

4、键盘输入\x41（A的ASCII码的16进制转义表示）

5、键盘输入65（A的ASCII码的十进制整数形式表示）

6、键盘输入CtrL+C（注意：是Ctrl+C组合键，注意不要有输入法栏）

7、键盘输入CtrL+z（注意：是Ctrl+z组合键，注意不要有输入法栏）

Microsoft Visual Studio 调试控制台

```
A
ch=65
ch=A
```

Microsoft Visual Studio 调试控制台

```
\b
ch=92
ch=\
```

Microsoft Visual Studio 调试控制台

```
\101
ch=92
ch=\
```

Microsoft Visual Studio 调试控制台

```
\x41
ch=92
ch=\
```

Microsoft Visual Studio 调试控制台

```
65
ch=54
ch=6
```

Microsoft Visual Studio 调试控制台

```
ch=-52
```

Microsoft Visual Studio 调试控制台

```
^Z
ch=-52
ch=
```

§.基础知识题 - cin与cout的基本使用

2.cin的基本理解 - 单数据情况

G.运行下面的程序，观察不同输入下的运行结果

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float f;
    cin >> f;

    cout << f << endl;
    cout << setprecision(20) << f << endl;

    return 0;
}


//注：setprecision(20)表示输出时保留小数点后20位
//    （已超float和double的有效位数）
```

1、键盘输入123.456 （合理范围正数，小数形式）

2、键盘输入1.23456e2 （合理范围正数，指数形式）

3、键盘输入-123.456（合理范围负数，小数形式）

4、键盘输入-1.23456e2 （合理范围负数，指数形式）

5、键盘输入123.456789（合理范围，但超有效位数）

6、键盘输入6.7e38（尾数超上限但数量级未超，仍是10^{38}）

7、键盘输入1.7e39（超上限且数量级已超10^{38}）

8、键盘输入-2.3e39（超上限且数量级已超10^{38}）

9、键盘输入1.23e-30（合理范围整数但指数很小）

10、键盘输入-1.23e-30（合理范围负数但指数很小）

Microsoft Visual Studio 调试控制台
```
123.456
123.456
123.45600128173828125
```

Microsoft Visual Studio 调试控制台
```
1.23456e2
123.456
123.45600128173828125
```

Microsoft Visual Studio 调试控制台
```
-123.456
-123.456
-123.45600128173828125
```

Microsoft Visual Studio 调试控制台
```
-1.23456e2
-123.456
-123.45600128173828125
```

Microsoft Visual Studio 调试控制台
```
123.456789
123.457
123.456787109375
```

Microsoft Visual Studio 调试控制台
```
6.7e38
0
0
```

Microsoft Visual Studio 调试控制台
```
1.7e39
0
0
```

Microsoft Visual Studio 调试控制台
```
-2.3e39
0
0
```

Microsoft Visual Studio 调试控制台
```
1.23e-30
1.23e-30
1.2299999549998595325e-30
```

Microsoft Visual Studio 调试控制台
```
-1.23e-30
-1.23e-30
-1.2299999549998595325e-30
```

§.基础知识题 - cin与cout的基本使用

3.cin的基本理解 - 多个同类型数据的情况

A.观察下列3个程序的运行结果，回答问题并将各程序的运行结果截图贴上

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4
1
2
3
4
```

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    cin >> a
        >> b
        >> c
        >> d;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4
1
2
3
4
```

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    cin >> a;
    cin >> b;
    cin >> c;
    cin >> d;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4
1
2
3
4
```

1、程序运行后，输入：1 2 3 4↙，观察输出结果

2、解释第2个和第3个程序的cin语句的使用区别：

第2个程序是1个cin语句分4行，前三行无分号；第3个程序是4个cin语句，每行有分号。

§.基础知识题 - cin与cout的基本使用

3.cin的基本理解 - 多个同类型数据的情况

B.程序同A，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

1、输入：1 2 3 4↙

2、输入：1　2　3　4↙(每个数字间多于一个空格)

3、输入：1↙
2↙
3↙
4↙ (每个数字后立即加回车)

4、输入：1↙
↙
↙
2↙
↙
3↙
↙
4↙ (每个数字后立即加回车 + 多个空回车)

结论：在输入正确的情况下，回车和空格的作用？
中止输入

Microsoft Visual Studio 调试控制台

```
1 2 3 4
1
2
3
4
```

Microsoft Visual Studio 调试控制台

```
1   2   3   4
1
2
3
4
```

Microsoft Visual Studio 调试控制台

```
1
2
3
4
1
2
3
4
```

Microsoft Visual Studio 调试控制台

```
1

2

3

4
1
2
3
4
```

§.基础知识题 - cin与cout的基本使用

3.cin的基本理解 - 多个同类型数据的情况

C.程序同A，观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    int a, b, c, d;
    cin >> a >> b >> c >> d;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    cout << d << endl;

    return 0;
}
```

1、输入：1 2 3 4m↙

2、输入：1 2 3m 4↙

3、输入：1 2m 3 4↙

4、输入：1m 2 3 4↙

5、输入：1 2 3 m↙

6、输入：1 2 m 4↙

7、输入：1 m 3 4↙

8、输入：m 2 3 4↙

Microsoft Visual Studio 调试控制台

```
1 2 3 4m
1
2
3
4
```

Microsoft Visual Studio 调试控制台

```
1 2 3 m
1
2
3
0
```

Microsoft Visual Studio 调试控制台

```
1 2 3m 4
1
2
3
0
```

Microsoft Visual Studio 调试控制台

```
1 2 m 4
1
2
0
-858993460
```

Microsoft Visual Studio 调试控制台

```
1 2m 3 4
1
2
0
-858993460
```

Microsoft Visual Studio 调试控制台

```
1 m 3 4
1
0
-858993460
-858993460
```

Microsoft Visual Studio 调试控制台

```
1m 2 3 4
1
0
-858993460
-858993460
```

Microsoft Visual Studio 调试控制台

```
m 2 3 4
0
-858993460
-858993460
-858993460
```

总结：多个cin输入时，错误输入出现在不同位置对输入正确性的影响

要求：综合观察运行结果，加上自己的思考，给出总结性的结论，这个结论要能对多个输入情况下不同位置的错误情况有普遍适应性，而不仅仅是简单的根据结论说错在1/2/3/4位置

当错误输入出现时，cin语句会中止输入，这时错误输入位置之前已输入的值是可信的，错误输入位置之后的值是不可信的。

§.基础知识题 - cin与cout的基本使用

3.cin的基本理解 - 多个同类型数据的情况

D.观察不同输入下的运行结果

```
#include <iostream>
using namespace std;

int main()
{
    char a, b, c;
    cin >> a >> b >> c;

    cout << "a=" << int(a) << endl;
    cout << "b=" << int(b) << endl;
    cout << "c=" << int(c) << endl;

    return 0;
}
```

1、输入：XYZ↙

2、输入：X YZ↙

3、输入：Ctrl+C↙

4、输入：XCtrl+C↙

5、输入：XYCtrl+C↙

6、输入：XYZCtrl+C↙

7、输入：Ctrl+z↙
（若未出结果则继续输入，可以按回车后多行输入，打印后观察结果）

8、输入：Ctrl+zXYZ↙
（若未出结果则继续输入，可以按回车后多行输入，打印后观察结果）

总结：多个cin输入时char型数据时

1、能否输入空格
能
2、Ctrl+C在输入中表示什么？（可自行查阅资料，若资料与表现不符，信哪个？）
Ctrl+C在输入中表示强制中断程序的执行，进程被完全关闭。（信表现）
3、Ctrl+z在输入中表示什么？（可自行查阅资料，若资料与表现不符，信哪个？）
Ctrl+z在输入中表示中断任务，挂起一个进程，进程未被关闭，仍然存在。（信表现）
4、Ctrl+z后不按回车而继续输入的其它字符，能否被读入？
不能

Microsoft Visual Studio 调试控制台
```
XYZ
a=88
b=89
c=90
```

Microsoft Visual Studio 调试控制台
```
X YZ
a=88
b=89
c=90
```

Microsoft Visual Studio 调试控制台
```
a=-5
```

Microsoft Visual Studio 调试控制台
```
Xa=-52
```

Microsoft Visual Studio 调试控制台
```
XYa=-52
```

Microsoft Visual Studio 调试控制台
```
XYZa=-52
```

Microsoft Visual Studio 调试控制台
```
^Z
a=-52
b=-52
c=-52
```

Microsoft Visual Studio 调试控制台
```
^ZXYZ
a=-52
b=-52
c=-52
```

§.基础知识题 - cin与cout的基本使用

3.cin的基本理解 - 多个同类型数据的情况

E.自行构造测试数据，观察不同输入下的运行结果

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float a, b, c;
    cin >> a >> b >> c;

    cout << "a=" << a << endl;
    cout << setprecision(20) << a << endl;

    cout << "b=" << b << endl;
    cout << setprecision(20) << b << endl;

    cout << "c=" << c << endl;
    cout << setprecision(20) << c << endl;

    return 0;
}
```

1、输入：1.7e39 123.456 1.23456e2↙ （第1个超上限，2/3正常）

2、输入：-4e38 123.456 1.23456e2↙ （第1个超下限，2/3正常）

3、输入：123.456 1.7e39 1.23456e2↙ （1/3正常，第2个超上限）

4、输入：123.456 -4e38 1.23456e2↙ （1/3正常，第2个超下限）

5、输入：123.456 1.23456e2 1.7e39↙ （1/2正常，第3个超上限）

6、输入：123.456 1.23456e2 -4e38↙ （1/2正常，第3个超下限）

总结：

1、多个cin输入时，错误输入出现在不同位置对输入正确性的影响

要求：综合观察运行结果，加上自己的思考，给出总结性的结论，这个结论要能对多个输入情况下不同位置的错误情况有普遍适应性，而不仅仅是简单的根据结论说错在1/2/3位置

错误输入位置之前已输入的值是可信的，错误输入位置之后的值是不可信的。

2、将float替换为double，上述结论是否仍然成立？

仍然成立

Microsoft Visual Studio 调试控制台

```
1.7e39 123.456 1.23456e2
a=0
0
b=-107374176
-107374176
c=-107374176
-107374176
```

Microsoft Visual Studio 调试控制台

```
-4e38 123.456 1.23456e2
a=0
0
b=-107374176
-107374176
c=-107374176
-107374176
```

Microsoft Visual Studio 调试控制台

```
123.456 1.7e39 1.23456e2
a=123.456
123.45600128173828125
b=0
0
c=-107374176
-107374176
```

Microsoft Visual Studio 调试控制台

```
123.456 -4e38 1.23456e2
a=123.456
123.45600128173828125
b=0
0
c=-107374176
-107374176
```

Microsoft Visual Studio 调试控制台

```
123.456 1.23456e2 1.7e39
a=123.456
123.45600128173828125
b=123.45600128173828125
123.45600128173828125
c=0
0
```

Microsoft Visual Studio 调试控制台

```
123.456 1.23456e2 -4e38
a=123.456
123.45600128173828125
b=123.45600128173828125
123.45600128173828125
c=0
0
```

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

A.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, c;
    cin >> a,b,c;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

1、如果编译有error或warning，则贴相应信息的截图

```
error C4700: 使用了未初始化的局部变量“b”
 error C4700: 使用了未初始化的局部变量“c”
```

2、如果能运行(包括有warning)，则输入三个正确的int型数据（例 :1 2 3↙），观察输出

```
D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe
1 2 3
1
4242432
7929704
```

3、分析为什么只有某个变量的结果是正确的

一个流提取运算符>>只能输入1个数据，因此只输入了a的值，未输入b和c的值，所以会出现error C4700: 使用了未初始化的局部变量“b”和error C4700: 使用了未初始化的局部变量“c”。

VS与Dev编译结果不一致

VS有error，不能运行；Dev无error，能运行

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

B.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    int a=66, b=67, c=68;
    cin >> a,b,c;

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

1、运行后，输入三个正确的int型数据(例 :1 2 3↙，注意不要是预置值)，观察输出

2、通过观察三个变量的输出，你得到了什么结论？

a,b,c均被初始化赋了初值，一个流提取运算符>>只能输入1个数据，因此只输入了a的值，未输入b和c的值，所以输出a的值被赋了新值，输出b和c的值仍为初值。

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

C.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    cin >> 5;
    cin >> a+10;

    cout << a << endl;
    return 0;
}
```

1、如果编译有error或warning，则贴相应信息的截图(信息太多则前五行)

```
error C2678: 二进制“>>”: 没有找到接受“std::istream”类型的左操作数的运算符(或没有可接受的转换)
message : 可能是“std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(std::basic_streambuf<char,std::char_traits<char>> *)”
message : 或       “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(void *&)”
message : 或       “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(long double &)”
message : 或       “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(double &)”
[Error] no match for 'operator>>' (operand types are 'std::istream' {aka 'std::basic_istream<char>'} and 'int')
In file included from C:/Program Files (x86)/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/9.2.0/include/c++/iostream
                 from D:\Learning\高级语言程序设计\My Project\My Project\My Project.cpp
[Note] candidate: 'std::basic_istream<_CharT, _Traits>::__istream_type& std::basic_istream<_CharT, _Traits>::operator>>(std::basic_istream<_CharT, _Traits>::__istream_type& (*)(std::basic_istream<_CharT, _Traits>::__istream_type&)) [with _CharT = char; _Trait
[Note] conversion of argument 1 would be ill-formed:
```

2、分析为什么编译有错

“cin >> 5;”：

cin语句无法给5赋值，因为5是常量，常量不能被赋值；

“cin >> a+10;”：

cin语句无法给a+10赋值，因为a+10是表达式，表达式不能被赋值。

3、结论：流提取运算符后面必须跟<u>变量</u>，不能是<u>常量或表达式</u>。

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

D.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    int a=66, b=67, c=68;
    cin >> (a,b,c);

    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
    return 0;
}
```

1、运行后，输入三个正确的int型数据(例 :1 2 3↙，注意不要是预置值)，观察输出

2、通过观察三个变量的输出，你得到了什么结论？

流提取运算符>>的优先级低于后续运算符时，会先进行后续的计算，之后再对计算出的结果进行输入。

3、和B进行比较，分析为什么结果有差异？

因为流提取运算符>>的优先级低于()运算符，故先计算(a,b,c)的值为c，这时为cin >> c，可以输入c的值，而B是输入a的值，所以结果有差异。

4、和C进行比较，与C得出的结论矛盾吗？

不矛盾，因为(a,b,c)的值为c，c是变量，满足“流提取运算符后面必须跟变量”的条件。

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

E.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    char c1, c2;
    int a;
    float b;
    cin >> c1 >> c2 >> a >> b;

    cout << c1 << ' '<< c2
<<' '<< a <<' '<< b << endl;
    return 0;
}
```

注：⌣表示空格

1、输入：1234⌣56.78↙

输出：

2、输入：1⌣2⌣34⌣56.78↙

输出：

3、分析在以上两种不同输入的情况下，为什么输出相同。

cin是按格式读入，到空格、回车、非法为止，c系统会自动根据in后变量的类型按最长原则来读取合理数据。

在输入1234⌣56.78时，按照最长原则，首先读取1赋值给char型数据c1，读取2赋值给char型数据c2，之后读取34（因为空格输入中止）赋值给int型数据a，最后读取56.78赋值给float型数据b；

在输入1⌣2⌣34⌣56.78时，读取到空格输入中止，1、2、34、56.78分别给赋值给char型数据c1、char型数据c2、int型数据a、float型数据b。

所以在以上两种不同输入的情况下输出相同，最后的输出为1⌣2⌣34⌣56.78。

§.基础知识题 - cin与cout的基本使用

4.cin的基本理解 - 其他情况

F.程序如下，观察编译及运行结果

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    cin >> a >> endl;

    return 0;
}
```

1、如果编译有error或warning，则贴相应信息的截图(信息太多则前五行)

```
error C2679: 二元“>>”: 没有找到接受“overloaded-function”类型的右操作数的运算符(或没有可接受的转换)
message : 可能是“std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(std::basic_streambuf<char,std::char_traits<char>> *)”
message : 或     “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(void *&)”
message : 或     “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(long double &)”
message : 或     “std::basic_istream<char,std::char_traits<char>> &std::basic_istream<char,std::char_traits<char>>::operator >>(double &)”
[Error] no match for 'operator>>' (operand types are 'std::basic_istream<char>::__istream_type' {aka 'std::basic_istream<char>'} and '<unresolved overloaded function type>')
In file included from C:/Program Files (x86)/Dev-Cpp/MinGW64/lib/gcc/x86_64-w64-mingw32/9.2.0/include/c++/iostream
                 from D:\Learning\高级语言程序设计\My Project\My Project\My Project.cpp
[Note] candidate: 'std::basic_istream<_CharT, _Traits>::__istream_type& std::basic_istream<_CharT, _Traits>::operator>>(std::basic_istream<_CharT, _Traits>::__istream_type& (*)(std::basic_istream<_CharT, _Traits>::__istream_type&)) [with _CharT = char; _Trait
[Note] no known conversion for argument 1 from '<unresolved overloaded function type>' to 'std::basic_istream<char>::__istream_type& (*)(std::basic_istream<char>::__istream_type&)' {aka 'std::basic_istream<char>& (*)(std::basic_istream<char>&)'}
```

2、结论：在cin中不能跟endl

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

A.进制前导符的使用：回答问题并将程序的运行结果截图贴上

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    short a1 = 1234, a2 = 0x1234, a3 = 01234, a4 = 0b1101001;  //常量为各进制表示正数
    cout << "dec:" << dec << a1 << ' ' << a2 << ' ' << a3 << ' ' << a4 << endl;
    cout << "hex:" << hex << a1 << ' ' << a2 << ' ' << a3 << ' ' << a4 << endl;
    cout << "oct:" << oct << a1 << ' ' << a2 << ' ' << a3 << ' ' << a4 << endl;
    cout << endl;

    short b1 = -1234, b2 = -0x1234, b3 = -01234, b4 = -0b1101001;  //常量为各进制表示负数
    cout << "dec:" << dec << b1 << ' ' << b2 << ' ' << b3 << ' ' << b4 << endl;
    cout << "hex:" << hex << b1 << ' ' << b2 << ' ' << b3 << ' ' << b4 << endl;
    cout << "oct:" << oct << b1 << ' ' << b2 << ' ' << b3 << ' ' << b4 << endl;
    cout << endl;

    short c1 = 40000, c2 = 0x9876, c3 = 0171234, c4 = 0b1101010100111100;  //赋值后最高位均为1，有warning
    cout << "dec:" << dec << c1 << ' ' << c2 << ' ' << c3 << ' ' << c4 << endl;
    cout << "hex:" << hex << c1 << ' ' << c2 << ' ' << c3 << ' ' << c4 << endl;
    cout << "oct:" << oct << c1 << ' ' << c2 << ' ' << c3 << ' ' << c4 << endl;
    cout << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
dec:1234 4660 668 105
hex:4d2 1234 29c 69
oct:2322 11064 1234 151

dec:-1234 -4660 -668 -105
hex:fb2e edcc fd64 ff97
oct:175456 166714 176544 177627

dec:-25536 -26506 -3428 -10948
hex:9c40 9876 f29c d53c
oct:116100 114166 171234 152474
```

```
warning C4309: “初始化”: 截断常量值
warning C4309: “初始化”: 截断常量值
warning C4309: “初始化”: 截断常量值
warning C4309: “初始化”: 截断常量值
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

A.总结及结论：

1、源程序中的整数，有 3 种不同进制的表示形式

2、无论源程序中整型常量表示为何种进制，它的机内存储均为 二进制补码 形式

3、如果想使数据输出时使用不同进制，要加 dec、hex、oct 等进制前导符

4、输出 无 二进制前导符

5、只有 10 进制有负数形式输出；
16进制输出负数时，特征是输出的16进制数是原负数的二进制补码直接转换的16进制数
8进制输出负数时，特征是输出的8进制数是原负数的二进制补码直接转换的8进制数

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

B.进制前导符的连续使用：回答问题并将程序的运行结果截图贴上

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 10;

    cout << a << ' ' << a+1 << ' ' << a+2 << endl;
    cout << hex;
    cout << a << ' ' << a+1 << ' ' << a+2 << endl;
    cout << oct;
    cout << a << ' ' << a+1 << ' ' << a+2 << endl;
    cout << dec;
    cout << a << ' ' << a+1 << ' ' << a+2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
10 11 12
a b c
12 13 14
10 11 12
```

结论：

dec/hex/oct等进制前导符设置后，对后面的<u>所有</u>数据有效，直到用另一个控制符去改变为止

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

C. setbase的使用：同1. A的形式，按要求自行构造测试程序，回答问题并将程序的运行结果截图贴上

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 10;

    cout << setbase(8);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << setbase(10);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << setbase(16);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << setbase(2);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
12 13 14
10 11 12
a b c
10 11 12
```

自行构造若干组测试数据，运行并截图

结论：

1、setbase中允许的合法值有<u>8、10、16</u>

2、当setbase中出现非法值时，处理方法是<u>以默认十进制形式输出数据</u>

3、setbase设置后，对后面的<u>所有</u>数据有效，直到用另一个setbase去改变为止

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

D. ios::uppercase的使用：按要求自行构造测试程序，能对比看出用和不用的差别即可

My Project.cpp

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int a = 10;

    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    cout << setiosflags(ios::uppercase);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    cout << resetiosflags(ios::uppercase);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    return 0;
}
```

测试程序中的数据类型为int，自行构造若干组测试数据，运行并截图

结论：

1、uppercase和 16 进制一起使用才能看出效果

2、uppercase设置后，对后面的 所有 数据有效

3、同一个程序中，设置完uppercase，如果想恢复小写，具体的做法是使用resetiosflags(ios::uppercase)；终止已设置的输出格式状态

Microsoft Visual Studio 调试控制台

```
10 11 12
a b c
12 13 14
10 11 12
10 11 12
A B C
12 13 14
10 11 12
10 11 12
a b c
12 13 14
10 11 12
```

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

E. ios::showpos的使用：按要求自行构造测试程序，能对比看出用和不用的差别即可

My Project.cpp

My Project (全局范围)

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int a = 10;

    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    cout << setiosflags(ios::showpos);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    cout << resetiosflags(ios::showpos);
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << hex;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << oct;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;
    cout << dec;
    cout << a << ' ' << a + 1 << ' ' << a + 2 << endl;

    return 0;
}
```

测试程序中的数据类型为int，自行构造若干组测试数据，运行并截图

结论：

1、showpos和 10 进制一起使用才能看出效果

2、showpos设置后，对后面的 所有 数据有效

3、同一个程序中，设置完showpos，如果想取消，具体的做法是使用resetiosflags(ios::showpos)；终止已设置的输出格式状态

Microsoft Visual Studio 调试控制台

```
10 11 12
a b c
12 13 14
10 11 12
+10 +11 +12
a b c
12 13 14
+10 +11 +12
10 11 12
a b c
12 13 14
10 11 12
```

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

F. setprecision的使用 – 单独使用 – (1)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float f1 = 1234.5678F;
    float f2 = 8765.4321F;

    /* 第1组：不设或非法 */
    cout << f1 << ' ' << f2 << endl;
    cout << setprecision(0) << f1 << ' ' << f2 << endl;

    /* 第2组：小于等于整数位数 */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(2) << f1 << ' ' << f2 << endl;
    cout << setprecision(3) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;

    /* 第3组：大于整数位数，但小与等于float型有效数字 */
    cout << endl;
    cout << setprecision(5) << f1 << ' ' << f2 << endl;
    cout << setprecision(6) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;

    /* 第4组：大于float型有效数字 */
    cout << endl;
    cout << setprecision(8) << f1 << ' ' << f2 << endl;
    cout << setprecision(9) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1234.57 8765.43
1e+03 9e+03

1e+03 9e+03
1.2e+03 8.8e+03
1.23e+03 8.77e+03
1235 8765

1234.6 8765.4
1234.57 8765.43
1234.568 8765.432

1234.5677 8765.4316
1234.56775 8765.43164
1234.567749 8765.431641
1234.5677490234375 8765.431640625
```

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

F. setprecision的使用 – 单独使用 – (2)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float f1 = 1234567890123456789.0F;
    float f2 = 9876543210987654321.0F;

    /* 第1组：不设或非法 */
    cout << f1 << ' ' << f2 << endl;
    cout << setprecision(0) << f1 << ' ' << f2 << endl;

    /* 第2组：小于等于整数位数 并且 小与等于float型有效数字 */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(2) << f1 << ' ' << f2 << endl;
    cout << setprecision(3) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(5) << f1 << ' ' << f2 << endl;
    cout << setprecision(6) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;

    /* 第3组：大于float型有效数字 */
    cout << endl;
    cout << setprecision(8) << f1 << ' ' << f2 << endl;
    cout << setprecision(9) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl; //为什么f1比f2少一位？
    cout << setprecision(11) << f1 << ' ' << f2 << endl;   最后一位是0被省略
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1.23457e+18 9.87654e+18
1e+18 1e+19

1e+18 1e+19
1.2e+18 9.9e+18
1.23e+18 9.88e+18
1.235e+18 9.877e+18
1.2346e+18 9.8765e+18
1.23457e+18 9.87654e+18
1.234568e+18 9.876544e+18

1.2345679e+18 9.8765435e+18
1.23456794e+18 9.87654352e+18
1.23456794e+18 9.876543516e+18
1.2345679396e+18 9.8765435164e+18
1234567939550609408 9876543516404875264
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

F.setprecision的使用 – 单独使用 – (3)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    float f1 = 0.12345678F;
    float f2 = 0.87654321F;

    /* 第1组：不设或非法 */
    cout << f1 << ' ' << f2 << endl;
    cout << setprecision(0) << f1 << ' ' << f2 << endl;

    /* 第2组：小于等于float型有效数字 */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(2) << f1 << ' ' << f2 << endl;
    cout << setprecision(3) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(5) << f1 << ' ' << f2 << endl;
    cout << setprecision(6) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;

    /* 第3组：大于float型有效数字 */
    cout << endl;
    cout << setprecision(8) << f1 << ' ' << f2 << endl;
    cout << setprecision(9) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0.123457 0.876543
0.1 0.9

0.1 0.9
0.12 0.88
0.123 0.877
0.1235 0.8765
0.12346 0.87654
0.123457 0.876543
0.1234568 0.8765432

0.12345678 0.87654322
0.123456784 0.876543224
0.1234567836 0.8765432239
0.1234567835927009582519531 0.8765432238578796386718 75
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

F.setprecision的使用 – 单独使用 – 总结

重要结论：setprecision指定输出位数后，系统会按指定位数输出，即使指定位数超过数据的有效位数
（即：输出数据的某位开始是不可信的，但依然会输出）

1、给出setprecision单独使用时的显示规律总结（如果数据不够，可以再自己构造测试数据）

(1)当不设置setprecision时，以默认形式输出数据，即保留6位有效数字，超出有效位数的部分四舍五入；

(2)当非法设置setprecision时，输出数据保留1位有效数字，超出有效位数的部分四舍五入；

(3)当设置setprecision小于等于float型有效数字时，输出数据保留设定位数的有效数字，超出有效位数的部分四舍五入；

(4)当设置setprecision大于float型有效数字时，将按照IEEE754存储的数据四舍五入至对应有效位数，当指定位数超过数据的有效位数时，输出数据从这一位开始是不可信的，但依然会输出。

2、将1.F-(1)~(3)中的数据类型换为double型（有效位数为15位），自行构造测试数据，验证总结出的float型数据的显示规律是否同样适用于double型

适用

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

G.setprecision的使用 – 和ios::fixed一起 – (1)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F;
    float f2 = 8765.4321F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1234.57 8765.43
1234.567749 8765.431641

1234.6 8765.4
1234.5677 8765.4316
1234.5677490 8765.4316406
1234.5677490234 8765.4316406250
1234.5677490234375000000000000 8765.4316406250000000000000000
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

G.setprecision的使用 – 和ios::fixed一起 – (2)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234567890123456789.0F;
    float f2 = 9876543210987654321.0F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1.23457e+18 9.87654e+18
1234567939550609408.000000 9876543516404875264.000000

1234567939550609408.0 9876543516404875264.0
1234567939550609408.0000 9876543516404875264.0000
1234567939550609408.0000000 9876543516404875264.0000000
1234567939550609408.0000000000 9876543516404875264.0000000000
1234567939550609408.0000000000000000000000000 9876543516404875264.0000000000000000000000000
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

G.setprecision的使用 – 和ios::fixed一起 – (3)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 0.12345678F;
    float f2 = 0.87654321F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0.123457 0.876543
0.123457 0.876543

0.1 0.9
0.1235 0.8765
0.1234568 0.8765432
0.1234567836 0.8765432239
0.1234567835927009582519531 0.8765432238578796386718750
```

§.基础知识题 - C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

G.setprecision的使用 - 和ios::fixed一起 - 总结

1、给出setprecision+ios::fixed使用时的显示规律总结（如果数据不够，可以再自己构造测试数据）

(1)不设置setprecision，只设置ios::fixed时，浮点数保留6位小数位数输出，超出保留小数位数的部分四舍五入；

(2)先设置ios::fixed，再设置setprecision(n)时，浮点数保留n位小数位数输出，将按照IEEE754存储的数据四舍五入至对应保留小数位数。

2、将1.G-(1)~(3)中的数据类型换为double型（有效位数为15位），自行构造测试数据，验证总结出的float型数据的显示规律是否同样适用于double型

适用

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

H.setprecision的使用 – 和ios::scientific一起 – (1)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F;
    float f2 = 8765.4321F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1234.57 8765.43
1.234568e+03 8.765432e+03

1.2e+03 8.8e+03
1.2346e+03 8.7654e+03
1.2345677e+03 8.7654316e+03
1.2345677490e+03 8.7654316406e+03
1.2345677490234375000000000e+03 8.7654316406250000000000000e+03
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

H.setprecision的使用 – 和ios::scientific一起 – (2)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234567890123456789.0F;
    float f2 = 9876543210987654321.0F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
1.23457e+18 9.87654e+18
1.234568e+18 9.876544e+18

1.2e+18 9.9e+18
1.2346e+18 9.8765e+18
1.2345679e+18 9.8765435e+18
1.2345679396e+18 9.8765435164e+18
1.2345679395506094080000000e+18 9.8765435164048752640000000e+18
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

H.setprecision的使用 – 和ios::scientific一起 – (3)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 0.12345678F;
    float f2 = 0.87654321F;

    /* 第1组：不设precision */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    /* 第2组：设置precision */
    cout << endl;
    cout << setprecision(1) << f1 << ' ' << f2 << endl;
    cout << setprecision(4) << f1 << ' ' << f2 << endl;
    cout << setprecision(7) << f1 << ' ' << f2 << endl;
    cout << setprecision(10) << f1 << ' ' << f2 << endl;
    cout << setprecision(25) << f1 << ' ' << f2 << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0.123457 0.876543
1.234568e-01 8.765432e-01

1.2e-01 8.8e-01
1.2346e-01 8.7654e-01
1.2345678e-01 8.7654322e-01
1.2345678359e-01 8.7654322386e-01
1.2345678359270095825195312e-01 8.7654322385787963867187500e-01
```

§.基础知识题 - C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

H.setprecision的使用 - 和ios::scientific一起 - 总结

1、给出setprecision+ios::scientific使用时的显示规律总结（如果数据不够，可以再自己构造测试数据）

(1)不设置setprecision，只设置ios::scientific时，浮点数保留6位小数位数，以科学计数法（即指数形式）输出，超出保留小数位数的部分四舍五入；

(2)先设置ios::scientific，再设置setprecision(n)时，浮点数保留n位小数位数，以科学计数法（即指数形式）输出，将按照IEEE754存储的数据四舍五入至对应保留小数位数。

2、将1.H-(1)~(3)中的数据类型换为double型（有效位数为15位），自行构造测试数据，验证总结出的float型数据的显示规律是否同样适用于double型

适用

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

I.ios::fixed和ios::scientific的混合使用 – 错误用法

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F, f2 = 8765.4321F;

    /* 第1组 */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    /* 第2组 */
    cout << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    return 0;
}
```

运行截图：

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F, f2 = 8765.4321F;

    /* 第1组 */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    /* 第2组 */
    cout << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    return 0;
}
```

运行截图：

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

I.ios::fixed和ios::scientific的混合使用 – 在上一页的基础上将程序改正确，并给出截图

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F, f2 = 8765.4321F;

    /* 第1组 */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    /* 第2组 */
    cout << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    return 0;
}
```

运行截图：

Microsoft Visual Studio 调试控制台

```
1234.57 8765.43
1234.567749 8765.431641

0x1.34a4560000000p+10 0x1.11eb740000000p+13
```

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    float f1 = 1234.5678F, f2 = 8765.4321F;

    /* 第1组 */
    cout << f1 << ' ' << f2 << endl;
    cout << setiosflags(ios::scientific) << f1 << ' ' << f2 << endl;

    /* 第2组 */
    cout << endl;
    cout << setiosflags(ios::fixed) << f1 << ' ' << f2 << endl;

    return 0;
}
```

运行截图：

Microsoft Visual Studio 调试控制台

```
1234.57 8765.43
1.234568e+03 8.765432e+03

0x1.34a4560000000p+10 0x1.11eb740000000p+13
```

结论：

如果想要在一个程序中同时显示fixed和scientific形式，需要在两者之间加入一句：

cout << setiosflags(ios::fixed); 或 cout << setiosflags(ios::scientific);

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

J. setw的基本使用 – (1)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 12345;

    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;

    cout << setw(3) << a << '#' << a + 1 << '*' << endl;
    cout << setw(6) << a << '#' << a + 1 << '*' << endl;
    cout << setw(10) << a << '#' << a + 1 << '*' << endl;
    cout << setw(15) << a << '#' << a + 1 << '*' << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
12345#12346*
 12345#12346*
     12345#12346*
          12345#12346*
```

结论：

1、setw指定的宽度是总宽度，当总宽度大于数据宽度时，显示规律为输出宽度被设置为n，缺省为右对齐，并用空格填充；当总宽度小于数据宽度时，显示规律为setw设置的宽度将被忽略，直接以默认形式输出数据。

2、setw的设置后，对后面的 仅一个 数据有效

3、程序最前面两行的输出，目的是什么？易于阅读，方便确定输出宽度，显示每个数字对应的位置。

4、每行输出的最后一个*，目的是什么？a+1已输出完毕，#与*之间输出的内容即为a+1输出的内容，用于显示末尾的位置，也是为了看清是否有隐含空格，便于确定输出宽度。

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

J.setw的基本使用 – (2)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    double a = 0.123456789012345;

    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;

    cout << setw(6) << a << '*' << endl;
    cout << setw(9) << a << '*' << endl;
    cout << setw(15) << a << '*' << endl;
    cout << setw(30) << a << '*' << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
0.123457*
 0.123457*
       0.123457*
                      0.123457*
```

结论：

1、setw指定的宽度是总宽度，对于实型数据，<u>包含</u>小数点

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

K.setw+setfill的使用

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 12345;

    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;

    cout << setfill('=') << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    cout << setw(15) << setfill('-') << a << '#' << a + 1 << '*' << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
=====12345#=====12346*
----------12345#12346*
```

结论：

1、setfill的作用是<u>setfill(c)可以设置填充字符，c可以是字符常量或字符变量</u>

2、setfill的设置后，对后面的<u>所有</u>数据有效

3、解释为什么第4行的第2个数(12346)前面没有-：<u>因为setw的设置后，对后面的仅一个数据有效，所以数据12346以默认形式输出，无空格填充，故无字符填充，12346前面没有-。</u>

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

L.setw/setfill与ios::left/ios::right的混合使用 – (1)

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    cout << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    cout << setiosflags(ios::left);
    cout << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
     12345#     12346*
12345     #12346     *
```

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    cout << setfill('=') << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    cout << setiosflags(ios::left);
    cout << setfill('=') << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;
}
```

结论：

1、ios::left的作用是<u>使输出数据左对齐</u>

2、如果不设置，缺省是<u>右对齐</u>

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
=====12345#=====12346*
12345=====#12346=====*
```

§.基础知识题 – C++方式输入输出的格式化控制

1.在cout中使用格式化控制符

L.setw/setfill与ios::left/ios::right的混合使用 – (2) – 同时使用(错误)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    /* 左对齐 */
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 右对齐 */
    cout << setiosflags(ios::right) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 左对齐 */
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
12345     #12346    *
     12345#    12346*
     12345#    12346*
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    /* 右对齐 */
    cout << setiosflags(ios::right) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 左对齐 */
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
     12345#    12346*
     12345#    12346*
```

§. 基础知识题 – C++方式输入输出的格式化控制

1. 在cout中使用格式化控制符

L. setw/setfill与ios::left/ios::right的混合使用 – 在上一页的基础上将程序改正确

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    /* 左对齐 */
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 右对齐 */
    cout << setiosflags(ios::right) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 左对齐 */
    cout << resetiosflags(ios::right);
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;   }
```

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
12345     #12346    *
     12345#    12346*
12345     #12346    *
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a = 12345;
    cout << "0         1         2         3" << endl;
    cout << "0123456789012345678901234567890123456789" << endl;
    /* 右对齐 */
    cout << setiosflags(ios::right) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    /* 左对齐 */
    cout << resetiosflags(ios::right);
    cout << setiosflags(ios::left) << setw(10) << a << '#' << setw(10) << a + 1 << '*' << endl;
    return 0;   }
```

结论：
如果想要right对齐后再left对齐，需要在两者之间加入一句：
cout << resetiosflags(ios::right);

Microsoft Visual Studio 调试控制台

```
0         1         2         3
0123456789012345678901234567890123456789
     12345#    12346*
12345     #12346    *
```

§.基础知识题 – C++方式输入输出的格式化控制

2.在cin中使用格式化控制符

A.基本要求：从键盘输入16进制数

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    short a;
    cin >> hex >> a;

    cout << "dec:" << dec << a << endl;
    cout << "hex:" << hex << a << endl;
    cout << "oct:" << oct << a << endl;

    return 0;
}
```

1、输入：1a2b↙
（合理正数）

2、输入：a1b2↙
（超上限但未超同类型的unsigned上限）

3、输入：fffff↙
（超上限且超过同类型的unsigned上限）

4、输入：-1a2b↙
（合理负数）

5、输入：-fffff↙
（超下限）

Microsoft Visual Studio 调试控制台

```
1a2b
dec:6699
hex:1a2b
oct:15053
```

Microsoft Visual Studio 调试控制台

```
a1b2
dec:32767
hex:7fff
oct:77777
```

Microsoft Visual Studio 调试控制台

```
fffff
dec:32767
hex:7fff
oct:77777
```

Microsoft Visual Studio 调试控制台

```
-1a2b
dec:-6699
hex:e5d5
oct:162725
```

Microsoft Visual Studio 调试控制台

```
-fffff
dec:-32768
hex:8000
oct:100000
```

§.基础知识题 – C++方式输入输出的格式化控制

2.在cin中使用格式化控制符

B.基本要求：从键盘输入8进制数（自行构造测试数据）

```
#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
    int a;
    cin >> setbase(8) >> a;

    cout << "dec:" << dec << a << endl;
    cout << "hex:" << hex << a << endl;
    cout << "oct:" << oct << a << endl;

    return 0;
}
```

1、输入：7346545000↙
（合理正数）

2、输入：26264057000↙
（超上限但未超同类型的unsigned上限）

3、输入：45201371000↙
（超上限且超过同类型的unsigned上限）

4、输入：-7346545000↙
（合理负数）

5、输入：-26264057000↙
（超下限）

Microsoft Visual Studio 调试控制台

```
7346545000
dec:1000000000
hex:3b9aca00
oct:7346545000
```

Microsoft Visual Studio 调试控制台

```
26264057000
dec:2147483647
hex:7fffffff
oct:17777777777
```

Microsoft Visual Studio 调试控制台

```
45201371000
dec:2147483647
hex:7fffffff
oct:17777777777
```

Microsoft Visual Studio 调试控制台

```
-7346545000
dec:-1000000000
hex:c4653600
oct:30431233000
```

Microsoft Visual Studio 调试控制台

```
-26264057000
dec:-2147483648
hex:80000000
oct:20000000000
```

§.基础知识题 – C++方式输入输出的格式化控制

2.在cin中使用格式化控制符

C.格式控制符setiosflags(ios::skipws)的使用

```
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cin >> a >> b;
    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

假设键盘输入为：12 34↙

则输出为：12
34

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a, b;
    cin >> setiosflags(ios::skipws);
    cin >> a >> b;
    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

假设键盘输入为：12 34↙

则输出为：12
34

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a, b;
    cin.unsetf(ios::skipws);
    cin >> a >> b;
    cout << a << endl;
    cout << b << endl;
    return 0;
}
```

假设键盘输入为：12 34↙

则输出为：12
0

综合以上三个例子可以得到如下结论：

1、"忽略前导空格"的意思，是空格不作为 终止输入条件 ，而是做为 字符 （因此导致第3个例子b未取得34）

2、setiosflags(ios::skipws)在缺省情况下是 有效 的，即不设置也生效

3、如果想取消"忽略前导空格"的设置，应使用cout << resetiosflags(ios::skipws);

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

A.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    int a=10, b=5;
    printf("a=%d, b=%d\n",a,b);

    printf("Hello, Welcome!\n");
    printf("Hello, Welcome\x21\n");
    return 0;
}
```

运行结果：

\x21是哪个ASCII字符的16进制转义表示？

!

转义符在格式控制表列中的输出形式是：

字符

```
//写出与左侧程序输出完全一致的，用C++方式的cout实现的代码

#include <iostream>
using namespace std;

int main()
{
    int a = 10, b = 5;
    cout << "a=" << a << ", b=" << b << endl;
    cout << "Hello, Welcome!" << endl;
    cout << "Hello, Welcome\x21" << endl;

    return 0;
}
```

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

B.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    int a=10, b=5;
    printf("a=%d\n", a, b);

    printf("Hello, Welcome!\n");
    return 0;
}
```

运行结果：

Microsoft Visual Studio 调试控制台

```
a=10
Hello, Welcome!
```

```
warning C4474: printf: 格式字符串中传递的参数太多
message : 占位符和其参数预计 1 可变参数，但提供的却是 2 参数
```

结论：如果%d(格式符的数量)小于后面输出表列的数量，则只输出格式符数量个数的变量，超过格式符数量的变量将不会被输出。

```
#include <stdio.h>

int main()
{
    int a=10, b=5;
    printf("a=%d %d %d\n", a, b);

    printf("Hello, Welcome!\n");
    return 0;
}
```

运行结果：

Microsoft Visual Studio 调试控制台

```
a=10 5 13111331
Hello, Welcome!
```

```
warning C4473: “printf”: 没有为格式字符串传递足够的参数
message : 占位符和其参数预计 3 可变参数，但提供的却是 2 参数
message : 缺失的可变参数 3 为格式字符串“%d”所需
```

结论：如果%d(格式符的数量)大于后面输出表列的数量，则前输出表列数量个数的变量输出是正确的，超过输出表列数量的变量输出是不可信的。

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

C.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    int a=10, b=5;
    int ret1, ret2, ret3, ret4, ret5;

    ret1 = printf("a=%d, b=%d\n", a, b);
    ret2 = printf("a=%d b=%d\n", a, b); //跟上面比，少一个逗号

    ret3 = printf("a=%d\n", a*1000);

    ret4 = printf("Hello\n");
    ret5 = printf("Hello"); //跟上面比，少一个\n
    printf("\n");

    printf("%d %d %d %d %d\n", ret1, ret2, ret3, ret4, ret5);

    return 0;
}
```

运行结果：

printf的返回值的含义是：

printf的返回值的数据类型为int型，其含义是打印的所有字符的总数（包括转义字符，比如回车\n）

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

D.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    short a = -2;
    printf("a=%hi %hd %hu %ho %hx %hX\n", a, a, a, a, a, a);
    printf("a=%i %d %u %o %x %X\n", a, a, a, a, a, a);
    printf("a=%li %ld %lu %lo %lx %lX\n", a, a, a, a, a, a);

    unsigned short b = 40000;
    printf("b=%hi %hd %hu %ho %hx %hX\n", b, b, b, b, b, b);
    printf("b=%i %d %u %o %x %X\n", b, b, b, b, b, b);
    printf("b=%li %ld %lu %lo %lx %lX\n", b, b, b, b, b, b);

    int c = 70000;
    printf("c=%hi %hd %hu %ho %hx %hX\n", c, c, c, c, c, c);
    printf("c=%i %d %u %o %x %X\n", c, c, c, c, c, c);
    printf("c=%li %ld %lu %lo %lx %lX\n", c, c, c, c, c, c);

    return 0;
}
```

运行结果：

Microsoft Visual Studio 调试控制台

```
a=-2 -2 65534 177776 fffe FFFE
a=-2 -2 4294967294 37777777776 fffffffe FFFFFFFE
a=-2 -2 4294967294 37777777776 fffffffe FFFFFFFE
b=-25536 -25536 40000 116100 9c40 9C40
b=40000 40000 40000 116100 9c40 9C40
b=40000 40000 40000 116100 9c40 9C40
c=4464 4464 4464 10560 1170 1170
c=70000 70000 70000 210560 11170 11170
c=70000 70000 70000 210560 11170 11170
```

参考printf的格式控制符和附加格式控制符，给出解释：

附加控制符l的作用：
表示长整型整数

附加控制符h的作用：
表示短整型整数

★ 在C方式中，如果要输出的数据类型与格式控制符的类型不一致，则以 格式控制符 为准

§. 基础知识题 – C方式输入输出的格式化控制

1. 格式化输出函数printf的基本理解

E. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    int a = 70000;
    printf("a=%ld*\n", a);
    printf("a=%10ld*\n", a);
    printf("a=%-10ld*\n\n", a);

    printf("a=%d*\n", a);
    printf("a=%10d*\n", a);
    printf("a=%10d*\n", -a);
    printf("a=%-10d*\n\n", a);
    printf("a=%-10d*\n", -a);

    printf("a=%hd*\n", a);
    printf("a=%10hd*\n", a);
    printf("a=%-10hd*\n\n", a);

    return 0;
}//注：最后加*的目的，是为了看清是否有隐含空格
```

运行结果：

参考printf的格式控制符和附加格式控制符，给出解释：

%ld　　：以长整型类型的数据类型输出
%10ld ：以长整型类型输出，总宽度10，右对齐
%-10ld：以长整型类型输出，总宽度10，左对齐
%d　　：以整型类型的数据类型输出
%10d　：以整型类型输出，总宽度10，右对齐
%-10d ：以整型类型输出，总宽度10，左对齐
%hd　　：以短整型类型的数据类型输出
%10hd ：以短整型类型输出，总宽度10，右对齐
%-10hd：以短整型类型输出，总宽度10，左对齐

如果输出负数且指定宽度，负号 占 总宽度

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

F.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    float f = 123.456f;
    printf("f=%f\n", f);
    printf("f=%e\n", f);
    printf("f=%E\n", f);
    printf("f=%g\n", f);
    printf("f=%G\n\n", f);

    f = 0.123456789f;
    printf("f=%f\n", f);
    printf("f=%e\n", f);
    printf("f=%E\n", f);
    printf("f=%g\n", f);
    printf("f=%G\n\n", f);

    f = 123456789.0f;
    printf("f=%f\n", f);
    printf("f=%e\n", f);
    printf("f=%E\n", f);
    printf("f=%g\n", f);
    printf("f=%G\n\n", f);

    return 0;
}
```

运行结果：

Microsoft Visual Studio 调试控制台

```
f=123.456001
f=1.234560e+02
f=1.234560E+02
f=123.456
f=123.456

f=0.123457
f=1.234568e-01
f=1.234568E-01
f=0.123457
f=0.123457

f=123456792.000000
f=1.234568e+08
f=1.234568E+08
f=1.23457e+08
f=1.23457E+08
```

参考printf的格式控制符和附加格式控制符，给出解释：

%f：将浮点数以十进制的<u>小数</u>形式输出

%e：将浮点数以十进制的<u>指数</u>形式输出

%E：将浮点数以十进制的<u>指数</u>形式输出

%e和%E的区别是：<u>以十进制的指数形式输出时，%e输出小写e，%E输出大写E</u>

%g/%G：输出形式为<u>从f,e中选择宽度较短的形式输出浮点数</u>

★ 仔细观察并叙述清楚，如果觉得左例还不足以理解，可以自己再构造测试数据

%g/%G：<u>输出形式的差别为若以小数形式输出则无差别，若以指数形式输出%g输出小写e，%G输出大写E</u>

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

G.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>
int main()
{
    double f = 123.456;
    printf("f=%f\n", f);
    printf("f=%lf\n", f);
    printf("f=%e\n", f);
    printf("f=%le\n", f);
    printf("f=%g\n", f);
    printf("f=%lg\n\n", f);

    f = 0.123456789;
    printf("f=%f\n", f);
    printf("f=%lf\n", f);
    printf("f=%e\n", f);
    printf("f=%le\n", f);
    printf("f=%g\n", f);
    printf("f=%lg\n\n", f);

    f = 123456789.0;
    printf("f=%f\n", f);
    printf("f=%lf\n", f);
    printf("f=%e\n", f);
    printf("f=%le\n", f);
    printf("f=%g\n", f);
    printf("f=%lg\n\n", f);
    return 0;
}
```

运行结果：

参考printf的格式控制符和附加格式控制符，给出解释：

(1)附加控制符f表示以小数形式输出

(2)附加控制符e表示以指数形式输出

(3)附加控制符g表示从f和e中选择宽度较短的形式输出

对于double数据：

1、格式符%f和%lf是否有区别？
无区别

2、如何证明你给出的1的结论？
三组数据均可证明该结论

Microsoft Visual Studio 调试控制台

```
f=123.456000
f=123.456000
f=1.234560e+02
f=1.234560e+02
f=123.456
f=123.456

f=0.123457
f=0.123457
f=1.234568e-01
f=1.234568e-01
f=0.123457
f=0.123457

f=123456789.000000
f=123456789.000000
f=1.234568e+08
f=1.234568e+08
f=1.23457e+08
f=1.23457e+08
```

§. 基础知识题 – C方式输入输出的格式化控制

1. 格式化输出函数printf的基本理解

H. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    double f = 123456.789;

    printf("f=%f*\n", f);
    printf("f=%.2f*\n", f);
    printf("f=%10.2f*\n", f);
    printf("f=%-10.2f*\n\n", f);

    printf("f=%e*\n", f);
    printf("f=%.2e*\n", f);
    printf("f=%10.2e*\n", f);
    printf("f=%-10.2e*\n\n", f);

    printf("f=%g*\n", f);
    printf("f=%.2g*\n", f);
    printf("f=%.3g*\n", f);
    printf("f=%10.2g*\n", -f);
    printf("f=%10.3g*\n", f);
    printf("f=%-10.2g*\n", -f);
    printf("f=%-10.3g*\n", f);

    return 0;
}
//注：最后加*的目的，是为了看清是否有隐含空格
```

运行结果：

参考printf的格式控制符和
附加格式控制符，给出解释：

%10.2f ：以小数类型输出，总宽度10，小数点后2位，右对齐
%-10.2f：以小数类型输出，总宽度10，小数点后2位，左对齐
%10.2e ：以指数类型输出，总宽度10，小数点后2位，右对齐
%-10.2e：以指数类型输出，总宽度10，小数点后2位，左对齐

对%f和%e而言，指定的总宽度包含小数点

对%g而言，%m.n中n代表的位数是指保留小数的位数

如果输出负数且指定宽度，负号占总宽度

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

I.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

int main()
{
    float f = 123456789.123;

    printf("f=%f*\n", f);
    printf("f=%10.2f*\n", f);
    printf("f=%-10.2f*\n", f);
    printf("f=%.2f*\n\n", f);

    double d = 12345678901234567.6789;

    printf("d=%f*\n", d);
    printf("d=%10.2f*\n", d);
    printf("d=%-10.2f*\n", d);
    printf("d=%.2f*\n\n", d);

    return 0;
}
//注：最后加*的目的，是为了看清是否有隐含空格
```

运行结果：

给出下面两个概念的结论：

1、在数据的有效位数超过精度时：
按照精度输出对应位数的数据，输出的数据在精度之前的位数可信，在精度之后的位数不可信

2、如果指定的总宽度小于有效位数的宽度，则：
按照有效位数的宽度输出

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

J.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

#define str "abcdefghijklmnopqrstuvwxyz"

int main()
{
    printf("str=%s*\n", str);
    printf("str=%30s*\n", str);
    printf("str=%-30s*\n", str);
    printf("str=%5s*\n", str);
    printf("str=%-5s*\n", str);
    printf("str=%.5s*\n", str);
    printf("str=%-.5s*\n", str);
    printf("str=%10.5s*\n", str);
    printf("str=%-10.5s*\n", str);

    return 0;
}
//注：最后加*的目的，是为了看清是否有隐含空格
```

运行结果：

参考printf的格式控制符和附加格式控制符，给出解释：

%s　：输出字符串类型的数据

%30s ：输出字符串类型的数据，总宽度30，右对齐

%-30s：输出字符串类型的数据，总宽度30，左对齐

如果指定的总宽度小于字符串的长度，则：按照实际字符串的长度输出字符串类型的数据，不受宽度限制

对%s而言，%m.n中n代表的位数是指前n个字符

§.基础知识题 – C方式输入输出的格式化控制

1.格式化输出函数printf的基本理解

K.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#include <stdio.h>

#define str "Student"
int main()
{
    int a = 65;
    printf("a=%o\n", a);
    printf("a=%x\n", a);
    printf("ch=%c\n", a);
    printf("s=%s\n\n", str);

    printf("a=0%o\n", a);
    printf("a=0x%x\n", a);
    printf("ch=\'%c\'\n", a);
    printf("s=\"%s\"\n\n", str);

    double d = 0.783;
    printf("百分比=%.2f%%\n", d * 100);

    return 0;
}
```

运行结果：

1、对比第1组和第2组输出，得出的结论是：

格式控制符/附加格式控制符，只负责给出<u>最原始数据</u>的输出，若需要前导字符、单双引号等，需要<u>自行加入</u>

2、输出字符'%'的方法是：<u>在格式控制表列中在对应位置添加%%</u>

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

A. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a;
    scanf("%d", a);
    printf("a=%d\n", a);
    return 0;
}
```

在VS中编译：

```
warning C4477: "scanf": 格式字符串"%d"需要类型"int *"的参数，但可变参数 1 拥有了类型"int"
error C4700: 使用了未初始化的局部变量"a"
```

在Dev中编译：

假设键盘输入为：10↙

则输出为：a=11739468

```
D:\Learning\高级语言程序设计\C-demo\C-demo\C-demo.exe
10
a=11739468
```

（经多次测试，发现每次输出a的值是一个随机数）

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

int main()
{
    int a = 0;
    scanf("%d", a);
    printf("a=%d\n", a);
    return 0;
}
```

在VS中编译：

```
warning C4477: "scanf": 格式字符串"%d"需要类型"int *"的参数，但可变参数 1 拥有了类型"int"
```

假设键盘输入为：10↙

则输出为：无

```
Microsoft Visual Studio 调试控制台
10
```

在Dev中编译：

假设键盘输入为：10↙

则输出为：无

```
D:\Learning\高级语言程序设计\C-demo\C-demo\C-demo.exe
10
```

结论：用scanf输入时，如果地址表列中直接跟变量名，则错误，其中VS的表现是显示warning C4477，或者弹出错误窗口，程序终止运行，Dev的表现是得到一个不可信的随机值，或者程序终止运行。

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

B. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a, b;
    scanf("%d %d", &a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

假设键盘输入为：10↙
15↙

则输出为：

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

int main()
{
    int a, b;
    scanf("%d%d", &a, &b);   //%d间无空格
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

假设键盘输入为：10↙
15↙

则输出为：

结论：

多个输入时，格式控制符间是否有空格不影响正确性。

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

C. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    int a=0, b=0;
    scanf("%d", &a, &b); //地址表列多
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

```
warning C4474: scanf: 格式字符串中传递的参数太多
message : 占位符和其参数预计 1 可变参数，但提供的却是 2 参数
```

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    int a;
    scanf("%d %d", &a);  //格式符多
    printf("a=%d\n", a);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

```
Microsoft Visual Studio 调试控制台
10 15
a=10, b=0
```

```
Microsoft Visual Studio 调试控制台
10 15
```

假设键盘输入为：10↙

则输出为：

```
Microsoft Visual Studio 调试控制台
10
a=10, b=0
```

```
Microsoft Visual Studio 调试控制台
10
15
```

结论：当地址表列的个数多于格式控制符时输入的值只能赋值给格式控制符数量的变量，不能赋值给其余变量

VS：

```
warning C4473: “scanf”：没有为格式字符串传递足够的参数
message : 占位符和其参数预计 2 可变参数，但提供的却是 1 参数
message : 缺失的可变参数 2 为格式字符串“%d”所需
```

假设键盘输入为：10 15↙

则输出为：无输出，程序终止运行

假设键盘输入为：10↙

15↙

则输出为：无输出，程序终止运行

结论：

当格式控制符的个数多于地址表列时VS会显示warning C4473，Dev会将输入的值赋值给地址表列数量的变量，其余的格式控制符会被忽略

Dev：

假设键盘输入为：10 15↙

则输出为：

```
D:\Learning\高级语言程序设计\C-demo\C-demo\C-demo.exe
10 15
a=10
```

假设键盘输入为：10↙

15↙

则输出为：

```
D:\Learning\高级语言程序设计\C-demo\C-demo\C-demo.exe
10
15
a=10
```

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

D.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a, ret;
    ret = scanf("%d", &a);
    printf("a=%d, ret=%d\n", a, ret);
    return 0;
}
```

假设键盘输入为：10↙

则输出为：

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

int main()
{
    int a, b, ret;
    ret = scanf("%d %d", &a, &b);
    printf("a=%d, b=%d ret=%d\n", a, b, ret);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

结论：在输入正确时，scanf的返回值是正确按指定格式输入变量的个数，即能正确接收到值的变量个数

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

E. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a, b;
    scanf("%d,%d", &a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

假设键盘输入为：10,15↙

则输出为：

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

int main()
{
    int a, b;
    scanf("a=%d,b=%d", &a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

假设键盘输入为：10 15↙

则输出为：

假设键盘输入为：10,15↙

则输出为：

假设键盘输入为：a=10,b=15↙

则输出为：

结论：当格式控制符中有其它字符（逗号，a=等）时，对这些字符的输入方法是在格式控制符的位置输入内容，其他对应位置依旧输入原来的字符

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

F. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    short c;

    scanf("%d", &c);
    printf("c=%hd\n", c);

    return 0;
}
```

假设键盘输入为：10↙

则输出为：（弹出错误窗口）

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

int main()
{
0;
} int c;

    scanf("%hd", &c);
    printf("c=%d\n", c);

    return
```

假设键盘输入为：10↙

则输出为：

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

int main()
{
    short c;

    scanf("%hd", &c);
    printf("c=%hd\n", c);

    return 0;
}
```

假设键盘输入为：10↙

则输出为：

假设键盘输入为：70000↙

则输出为：

结论：

1、附加格式控制符h的作用是输入短整型数

2、如果格式控制符的数据类型和要读取的变量类型的字节大小不一致（例：4/2字节），则显示warning C4477，输出一个不可信的值，或弹出错误窗口。

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

G.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a, b, c;

    scanf("%d %x %o", &a, &b, &c);
    printf("a=%d, b=%d, c=%d\n", a, b, c);

    return 0;
}
```

假设键盘输入为：10 11 12↙

则输出为：

假设键盘输入为：12 ab 76↙

则输出为：

假设键盘输入为：10 -11 +12↙

则输出为：

假设键盘输入为：12 -ab +76↙

则输出为：

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

H.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    short a, b, c;

    scanf("%hd %hx %ho", &a, &b, &c);
    printf("a=%hd, b=%hd, c=%hd\n", a, b, c);

    return 0;
}
```

假设键盘输入为：10 11 12↙

则输出为：

假设键盘输入为：12 ab 76↙

则输出为：

假设键盘输入为：10 -11 +12↙

则输出为：

假设键盘输入为：12 -ab +76↙

则输出为：

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

I.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    int a;

    scanf("%3d", &a);
    printf("a=%d\n", a);

    return 0;
}
```

假设键盘输入为：12345678↙

则输出为：

结论：%md中的m表示：指定输入数据所占的宽度，即输入到达该指定宽度时，超过该指定宽度的内容将不再作为输入内容，在该指定宽度内的内容为有效输入内容。

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

int main()
{
    int a, b;

    scanf("%3d %*2d %3d", &a, &b);
    printf("a=%d b=%d\n", a, b);

    return 0;
}
```

假设键盘输入为：12345678↙

则输出为：

结论：*md的*m表示：本输入项不赋给相应的变量

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

J. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a;
    scanf("%d", &a);
    printf("%d\n", a);
    return 0;
}
```

假设键盘输入为：
123↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123
123
```

假设键盘输入为：
123 456↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123 456
123
```

假设键盘输入为：
123a**↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123a**
123
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a;
    scanf("%x", &a);
    printf("%d\n", a);
    return 0;
}
```

假设键盘输入为：
123↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123
291
```

假设键盘输入为：
123 456↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123 456
291
```

假设键盘输入为：
123a**↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123a**
4666
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a;
    scanf("%3d", &a);
    printf("%d\n", a);
    return 0;
}
```

假设键盘输入为：
123↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123
123
```

假设键盘输入为：
123a**↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123a**
123
```

假设键盘输入为：
12a**↙
则输出为：

```
Microsoft Visual Studio 调试控制台
12a**
12
```

结论：

scanf输入的终止条件是回车、空格、非法输入和达到指定输入数据所占的宽度

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

K. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a, b;
    scanf("%3d%3d", &a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

输入：12↙ 345↙ ，输出：

```
Microsoft Visual Studio 调试控制台
12
345
12 345
```

输入：12↙ 3456↙ ，输出：

```
Microsoft Visual Studio 调试控制台
12
3456
12 345
```

输入：123↙456↙ ，输出：

```
Microsoft Visual Studio 调试控制台
123
456
123 456
```

输入：1234␣5678↙ ，输出：

```
Microsoft Visual Studio 调试控制台
1234 5678
123 4
```

输入：123456↙ ，输出：

```
Microsoft Visual Studio 调试控制台
123456
123 456
```

输入：12345678↙ ，输出：

```
Microsoft Visual Studio 调试控制台
12345678
123 456
```

输出第4项的结果的原因：
首先读取123，达到指定输入数据所占的宽度3，终止输入，将123赋值给变量a，之后读取4，遇到空格，终止输入，将4赋值给变量b。

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a, b;
    scanf("%3d%*2d%3d", &a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

输入：123456↙ ，输出：

```
Microsoft Visual Studio 调试控制台
123456
123 6
```

输入：12345678↙ ，输出：

```
Microsoft Visual Studio 调试控制台
12345678
123 678
```

输入：123456789↙ ，输出：

```
Microsoft Visual Studio 调试控制台
123456789
123 678
```

输入：123 45 678↙ ，输出：

```
Microsoft Visual Studio 调试控制台
123 45 678
123 678
```

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

L. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    float f;

    scanf("%f", &f);
    printf("f=%f\n", f);

    return 0;
}
```

假设键盘输入为：
123.45↙
则输出为：

Microsoft Visual Studio 调试控制台
```
123.45
f=123.449997
```

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

int main()
{
    float f;

    scanf("%lf", &f);
    printf("f=%f\n", f);

    return 0;
}
```

假设键盘输入为：
123.45↙
则输出为：（弹出错误窗口）

D:\Learning\高级语言程序设计\C-demo\Debug\C-demo.exe
```
123.45
f=-107374184.000000
```

```
warning C4477: "scanf": 格式字符串"%lf"需要类型"double *"的参数，但可变参数 1 拥有了类型"float *"
message : 请考虑在格式字符串中使用"%f"
```

第二组程序的错误信息截图

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

int main()
{
    double f;

    scanf("%lf", &f);
    printf("f=%f\n", f);

    return 0;
}
```

假设键盘输入为：
123.45↙
则输出为：

Microsoft Visual Studio 调试控制台
```
123.45
f=123.450000
```

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

int main()
{
    double f;

    scanf("%f", &f);
    printf("f=%f\n", f);

    return 0;
}
```

假设键盘输入为：
123.45↙
则输出为：

Microsoft Visual Studio 调试控制台
```
123.45
f=-9255960494527271546043667176075719548209639652554213037178800.000000
```

```
warning C4477: "scanf": 格式字符串"%f"需要类型"float *"的参数，但可变参数 1 拥有了类型"double *"
message : 请考虑在格式字符串中使用"%lf"
```

第四组程序的错误信息截图

结论：

1、附加格式控制符l的作用是输入double型数

2、如果格式控制符的数据类型和要读取的变量类型的字节大小不一致（例：4/8字节），则赋值后输出的值不可信

3、printf中，输出double型数据时，%f 和 %lf 无差别
scanf中， 输入double型数据时，%f 和 %lf 有差别

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

M.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    float f;
    scanf("%7.2f", &f);
    printf("%f\n", f);
    return 0;
}
```

```
warning C4476: "scanf": 格式说明符中的类型字段字符 "." 未知
warning C4474: scanf: 格式字符串中传递的参数太多
message : 占位符和其参数预计 0 可变参数，但提供的却是 1 参数
```

假设键盘输入为：1234.56↙
则输出为：

```
Microsoft Visual Studio 调试控制台
1234.56
-107374176.000000
```

假设键盘输入为：12.3456↙
则输出为：

```
Microsoft Visual Studio 调试控制台
12.3456
-107374176.000000
```

假设键盘输入为：123↙
则输出为：

```
Microsoft Visual Studio 调试控制台
123
-107374176.000000
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    float f;
    scanf("%7f", &f);
    printf("%f\n", f);
    return 0;
}
```

假设键盘输入为：1234.5678↙
则输出为：

```
Microsoft Visual Studio 调试控制台
1234.5678
1234.560059
```

假设键盘输入为：12.345678↙
则输出为：

```
Microsoft Visual Studio 调试控制台
12.345678
12.345600
```

假设键盘输入为：12345678↙
则输出为：

```
Microsoft Visual Studio 调试控制台
12345678
1234567.000000
```

结论：

1、%mf/%mlf如果指定了宽度m，则scanf在读取时只读取宽度为m的数据

2、%m.nf/%m.nlf如果指定了精度（小数点后的位数），则scanf的%f/%lf不支持.n形式的附加格式控制符，会得到不可信值

§.基础知识题 – C方式输入输出的格式化控制

2.格式化输入函数scanf的基本理解

N.观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    char c1, c2;
    scanf("%c %c", &c1, &c2);
    printf("c1=%c c2=%c\n", c1, c2);
    return 0;
}
```

假设键盘输入为：ABCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
ABCD
c1=A c2=B
```

假设键盘输入为：A BCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A BCD
c1=A c2=B
```

假设键盘输入为：'A' BCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
'A' BCD
c1=' c2=A
```

假设键盘输入为：\n↙
则输出为：

```
Microsoft Visual Studio 调试控制台
\n
c1=\ c2=n
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    char c1, c2;
    scanf("%c%c", &c1, &c2); //两个%c间无空格
    printf("c1=%d c2=%d\n", c1, c2);
    return 0;
}
```

假设键盘输入为：ABCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
ABCD
c1=65 c2=66
```

假设键盘输入为：A BCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A BCD
c1=65 c2=32
```

假设键盘输入为：'A' BCD↙
则输出为：

```
Microsoft Visual Studio 调试控制台
'A' BCD
c1=39 c2=65
```

假设键盘输入为：\n↙
则输出为：

```
Microsoft Visual Studio 调试控制台
\n
c1=92 c2=110
```

结论：1、%c只读 1 个字符
2、%c在输入转义符/单引号等特殊字符时，得到的是特殊字符自身的ASCII码
3、空格是scanf中%c方式的有效输入，但必须注意在格式控制表列中%c间无空格时，空格才会被视为有效输入，在格式控制表列中%c间有空格时，空格不会被视为有效输入

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

0. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    short ch;

    scanf("%c", &ch);
    printf("ch=%hd\n", ch);

    return 0;
}
```

假设键盘输入为：A↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A
ch=-13247
```

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

int main()
{
    int ch;

    scanf("%c", &ch);
    printf("ch=%d\n", ch);

    return 0;
}
```

假设键盘输入为：A↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A
ch=-858993599
```

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

int main()
{
    long ch;

    scanf("%c", &ch);
    printf("ch=%ld\n", ch);

    return 0;
}
```

假设键盘输入为：A↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A
ch=-858993599
```

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

int main()
{
    float ch;

    scanf("%c", &ch);
    printf("ch=%f\n", ch);

    return 0;
}
```

假设键盘输入为：A↙
则输出为：

```
Microsoft Visual Studio 调试控制台
A
ch=-107373064.000000
```

第一组程序错误信息截图

```
warning C4477: “scanf”：格式字符串“%c”需要类型“char *”的参数，但可变参数 1 拥有了类型“short *”
message : 请考虑在格式字符串中使用“%lc”
message : 请考虑在格式字符串中使用“%llc”
message : 请考虑在格式字符串中使用“%Lc”
message : 请考虑在格式字符串中使用“%wc”
```

第二组程序错误信息截图

```
warning C4477: “scanf”：格式字符串“%c”需要类型“char *”的参数，但可变参数 1 拥有了类型“int *”
```

第三组程序错误信息截图

```
warning C4477: “scanf”：格式字符串“%c”需要类型“char *”的参数，但可变参数 1 拥有了类型“long *”
```

第四组程序错误信息截图

```
warning C4477: “scanf”：格式字符串“%c”需要类型“char *”的参数，但可变参数 1 拥有了类型“float *”
```

结论：

%c方式读入时，地址表列中的变量不能是非字符类型。

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

P. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

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

int main()
{
    char s1[10], s2[10];//s1/s2是数组(后续内容)

    scanf("%s %s", s1, s2);
    printf("s1=%s\ns2=%s\n", s1, s2);

    return 0;
}
```

假设键盘输入为：tong ji↙
则输出为：

假设键盘输入为：tong↙
ji↙
则输出为：

假设键盘输入为：tong↙
hello1234↙
(9个字符)
则输出为：

假设键盘输入为：tong↙
hello12345↙
(10个字符)
则输出为：

假设键盘输入为：tongjiuniversity↙
(超过10个)
hello↙
则输出为：

结论：

1、%s不能读入含空格的字符串

2、%s输入时，如果数组的大小为n，则最多输入n-1个字符

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

Q. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    char s[80];
    scanf("%s", s);
    printf("%s\n", s);
    return 0;
}
```

假设键盘输入为：<u>"\r\n\tabc"</u>↙

则输出为：

该字符串真正的内存存储为 <u>10</u> 个字节，这些字节的值分别是<u>92、114、92、110、92、116、97、98、99、0</u>

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    char s[80], t[80];
    scanf("%s,%s", s,t);
    printf("s=%s\n", s);
    printf("t=%s\n", t);
    return 0;
}
```

假设键盘输入为：<u>abc,def</u>↙

则输出为：

与2-E不同，"%s,%s"之间的逗号是<u>当做第一个字符串的有效字符</u>

§. 基础知识题 – C方式输入输出的格式化控制

2. 格式化输入函数scanf的基本理解

R. 观察下列程序的运行结果，回答问题并将程序的运行结果截图贴上(如果有错则贴错误信息截图)

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a, ret;
    ret = scanf("%d", &a);
    printf("a=%d ret=%d\n", a, ret);
    return 0;
}
```

假设键盘输入为：10↙
则输出为：

```
Microsoft Visual Studio 调试控制台
10
a=10 ret=1
```

假设键盘输入为：10a↙
则输出为：

```
Microsoft Visual Studio 调试控制台
10a
a=10 ret=1
```

假设键盘输入为：abc↙
则输出为：

```
Microsoft Visual Studio 调试控制台
abc
a=-858993460 ret=0
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{
    int a, b, ret;
    ret = scanf("%d %d", &a, &b);
    printf("a=%d b=%d ret=%d\n", a, b, ret);
    return 0;
}
```

假设键盘输入为：10 20↙
则输出为：

```
Microsoft Visual Studio 调试控制台
10 20
a=10 b=20 ret=2
```

假设键盘输入为：10 20a↙
则输出为：

```
Microsoft Visual Studio 调试控制台
10 20a
a=10 b=20 ret=2
```

假设键盘输入为：10a20↙
则输出为：

```
Microsoft Visual Studio 调试控制台
10a20
a=10 b=-858993460 ret=1
```

假设键盘输入为：abc↙
则输出为：

```
Microsoft Visual Studio 调试控制台
abc
a=-858993460 b=-858993460 ret=0
```

结论：scanf返回值是正确按指定格式输入变量的个数，即能正确接收到值的变量个数

§.基础知识题 – 字符的输入与输出

1.putchar的基本使用

A.程序如下，观察编译及运行结果

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char ret1;
    cout << (ret1 = putchar('A')) << endl;
    int ret2;
    cout << (ret2 = putchar('B')) << endl;
    return 0;
}
```

1、观察运行结果

2、分析运行结果中各输出是哪个语句/函数造成的

第一行：

第一个A：putchar

第二个A：cout

第二行：

B：putchar

66：cout

3、这个例子能确认上个Page的基本知识中的说法：

"返回值是int型，是输出字符的ASCII码"

完全正确/部分正确吗？

不能。putchar的返回值被赋值给char型变量ret1或int型变量ret2，cout根据数据类型决定输出形式，所以我们不能确认“返回值是int型，是输出字符的ASCII码”是完全正确/部分正确的。

§.基础知识题 – 字符的输入与输出

1.putchar的基本使用

B.自行构造测试程序，证明putchar的返回值是int型而不是char型

```
//方法一

#include <iostream>
using namespace std;

int main()
{
        cout << typeid(putchar('A')).name()
             << endl;
}
```

Microsoft Visual Studio 调试控制台

```
int
```

```
//方法2

#include <iostream>
using namespace std;

int main()
{
        cout << sizeof(putchar('A')) << endl;
}
```

Microsoft Visual Studio 调试控制台

```
4
```

§.基础知识题 – 字符的输入与输出

2.getchar的基本使用

A.程序如下，观察编译及运行结果（可手填）

`#include <iostream>`
`#include <cstdio>`
`using namespace std;`
`int main()`
`{`
`  char ch;`
`  ch = getchar();`
`  cout << ch << endl;`
 	
`  return 0;`
`}`
	`#include <iostream>`
	`#include <cstdio>`
	`using namespace std;`
	`int main()`
	`{`
	`  char ch;`
	`  cout << (ch = getchar()) << endl;`
	
	`  return 0;`
	`}`
		`#include <iostream>`
		`#include <cstdio>`
		`using namespace std;`
		`int main()`
		`{`
		`  int ch;`
		`  ch = getchar();`
		`  cout << ch << endl;`
		
		`  return 0;`
		`}`

输入：a↙ 输出：a 输出的是：ch的值	输入：a↙ 输出：a 输出的是：赋值表达式值	输入：a↙ 输出：97

§.基础知识题 – 字符的输入与输出

2.getchar的基本使用

B.自行构造测试程序，证明getchar的返回值是int型而不是char型

```
//方法一

#include <iostream>
using namespace std;

int main()
{
        cout << typeid(getchar()).name()
             << endl;
}
```

Microsoft Visual Studio 调试控制台

```
int
```

```
//方法2

#include <iostream>
using namespace std;

int main()
{
        cout << sizeof(getchar()) << endl;
}
```

Microsoft Visual Studio 调试控制台

```
4
```

§.基础知识题 – 字符的输入与输出

2.getchar的基本使用

C.程序如下，观察编译及运行结果

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char ch;
    ch = getchar();
    cout << int(ch) << endl;

    return 0;
}
```

1、键盘输入：Hello↙ （5个字母+回车）

2、键盘输入：↙ （空回车）

3、键盘输入：␣↙ （空格+回车）

4、键盘输入：\n↙ （2个字符+回车）

5、键盘输入：\101↙ （4个字符+回车）

结论：

可以输入空格、回车等cin无法处理的非图形字符，但仍不能处理转义符。

§. 基础知识题 – 字符的输入与输出

2. getchar的基本使用

D. 程序如下，观察编译及运行结果

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    cout << "--Step1--" << endl;
    cout << getchar()   << endl;

    cout << "--Step2--" << endl;
    cout << getchar()   << endl;

    cout << "--Step3--" << endl;
    cout << getchar()   << endl;

    cout << "--Step4--" << endl;
    cout << getchar()   << endl;
    return 0;
}
```

1、每次输入一个回车
程序从开始执行到结束，共停顿了 4 次来等待输入
第1次停顿时，屏幕上输出的最后一行是Step1
第2次停顿时，屏幕上输出的最后一行是Step2
第3次停顿时，屏幕上输出的最后一行是Step3
第4次停顿时，屏幕上输出的最后一行是Step4

2、第一次输入一个字母+回车，以后每次停顿，均输入一个字母+回车
程序从开始执行到结束，共停顿了 2 次来等待输入
第1次停顿时，屏幕上输出的最后一行是Step1
第2次停顿时，屏幕上输出的最后一行是Step3
第3次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第4次停顿时，屏幕上输出的最后一行是Step(没有停顿)

3、第一次即输入4个以上的字母+回车
程序从开始执行到结束，共停顿了 1 次来等待输入
第1次停顿时，屏幕上输出的最后一行是Step1
第2次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第3次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第4次停顿时，屏幕上输出的最后一行是Step(没有停顿)

结论：getchar每次仅从输入缓冲区中取需要的字节，
多余的字节仍保留在输入缓冲区中供下次读取

思考：结合"cin与cout的基本使用"中3.c的例子，考虑一下3.c中非法m对int的影响（错在第几个数）与输入缓冲区的关系，为什么？
在输入缓冲区中，非法m之前的内容可以正常读取，其值是可信的，非法m之后的内容读取后，其值是不可信的。

Microsoft Visual Studio 调试控制台

```
--Step1--

10
--Step2--

10
--Step3--

10
--Step4--

10
```

Microsoft Visual Studio 调试控制台

```
--Step1--
a
97
--Step2--
10
--Step3--
b
98
--Step4--
10
```

Microsoft Visual Studio 调试控制台

```
--Step1--
abcde
97
--Step2--
98
--Step3--
99
--Step4--
100
```

§.基础知识题 – 字符的输入与输出

2.getchar的基本使用

E.自行构造证明D结论的使用cin读入的测试程序

```
#include <iostream>
using namespace std;
int main()
{
    char a, b, c, d;

    cout << "--Step1--" << endl;
    cin >> a;
    cout << a << endl;
    cout << "--Step2--" << endl;
    cin >> b;
    cout << b << endl;
    cout << "--Step3--" << endl;
    cin >> c;
    cout << c << endl;
    cout << "--Step4--" << endl;
    cin >> d;
    cout << d << endl;

    return 0;
}
```

因为cin不能读取空格、回车（有特殊方法可读，先忽略），因此测试有所不同

1、第一次输入两个字母+回车，以后每次停顿，均输入两个字母+回车
程序从开始执行到结束，共停顿了 2 次来等待输入
第1次停顿时，屏幕上输出的最后一行是Step1
第2次停顿时，屏幕上输出的最后一行是Step3
第3次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第4次停顿时，屏幕上输出的最后一行是Step(没有停顿)

2、第一次即输入4个以上的字母+回车
程序从开始执行到结束，共停顿了 1 次来等待输入
第1次停顿时，屏幕上输出的最后一行是Step1
第2次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第3次停顿时，屏幕上输出的最后一行是Step(没有停顿)
第4次停顿时，屏幕上输出的最后一行是Step(没有停顿)

结论：cin每次仅从输入缓冲区中取需要的字节，多余的字节仍保留在输入缓冲区中供下次读取

§. 基础知识题 – 字符的输入与输出

3. getchar、_getch与_getche的基本使用

A. 程序如下，观察编译及运行结果

1、测试时cmd窗口下面不能是中文输入法
2、<conio.h>是_getch()/_getche()需要的头文件

VS与Dev编译结果一致

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = getchar();
    cout << (int)ch << endl;

    return 0;
}
```

1、输入：a↙

输出：97

输入回显：有

按回车生效：是

```
Microsoft Visual Studio 调试控制台
a
97
```

2、输入：↙

输出：10

```
Microsoft Visual Studio 调试控制台
10
```

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    char ch;
    ch = _getch();
    cout << (int)ch << endl;

    return 0;
}
```

1、输入：a↙

输出：97

输入回显：无

按回车生效：否

```
Microsoft Visual Studio 调试控制台
97
```

2、输入：↙

输出：13

```
Microsoft Visual Studio 调试控制台
13
```

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    char ch;
    ch = _getche();
    cout << (int)ch << endl;

    return 0;
}
```

1、输入：a↙

输出：a97

输入回显：有

按回车生效：否

```
Microsoft Visual Studio 调试控制台
a97
```

2、输入：↙

输出：13

```
Microsoft Visual Studio 调试控制台
13
```

§. 基础知识题 – 字符的输入与输出

3. getchar、_getch与_getche的基本使用

B. 程序如下，观察编译及运行结果

1、测试时cmd窗口下面不能是中文输入法
2、<conio.h>是_getch()/_getche()需要的头文件

哪个编译器报错？VS

哪个编译器下结果同A？Dev

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    char ch;
    ch = getch();
    cout << (int)ch << endl;

    return 0;
}
```

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    char ch;
    ch = getche();
    cout << (int)ch << endl;

    return 0;
}
```

```
error C4996: 'getch': The POSIX name for this item is deprecated. Instead, use the ISO C and C++ conformant name: _getch. See online help for details.
error C4996: 'getche': The POSIX name for this item is deprecated. Instead, use the ISO C and C++ conformant name: _getche. See online help for details.
```

1、输入：a↙
输出：97
输入回显：无
按回车生效：否

```
D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe
97
```

1、输入：a↙
输出：a97
输入回显：有
按回车生效：否

```
D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe
a97
```

2、输入：↙
输出：13

```
D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe
13
```

2、输入：↙
输出：13

```
D:\Learning\高级语言程序设计\My Project\My Project\My Project.exe
13
```

§.基础知识题 - 字符数组的输入与输出

1.输入

逐个输入：scanf("%c",&数组元素)　　C方式

cin >> 数组元素　　C++方式

例1：C方式输入单个字符

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    scanf("%c%c", &a[3], &a[7]);

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

   return 0;
}
```

数组下标表示前有取地址符号&
因为scanf规定后面必须是变量的地址

```
scanf前首先输出10行，内容是：
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

scanf时，输入AB并回车，输出是：
-52（随机）
-52（随机）
-52（随机）
65
-52（随机）
-52（随机）
-52（随机）
66
-52（随机）
-52（随机）

//用不同颜色标注出有变化的内容
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
AB
-52
-52
-52
65
-52
-52
-52
66
-52
-52
```

§.基础知识题 – 字符数组的输入与输出

1.输入

逐个输入：scanf("%c",&数组元素)　　C方式

cin >> 数组元素　　C++方式

例2：C++方式输入单个字符

```
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    cin >> a[3] >> a[7];

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

   return 0;
}
```

数组下标表示前无取地址符号&

```
cin前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

cin时，输入AB并回车，输出是：
-52（随机）
-52（随机）
-52（随机）
65
-52（随机）
-52（随机）
-52（随机）
66
-52（随机）
-52（随机）

//用不同颜色标注出有变化的内容
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
AB
-52
-52
-52
65
-52
-52
-52
66
-52
-52
```

§.基础知识题 – 字符数组的输入与输出

1.输入

逐个输入：scanf("%c",&数组元素)　　C方式

cin >> 数组元素　　C++方式

例3：C方式多次逐个输入时回车的处理

```
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    scanf("%c%c", &a[3], &a[7]);
    scanf("%c", &a[0]);

   for(i=0; i<10; i++)
      cout << int(a[i]) << endl;

   return 0;
}
```

```
scanf前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

scanf时，输入AB并回车，输出是：
10
-52（随机）
-52（随机）
65
-52（随机）
-52（随机）
-52（随机）
66
-52（随机）
-52（随机）

//用不同颜色标注出有变化的内容
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
AB
10
-52
-52
65
-52
-52
-52
66
-52
-52
```

§.基础知识题 – 字符数组的输入与输出

1.输入

逐个输入：scanf("%c",&数组元素)　　C方式

cin >> 数组元素　　C++方式

例4：C++方式多次逐个输入时回车的处理

```
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    cin >> a[3] >> a[7];
    cin >> a[0];

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

cin前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

cin时，输入AB并回车，表现如何？

光标闪烁，正在等待读入字符。

多按几次回车，表现如何？

光标下移并闪烁，正在等待读入字符。

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
AB

C
67
-52
-52
65
-52
-52
-52
66
-52
-52
```

最后再输入C并回车，则输出是：
67
-52（随机）
-52（随机）
65
-52（随机）
-52（随机）
-52（随机）
66
-52（随机）
-52（随机）

//用不同颜色标注出有变化的内容

综合例3/4得到结论：当多次逐个输入时，

C方式处理回车的方式是：
在scanf("%c",&char)情况下将回车符作为合法的输入字符在输入缓冲区中被读取

C++方式处理回车的方式是：
回车符作为cin输入的终止条件（不作为合法的输入字符）不会在输入缓冲区中被读取

§.基础知识题 – 字符数组的输入与输出

1.输入

字符串形式：scanf("%s",数组名)　　C方式

cin >> 数组名　　C++方式

例5：C方式输入字符串(正确)

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    scanf("%s", a);

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

直接数组名，无下标，也不加&
因为C/C++规定，数组名代表数组的起始地址

scanf前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

等待键盘输入，输入Hello并回车，输出为
72
101
108
108
111
0
-52（随机）
-52（随机）
-52（随机）
-52（随机）
//用不同颜色标注出有变化的内容

问：

1、回车是否在数组中？

回车不在数组中。

2、Hello后面的一个字符是什么？

ASCII码为0的字符，即尾零('\0')。

§.基础知识题 – 字符数组的输入与输出

1.输入

字符串形式：scanf("%s",数组名)　　C方式

cin >> 数组名　　C++方式

例6：C方式输入字符串(错误)

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    scanf("%s", a);

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

直接数组名，无下标，也不加&
因为C/C++规定，数组名代表数组的起始地址

scanf前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
等待键盘输入：
测试1：输入9个及以内字符并回车，输出？（以输入12345678为例）
49
50
51
52
53
54
55
56
0
-52（随机）

输入9个及以内字符并回车，依次输出字符的ASCII码和一个尾零('\0')，若有未初始化的元素则输出随机值

测试2：输入10个及以上字符并回车，输出？（以输入0123456789为例）
48
49
50
51
52
53
54
55
56
57

输入10个及以上字符并回车，出现弹窗报错

问：如果要保证输入正确，输入的字符个数要<u>小于</u>定义的字符数组的长度

§.基础知识题 – 字符数组的输入与输出

1.输入

字符串形式：scanf("%s",数组名)　　C方式

cin >> 数组名　　C++方式

例7：C++方式输入字符串(正确)

```
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    cin >> a;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

直接数组名,无下标,也不加&

cin前首先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

等待键盘输入，输入Hello并回车，输出为
72
101
108
108
111
0
-52（随机）
-52（随机）
-52（随机）
-52（随机）
//用不同颜色标注出有变化的内容

问：

1、回车是否在数组中？

回车不在数组中。

2、Hello后面的一个字符是什么？

ASCII码为0的字符，即尾零('\0')。

§.基础知识题 – 字符数组的输入与输出

1.输入

字符串形式：scanf("%s",数组名)　　C方式

cin >> 数组名　　C++方式

例8：C++方式输入字符串(错误)

```
#include <iostream>
using namespace std;

int main()
{
    char a[10];
    int i;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    cin >> a;

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

直接数组名,无下标,也不加&

cin前首先输出10行，内容是

-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

等待键盘输入：

测试1：输入9个及以内字符并回车，输出？（以输入12345678为例）

49
50
51
52
53
54
55
56
0
-52（随机）

输入9个及以内字符并回车，依次输出字符的ASCII码和一个尾零('\0')，若有未初始化的元素则输出随机值

测试2：输入10个及以上字符并回车，输出？（以输入0123456789为例）

48
49
50
51
52
53
54
55
56
57

输入10个及以上字符并回车，出现弹窗报错

问：如果要保证输入正确，输入的字符个数要 小于 定义的字符数组的长度

§.基础知识题 – 字符数组的输入与输出

2.输出

逐个：printf("%c",数组元素)　　C方式

cout << 数组元素　　C++方式

例9：C/C++方式输出单个字符

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char a[]="Student";    //长度缺省为8

    cout << sizeof(a) << endl;

    printf("%c*\n", a[5]);

    cout << a[3] << '*' << endl;

    return 0;
}
//输出加*是为了确认只输出了一个字符
```

输出为：

8

n*

d*

§.基础知识题 – 字符数组的输入与输出

2.输出

逐个：printf("%c",数组元素)　　C方式

cout << 数组元素　　C++方式

例10：C/C++方式以单个字符+循环形式输出整个数组

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int i;
    char a[]="Student";

    for(i=0; i<7; i++)
        printf("%c", a[i]);
    cout << endl; //换行

    for(i=0; i<7; i++)
        cout << a[i];
    cout << endl; //换行

    return 0;
}
```

数组 a 缺省长度为8
输出[0]-[6]，尾零不输出

输出为：
Student
Student

Microsoft Visual Studio 调试控制台

```
Student
Student
```

§.基础知识题 – 字符数组的输入与输出

2.输出

逐个：printf("%c",数组元素)　　C方式

cout << 数组元素　　C++方式

例11：C/C++方式以单个字符+循环形式输出整个数组

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    int i;
    char a[]="Student";

    for(i=0; i<7; i++)
        printf("%c,", a[i]);
    cout << endl; //换行

    for(i=0; i<7; i++)
        cout << a[i] << '*';
    cout << endl; //换行

    return 0;
}
```

%c后面多一个，cout方式每个字符后面多一个*

输出为：

S, t, u, d, e, n, t,

S*t*u*d*e*n*t*

§.基础知识题 – 字符数组的输入与输出

2.输出

字符串形式：printf("%s",数组名) C方式

cout << 数组名 C++方式

例12：C/C++以字符串方式输出字符数组

```
#include <iostream>
using namespace std;

int main()
{
    char a[]="Student";

    printf("%s\n", a);
    cout << a << endl;

    return 0;
}
```

跟数组名
不是数组元素名

输出为：
Student
Student

Microsoft Visual Studio 调试控制台

```
Student
Student
```

问：
尾零输出了吗？如何证明？

尾零没有输出，可以通过右侧程序的输出结果来证明尾零没有输出。右侧程序的输出结果为：

Microsoft Visual Studio 调试控制台

```
Student*
Student*
```

```
#include <iostream>
using namespace std;

int main()
{
    char a[] = "Student";

    printf("%s*\n", a);
    cout << a << "*" << endl;

    return 0;
}
```

§.基础知识题 – 字符数组的输入与输出

2.输出

字符串形式：printf("%s",数组名) C方式

cout << 数组名 C++方式

例13：C/C++以字符串方式输出字符数组

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char a[]="Student\0china";

    cout << sizeof(a) << endl;

    printf("%s*\n", a);
    cout << a << '*' << endl;

    cout << a[12] << endl;

    return 0;
}
```

输出为：

14

Student*

Student*

a

Microsoft Visual Studio 调试控制台

```
14
Student*
Student*
a
```

问1：从本例的结果可知，数组a的长度是 14 。

最后是否还有隐含的\0？ 是 。

a中的字符串的长度是 7 。

问2：字符串形式输出字符数组，如果数组中包含显式'\0'，则输出到 第一个显式'\0'(不包括'\0') 为止。

§.基础知识题 – 字符数组的输入与输出

2.输出

字符串形式：printf("%s",数组名) C方式

cout << 数组名 C++方式

例14：C/C++以字符串方式输出字符数组(不含尾零)

```
#include <iostream>
using namespace std;

int main()
{
    //注意：不能以字符串方式初始化
    char a[5]={'C','h','i','n','a'};

    printf("%s\n", a);
    cout << a << endl;

    return 0;
}
```

输出为：（每次输出表现随机）

China烫烫烫虜?0t

China烫烫烫虜?0t

问1：为什么会有乱字符？

输出跟数组名时字符数组是字符串方式输出（从数组的起始地址开始依次输出各字符的值，到\0为止），数组a是不含\0的数组，当字符串方式输出不含\0的数组时，会持续越界输出直到碰见\0为止，错误的具体表现为正常/乱码等多种形式。

问2：如果%s方式换成下面形式，还会看到乱字符吗？为什么？

```
int i;
for (i=0; i<5; i++)
    printf("%c", a[i]);
```

不会看到乱字符，因为依次输出字符数组中的每一个字符不是以字符串方式输出，不会出现越界的情况。

§.基础知识题 – 字符数组的输入与输出

2.输出

字符串形式：printf("%s",数组名) C方式

cout << 数组名 C++方式

例15：C/C++以字符串方式输出字符数组(不含尾零)

```
#include <iostream>
using namespace std;

int main()
{
    char a[5]; //不初始化

    printf("%s\n", a);
    cout << a << endl;

    return 0;
}
```

输出为：（每次输出表现随机）

烫烫烫烫烫烫畅IX,鸦

烫烫烫烫烫烫畅IX,鸦

问1：为什么会有乱字符？

char型数组a未进行初始化，当数组为自动变量时，数组元素的值不确定，输出跟数组名时字符数组是字符串方式输出，即从数组的起始地址开始依次输出各字符的值，到\0为止，由于数组元素的值不确定，所以会持续随机输出直到碰见\0为止，错误的具体表现为正常/乱码等多种形式。

问2：乱字符出现几行是正常的？一行？多行？或者都正常？

都正常。

结论：不能字符串形式输出不含 ’\0’ 的字符数组，否则可能会得到不正确的结果。

§.基础知识题 – 字符数组的输入与输出

3.从任一元素开始以字符串形式输入/输出

例16：从任一元素开始以字符串形式输出

```
#include <iostream>
using namespace std;

int main()
{
    char a[]="Student";

    printf("%s\n", &a[3]);

    cout << &a[3] << endl;

    return 0;
}
```

%s形式

&数组元素名形式

输出为：
dent
dent

Microsoft Visual Studio 调试控制台

```
dent
dent
```

§.基础知识题 – 字符数组的输入与输出

3.从任一元素开始以字符串形式输入/输出

例17：C方式从任一元素开始以字符串形式输入

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    int i;
    char a[10];

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    scanf("%s", &a[3]);

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

&数组元素名形式

scanf先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

等待键盘输入，输入Hello并回车，
输出为
-52（随机）
-52（随机）
-52（随机）
72
101
108
108
111
0
-52（随机）
//用不同颜色标注出有变化的内容

§.基础知识题 – 字符数组的输入与输出

3.从任一元素开始以字符串形式输入/输出

例18：C++方式从任一元素开始以字符串形式输入

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    char a[10];

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    cin >> &a[3];

    for(i=0; i<10; i++)
        cout << int(a[i]) << endl;

    return 0;
}
```

&数组元素名形式

cin先输出10行，内容是
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）
-52（随机）

等待键盘输入，输入Hello并回车，输出为
-52（随机）
-52（随机）
-52（随机）
72
101
108
108
111
0
-52（随机）
//用不同颜色标注出有变化的内容

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
```

```
-52
-52
-52
-52
-52
-52
-52
-52
-52
-52
Hello
-52
-52
-52
72
101
108
108
111
0
-52
```

综合例16-18的结果，得出的结论是：

C/C++方式从任一元素开始以字符串形式输入输出时，表示形式都是 &数组元素名 的形式

§.基础知识题 - 字符数组的输入与输出

1-3.总结

	C方式	C++方式
输入单个字符	scanf("%c", &元素名)	cin >> 元素名
输入字符串	scanf("%s", 数组名)	cin >> 数组名
输出单个字符	printf("%c", 元素名)	cout << 元素名
输出字符串	printf("%s", 数组名)	cout << 数组名
任一元素开始输入串	scanf("%s", &元素名)	cin >> &元素名
任一元素开始输出串	printf("%s", &元素名)	cout << &元素名

§.基础知识题 – 字符数组的输入与输出

4.多个字符串的输入

例19：C方式多个字符串的输入

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char a[10], b[20];

    scanf("%s%s", a, b);

    printf("%s-%s\n", a, b);

    return 0;
}
```

1、假设输入为abc空格def并回车

则输出为：
abc-def

2、假设输入为abc回车
def回车

则输出为：
abc-def

结论：空格是 输入分隔符 。

§.基础知识题 – 字符数组的输入与输出

4.多个字符串的输入

例20：C++方式多个字符串的输入

```
#include <iostream>
using namespace std;

int main()
{
    char a[10], b[20];

    cin >> a >> b;

    cout << a << '-' << b << endl;

    return 0;
}
```

1、假设输入为abc空格def并回车

则输出为：
abc-def

2、假设输入为abc回车
def回车

则输出为：
abc-def

结论：空格是 输入分隔符 。

综合例19-20可知：
scanf/cin从键盘上输入的字符串不能包含 空格和回车等非图形字符 。

§.基础知识题 – 字符数组的输入与输出

4.多个字符串的输入

★ 从键盘输入含空格字符串的方法(不同编译器不同)

例21：VS下用gets_s输入含空格的字符串（VS2022：有gets_s，无gets，有fgets）

```
#include <iostream>
using namespace std;

int main()
{
    char a[10], b[20];

    gets_s(a);
    gets_s(b);

    cout << a << endl;
    cout << b << endl;

    return 0;
}
```

1、键盘输入abc空格def并回车，会继续等待输入，再输入xyz并回车，则输出为：

abc def

xyz

2、键盘输入超过9个字符，观察弹窗报错。

3、键盘先输入Hello并回车，再输入超过19个字符，观察弹窗报错。

问：为什么a最长输入只能是9？

字符串方式输出下\0表示结束，所以要保证输入正确，输入的字符个数要小于定义的字符数组的长度，所以a最长输入只能是9。

为什么b最长输入只能是19？

字符串方式输出下\0表示结束，所以要保证输入正确，输入的字符个数要小于定义的字符数组的长度，所以a最长输入只能是19。

§.基础知识题 – 字符数组的输入与输出

4.多个字符串的输入

★ 从键盘输入含空格字符串的方法(不同编译器不同)

例22：Dev C++下用gets输入含空格的字符串（Dev C++：有gets，无gets_s，有fgets）

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char a[10], b[20];

    gets(a);
    gets(b);

    cout << a << endl;
    cout << b << endl;

    return 0;
}
```

1、键盘输入abc空格def并回车，会继续等待输入，再输入xyz并回车，则输出为：abc def
xyz

2、键盘输入超过9个字符，观察：

```
return value 3221225477
```

3、键盘先输入Hello并回车，再输入超过19个字符，观察：

```
C:\Us
abc def
xyz
abc def
xyz
```

```
C:\Users\lenovo\D
1abcdefghijklmn
2abcdefghijklmn
1abcdefghijklmn
2abcdefghijklmn
```

```
C:\Users\lenovo\Desktop\demo.exe
Hello
abcdefghijklmnopqrstuvwxyz
uvwxyz
abcdefghijklmnopqrstuvwxyz
```

问：为什么a最长输入只能是9？

字符串方式输出下\0表示结束，所以要保证输入正确，输入的字符个数要小于定义的字符数组的长度，所以a最长输入只能是9。

为什么b最长输入只能是19？

字符串方式输出下\0表示结束，所以要保证输入正确，输入的字符个数要小于定义的字符数组的长度，所以a最长输入只能是19。

§.基础知识题 – 字符数组的输入与输出

4.多个字符串的输入

★ 不同编译器从键盘输入含空格字符串的方法不同

例23：VS和Dev C++均可用fgets输入含空格的字符串（fgets函数的原型定义为：fgets(字符数组名，最大长度，stdin)；）

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
    char a[10], b[20];

    fgets(a,10,stdin);
    fgets(b,20,stdin);

    cout << a << endl;
    cout << b << endl;

    int i;
    for(i=0; a[i]!='\0'; i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    for(i=0; b[i]!='\0'; i++)
        cout << int(b[i]) << ' ';
    cout << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
abcdefghijklmnopqrstuvwxyzab
abcdefghi
jklmnopqrstuvwxyzab
97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 97 98
```

Microsoft Visual Studio 调试控制台

```
abcdefghijklmnopqrstuvwxyzabc
abcdefghi
jklmnopqrstuvwxyzab
97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 97 98
```

1、键盘输入abc空格def并回车，会继续等待输入，再输入xyz并回车，则输出为：

```
abc def

xyz

97 98 99 32 100 101 102 10
120 121 122 10
```

问1：和例21-22的输出区别在哪里？
例21-22没有输出回车符，例23输出回车符。

问2：后面两段红色代码的目的是什么？
输出读入的每个字符的ASCII码。

Microsoft Visual Studio 调试控制台

```
abc def
xyz
abc def

xyz

97 98 99 32 100 101 102 10
120 121 122 10
```

2、键盘输入9个字符并回车，则输出为：

```
abcdefghi


97 98 99 100 101 102 103 104 105
10
```

Microsoft Visual Studio 调试控制台

```
abcdefghi
abcdefghi


97 98 99 100 101 102 103 104 105
10
```

3、如果输入28个字符并回车，则输出为：

```
abcdefghi
jklmnopqrstuvwxyzab
97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 97 98
```

4、如果输入超过28个字符并回车，则输出为：

```
abcdefghi
jklmnopqrstuvwxyzab
97 98 99 100 101 102 103 104 105
106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 97 98
```

§.基础知识题 – 字符数组的输入与输出

5.二维字符数组的输入/输出

★ 数组名加双下标表示元素，单下标表示一维数组

例24：二维字符数组以双下标形式输出单个字符/单下标形式输出字符串

```
#include <iostream>
using namespace std;

int main()
{
    char a[3][30]={"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                   "abcdefghijklmnopqrstuvwxyz",
                   "0123456789" };
    // 单个字符输出(数组名+双下标)
    printf("a[0][2]=%c\n", a[0][2]);
    cout << "a[1][20]=" << a[1][20] << endl;

    // 字符串输出(数组名+单下标)
    printf("a[0]=%s\n",a[0]);
    cout << "a[2]=" << a[2] << endl;

    return 0;
}
```

输出为：

```
a[0][2]=C
a[1][20]=u
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[2]=0123456789
```

Microsoft Visual Studio 调试控制台

```
a[0][2]=C
a[1][20]=u
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[2]=0123456789
```

§.基础知识题 – 字符数组的输入与输出

5.二维字符数组的输入/输出

★ 数组名加双下标表示元素，单下标表示一维数组

例25：二维字符数组以双下标形式输入单个字符

```
#define _CRT_SECURE_NO_WARNINGS  //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[3][30]={"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                   "abcdefghijklmnopqrstuvwxyz",
                   "0123456789" };
    // 单字符输入(数组名+双下标)
    scanf("%c\n", &a[0][2]); //格式符为%c
    cin >> a[1][20];         //无&

    // 字符串输出(数组名+单下标)
    printf("a[0]=%s\n", a[0]);
    cout << "a[1]=" << a[1] << endl;

    return 0;
}
```

1、键盘输入#@并回车，输出为：

a[0]=AB#DEFGHIJKLMNOPQRSTUVWXYZ
a[1]=abcdefghijklmnopqrst@vwxyz

2、键盘输入#并回车，输入@并回车，输出为：

a[0]=AB#DEFGHIJKLMNOPQRSTUVWXYZ
a[1]=abcdefghijklmnopqrst@vwxyz

§.基础知识题 – 字符数组的输入与输出

5.二维字符数组的输入/输出

★ 数组名加双下标表示元素，单下标表示一维数组

例26：二维字符数组以单下标形式输入字符串

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[3][30]={"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                   "abcdefghijklmnopqrstuvwxyz",
                   "0123456789" };

    scanf("%s", a[1]);//a[1]是一维数组名,无&

    cout << "a[0]=" << a[0] << endl;
    cout << "a[1]=" << a[1] << endl;
    cout << "a[2]=" << a[2] << endl;

    return 0;
}
```

1、输入≤29个字符，输出为：（以连续输入10个#为例）

```
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=##########
a[2]=0123456789
```

2、输入30-59个字符，输出为：（以连续输入35个#为例）

```
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=###################################
a[2]=#####
```

3、输入60个以上字符，输出为：（以连续输入70个#为例）（弹窗报错）

```
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=##############################################################################
a[2]=##########################################
```

将scanf换为cin>>a[1];再重复1、2、3，观察结果：<u>结果相同</u>

问1：输入30~59个字符为什么不出现错误？a[2]中是什么？

第31~59个字符赋值给a[2]数组，未超过整个二维字符数组的合法空间，并未越界，所以不出现错误。a[2]中是第31~59个字符。

问2：简述你是怎么理解二维数组越界的？

可以将二维数组看成一维数组，二维数组在内存中也是连续存储的。数组的下标是有范围限制的（二维数组的下标范围为0~行数*列数-1），所以二维数组的下标小于0或者大于行数*列数-1时，都是越界访问，超出了数组的合法空间。

§.基础知识题 – 字符数组的输入与输出

5.二维字符数组的输入/输出

★ 数组名加双下标表示元素，单下标表示一维数组

例27：二维字符数组从任一位置开始输出字符串

```
#include <iostream>
using namespace std;

int main()
{
    char a[3][30]={"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                   "abcdefghijklmnopqrstuvwxyz",
                   "0123456789" };

    //（第1组）单字符输出(数组名+双下标)
    printf("a[0][2]=%c\n", a[0][2]);
    cout << "a[1][20]=" << a[1][20] << endl;

    //（第2组）字符串输出(&+数组名+双下标)
    printf("a[0][2]=%s\n", &a[0][2]);
    cout << "a[1][20]=" << &a[1][20] << endl;

    //（第3组）字符串输出(数组名+单下标)
    printf("a[0]=%s\n",a[0]);
    cout << "a[2]=" << a[2] << endl;

    return 0;
}
```

输出为：
a[0][2]=C
a[1][20]=u
a[0][2]=CDEFGHIJKLMNOPQRSTUVWXYZ
a[1][20]=uvwxyz
a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[2]=0123456789

问1：同样双下标形式（第1/2组），
怎样输出单个字符？ C方式 ：printf("%c", 数组名+双下标)
C++方式：cout << 数组名+双下标
怎样输出字符串？ C方式 ：printf("%s", &+数组名+双下标)
C++方式：cout << &+数组名+双下标

问2：如何修改第2组的输出（必须保持双下标形式不变），使输出结果与第3组一致？
改为对应一维数组中第一个元素的地址，即&a[0][2]改为&a[0][0]，&a[1][20]改为&a[2][0]。

§.基础知识题 – 字符数组的输入与输出

5.二维字符数组的输入/输出

★ 数组名加双下标表示元素，单下标表示一维数组

例28：二维字符数组从任一位置开始输入字符串

```
#define _CRT_SECURE_NO_WARNINGS   //VS需要
#include <iostream>
using namespace std;

int main()
{
    char a[3][30]={"ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                   "abcdefghijklmnopqrstuvwxyz",
                   "0123456789" };

    scanf("%s", &a[1][3]);//&+数组名+双下标

    cout << "a[0]=" << a[0] << endl;
    cout << "a[1]=" << a[1] << endl;
    cout << "a[2]=" << a[2] << endl;

    return 0;
}
```

1、输入≤26个字符，输出为：（以连续输入10个#为例）

a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=abc##########
a[2]=0123456789

2、输入27-56个字符，输出为：（以连续输入30个#为例）

a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=abc##############################
a[2]=###

3、输入56个以上字符，输出为：（以连续输入60个#为例）（弹窗报错）

a[0]=ABCDEFGHIJKLMNOPQRSTUVWXYZ
a[1]=abc##
a[2]=##################################

将scanf换为cin>>&a[1][3];再重复1、2、3，观察结果：<u>结果相同</u>

问1：输入27~56个字符为什么不出现错误？a[2]中是什么？

第28~56个字符赋值给a[2]数组，未超过整个二维字符数组的合法空间，并未越界，所以不出现错误。a[2]中是第28~56个字符。

问2：如果想不影响a[2]，例26中是≤29个字符，本例中是≤26个字符，差别在哪？

例26中是从一维数组中的第一个元素开始赋值，所以若想不影响a[2]应该≤29个字符，例28中是从一维数组中的第四个元素开始赋值，所以若想不影响a[2]应该≤26个字符。

§.基础知识题 – 字符数组的输入与输出

6.尾零的输出

例29：在不同的控制台及字体设置下尾零输出的差异

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    char a[10] = { 'c','h','i','n','a' };

    cout << "0         1         2         " << endl; //标尺
    cout << "0123456789012345678901234567 89" << endl; //标尺

    for (i = 0; i < 10; i++)
        cout << a[i] << '$'; //确认a[i]是否输出

    cout << '#' << endl; //加行尾识别符

    return 0;
}
```

1、新版控制台+新宋体28点阵

2、旧版控制台+新宋体28点阵

3、旧版控制台+新宋体16点阵

结论：

1、不要以字符形式输出\0，因为看到的内容 不可信 。

2、如果想准确得知某字符的值，转为 int 类型输出即可。

§.基础知识题 – 字符数组的输入与输出

6.尾零的输出

例30：在不同的控制台及字体设置下其它非图形字符输出的差异

（去ASCII码表中查表示扑克牌四种花色的字符，用测试程序打印含这4个字符的字符串）

```
#include <iostream>
using namespace std;

int main()
{
    char a[5] = "\3\4\5\6";
    cout << a;

    return 0;
}
```

1、新版控制台+点阵字体10×18点阵

2、新版控制台+新宋体28点阵

结论

上页的结论1也 适用 于其它非图形字符。

§.基础知识题 – C方式常用的字符串处理函数

1.strlen(const char s[])

例1：字符数组与字符串长度

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[]="Hello";
    cout << sizeof(str1) << endl;
    cout << strlen(str1) << endl;

    char str2[]="china\0Hello\0\0";
    cout << sizeof(str2) << endl;
    cout << strlen(str2) << endl;

    return 0;
} //读操作，不需要加_CRT_SECURE_NO_WARNINGS
```

//给出程序的运行结果

问题：

1、求数组长度时，无论是否有显式\0，最后一定有 隐式 的\0。

2、当含有多个\0(显式/隐式)时，字符串长度计算到 第一 个\0为止。

§.基础知识题 – C方式常用的字符串处理函数

2. strcat(char dst[], const char src[])

例2：字符串连接

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[30]="Tongji ";  //不能缺省，至少18字节
    char str2[]="University";
    cout << strcat(str1, str2) << '#' << endl;  //加#的目的：判断连接后字符串末尾位置

    return 0;
}
```

//给出程序的运行结果

问题：

1、str2数组的默认长度是 11 。

2、结合前面字符数组输入/输出的作业，strcat复制时 包含 src的\0。

§.基础知识题 – C方式常用的字符串处理函数

2. strcat(char dst[], const char src[])

例3：字符串连接（错误）

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[]="Tongji ";
    char str2[]="University";
    cout << strcat(str1, str2) << '#' << endl; //加#的目的：判断连接后字符串末尾位置

    return 0;
}
```

//给出程序的运行结果

问题：

1、str1数组的大小必须给出，不能默认，其最小长度是 18 （针对本例的一个具体数字）。

2、dst数组的最小长度是 dst和src两字符串总长度+1 才能保证正确。

§.基础知识题 - C方式常用的字符串处理函数

3. strncat(char dst[], const char src[], const unsigned int n)

例4：字符串连接前n个字符

```
//例：字符串连接前n个字符
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[30]="Tongji ";
    char str2[30]="Tongji ";
    char str3[]="University";
    cout << strncat(str1, str3, 3) << '*' << endl;
    cout << strncat(str2, str3, 300) << '*' << endl;

    return 0;
}
```

//给出程序的运行结果

问题：

但n超过src表示的字符串的长度时，连接规则是 <u>只连接src字符串长度个字符</u> 。

§.基础知识题 – C方式常用的字符串处理函数

3. strncat(char dst[], const char src[], const unsigned int n)

例5：字符串连接前n个字符（错误）

```
//例：字符串连接前n个字符
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[]="Tongji ";

    char str3[]="University"; //缺省长度为11
    cout << strncat(str1, str3, 3) << '*' << endl;

    return 0;
}
```

//给出程序的运行结果

问题：

1、str1数组的大小必须给出，不能默认，其最小长度是 11 （针对本例的一个具体数字）。

2、dst数组的最小长度是 原dst字符串长度+min(dst字符串长度,len)+1 才能保证正确。

§.基础知识题 - C方式常用的字符串处理函数

4.strcpy(char dst[], const char src[])

例6：字符串拷贝

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int i;
    char a[]="student", b[]="hello";
    strcpy(a,b);
    cout << a << endl;
    for(i=0;i<8;i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    return 0;
}
```

//给出程序的运行结果

问题：

1、字符串复制时，复制到src的 \0 为止， 包含 \0，之后的字符不再复制。

2、在运行截图中用箭头指出证明结论1的位置。

§.基础知识题 – C方式常用的字符串处理函数

4. strcpy(char dst[], const char src[])

例7：字符串拷贝

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int i;
    char a[]="student", b[]="hello\0china";
    strcpy(a,b);
    cout << a << endl;
    for(i=0;i<8;i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    return 0;
}
```

//给出程序的运行结果

问题：1、a数组的默认大小是 8 ，b数组的默认大小是 12 。

2、b数组的大小超过了a数组的大小，为什么运行不出错？字符串复制时，复制到src的\0为止，包含\0，之后的字符不再复制，因此复制的字符数量为6，未超过a数组的大小。

3、本例中，复制到b[5]就停止复制了。

§.基础知识题 – C方式常用的字符串处理函数

4. strcpy(char dst[], const char src[])

例8：字符串拷贝（有错）

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int i;
    char a[11]="student", b[]="hellochina";
    strcpy(a,b);
    cout << a << endl;
    for(i=0; a[i]!='\0'; i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    return 0;
}
```

//给出程序的运行结果

问题：1、本程序为什么会错？<u>因为b数组的大小超过了a数组的大小，且复制的字符数量超过了a数组的大小。</u>

2、仅改a的定义使正确，如何做？（直接在源程序中修改）

3、dst数组的最小长度是 <u>max(dst字符串长度,src字符串长度)+1</u> 才能保证正确。

§.基础知识题 - C方式常用的字符串处理函数

5. strncpy(char dst[], const char src[], unsigned int n)

例9：字符串拷贝前n个字符

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int i;
    char a[]="student", b[]="hello";
    strncpy(a, b, 2);
    cout << a << endl;
    for(i=0;i<8;i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    return 0;
}
```

//给出程序的运行结果

问题：

本程序证明了strncpy复制时，<u>不包含</u> \0。

§.基础知识题 - C方式常用的字符串处理函数

5. strncpy(char dst[], const char src[], unsigned int n)

例10：字符串拷贝前n个字符

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int i;
    char a[]="student", b[]="hello";
    strncpy(a, &b[2], 2);
    cout << a << endl;
    for(i=0;i<8;i++)
        cout << int(a[i]) << ' ';
    cout << endl;

    return 0;
}
```

//给出程序的运行结果

问题：

如果想从b[2]开始复制2个字符到a中，如何做？（期望输出：lludent）

（直接在源程序中修改）

§.基础知识题 – C方式常用的字符串处理函数

5. strncpy(char dst[], const char src[], unsigned int n)

例11：字符串拷贝前n个字符（深度讨论）

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    int i;
    char a[] = "student", b[] = "hello";
    for (i = 0; i < 12; i++) //12已越界，目的：
        cout << int(a[i]) << ' ';
    cout << endl;

    strncpy(a, b, 200);
    cout << a << endl;

    for (i = 0; i < 12; i++) //12已越界，目的：
        cout << int(a[i]) << ' ';
    cout << endl;
    return 0;
}
```

观察数组越界后的输出内容，判断strncpy中n超过字符数组src的长度时拷贝是否到\0为止

观察数组越界后的输出内容，判断strncpy中n超过字符数组src的长度时拷贝是否到\0为止

//给出VS下程序的运行结果

Microsoft Visual Studio 调试控制台

```
115 116 117 100 101 110 116 0 -52 -52 -52 -52
hello
104 101 108 108 111 0 0 0 0 0 0 0
```

//给出Dev下程序的运行结果

C:\Users\lenovo\Desktop\demo.exe

```
115 116 117 100 101 110 116 0 8 0 0 0
hello
104 101 108 108 111 0 0 0 8 0 0 0
```

问题：

观察两个for循环的后6个数字的输出，能得到什么结论？

<u>strncpy中n超过字符数组src的长度时，拷贝不是到\0为止，继续拷贝导致出错。</u>

§.基础知识题 – C方式常用的字符串处理函数

5. strncpy(char dst[], const char src[], unsigned int n)

例12：字符串拷贝前n个字符（深度讨论）

```
#define _CRT_SECURE_NO_WARNINGS //VS需要
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    int i;
    char a[] = "student", b[] = "hello";
    for (i = 0; i < 20; i++) //20已越界，目的：
        cout << int(a[i]) << ' ';
    cout << endl;

    strncpy(a, b, 200);
    cout << a << endl;

    for (i = 0; i < 20; i++) //20已越界，目的：
        cout << int(a[i]) << ' ';
    cout << endl;
    return 0;
}
```

同上例，数组越界到20

观察数组越界后的输出内容，判断strncpy中n超过字符数组src的长度时拷贝是否到\0为止

观察数组越界后的输出内容，判断strncpy中n超过字符数组src的长度时拷贝是否到\0为止

//给出VS下程序的运行结果

Microsoft Visual Studio 调试控制台

```
115 116 117 100 101 110 116 0 -52 -52 -52 -52 -52 -52 -52 -52 16 0 0 0
hello
104 101 108 108 111 0 0 0 0 0 0 0 0 0 0 0 16 0 0 0
```

//给出Dev下程序的运行结果

C:\Users\lenovo\Desktop\demo.exe

```
115 116 117 100 101 110 116 0 8 0 0 0 32 -68 64 0 -32 -2 120 0
hello
104 101 108 108 111 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0
```

问题：

如果n超过了src的长度，则 <u>继续拷贝导致出错</u> 。

§.基础知识题 - C方式常用的字符串处理函数

6.strcmp(const char s1[], const char s2[])

例13：字符串比较

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{   char str1[] = "house", str2[]  = "horse";
    char str3[] = "abcd",  str4[]  = "abcde";
    char str5[] = "abcd",  str6[]  = "abc";
    char str7[] = "abcd",  str8[]  = "abcd";
    char str9[] = "abcd",  str10[] = "abcd\0efgh";
    cout << strcmp(str1, str2)  << endl;
    cout << strcmp(str3, str4)  << endl;
    cout << strcmp(str5, str6)  << endl;
    cout << strcmp(str7, str8)  << endl;
    cout << strcmp(str9, str10) << endl;
    return 0;
}
```

//给出程序的运行结果

问题：两个字符串相等的条件是？

<u>字符串长度相等且对应位置字符的ASCII码相同。</u>

§.基础知识题 – C方式常用的字符串处理函数

6.strcmp(const char s1[], const char s2[])

例14：字符串比较（另一种形式）

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char str1[]="abcd", str2[]="abcde";
    int k = strcmp(str1, str2);
    if (k==0)
        cout << "串1 = 串2" << endl;
    else if (k<0)
        cout << "串1 < 串2" << endl;
    else
        cout << "串1 > 串2" << endl;

    return 0;
}
```

//给出程序的运行结果

问题：给出两个字符串比较的执行过程。

两个字符串自左向右逐个字符相比（按ASCII值大小相比较），先比较第一个字符，如果第一个字符不相等则返回非0，如果第一个字符相等则比较下一个字符，自左向右逐个字符相比，直到出现不同的字符或遇到'\0'为止。

§.基础知识题 – C方式常用的字符串处理函数

6. strcmp(const char s1[], const char s2[])

例15：字符串比较（编译不报错但运行结果与期望不符合）

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[]="house", str2[]="horse";
    int k;

    k = str1 < str2;
    cout << k << endl;

    return 0;
}
```

//给出程序的运行结果

//将str1和str2的内容互换，
给出运行结果

//将str1和str2都置为"house"，
给出运行结果

问题：

这个程序的运行结果是表示str1和str2的 首地址 进行比较。

§.基础知识题 – C方式常用的字符串处理函数

7. strncmp(const char s1[], const char s2[], const unsigned int n)

例16：字符串比较前n个字符

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str1[] = "abcd",  str2[]  = "abcde";

    cout << strncmp(str1, str2, 3)   << endl;
    cout << strncmp(str1, str2, 4)   << endl;
    cout << strncmp(str1, str2, 5)   << endl;
    cout << strncmp(str1, str2, 100) << endl;

    return 0;
}
```

//给出程序的运行结果

//将str2也置为"abcd"
给出程序的运行结果

问题：

1、当n小于短串长度时，则比较到 第n个字符 。

2、当n大于等于短串长度时，则比较到 短串末尾的\0 为止。

3、如果n超过长串的长度，则比较到 短串末尾的\0 为止。

§.基础知识题 - C方式常用的字符处理函数

C方式常用的字符处理函数

#include <ctype.h> //C方式常用的字符处理函数所需头文件

```
isdigit(char ch);  //判断是否数字(0~9)
isalpha(char ch);  //判断是否字母(A~Z, a~z)
isalnum(char ch);  //判断是否字母或数字(A~Z, a~z, 0~9)
isxdigit(char ch); //判断是否16进制数字(0~9, A~F, a~f)
isspace(char ch);  //判断是否空格(含回车/换行/换页/TAB/竖向TAB)
islower(char ch);  //判断是否小写字母(a~z)
isupper(char ch);  //判断是否大写字母(A~Z)
tolower(char ch);  //大写转小写(其余原值返回)
toupper(char ch);  //小写转大写(其余原值返回)
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例1：cin.get()

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    ch = cin.get();
    cout << ch << int(ch) << endl;
    ch = cin.get();
    cout << ch << int(ch) << endl;

    return 0;
}
```

输入一个字符+回车，输出：该字符和该字符的ASCII码
（以输入a为例）　　空行
　　回车的ASCII码10

输入一串字符+回车，输出：第一个字符及其ASCII码
（以输入abc为例）　　第二个字符及其ASCII码

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例2：cin.get()

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while((ch = cin.get())!='\n')
        cout << ch;
    cout << endl;

    return 0;
}
```

输入一串字符+回车，输出：输入的字符串

（以输入abc为例）

Microsoft Visual Studio 调试控制台

```
abc
abc
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例3：cin.get()

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while((ch = cin.get())!=EOF)
        cout << ch;
    cout << endl;

    return 0;
}
```

输入：连续多个一串字符+回车，串中可含CTRL+Z

（以输入123，回车，输入a+CTRL+Z+b，回车，输入CTRL+Z为例）

输出：第一个CTRL+Z之前的所有字符和一个不可见字符

输入：连续多个一串字符+回车，最后一行单独CTRL+Z

（以输入123，回车，输入abc，回车，输入CTRL+Z为例）

输出：第一个CTRL+Z之前的所有字符和一个不可见字符，最后一行单独CTRL+Z输入终止

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例4：cin.get(字符变量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    cin.get(ch);
    cout << ch << int(ch) << endl;
    cin.get(ch);
    cout << ch << int(ch) << endl;

    return 0;
}
```

输入一个字符+回车，输出：该字符和该字符的ASCII码

（以输入a为例）　　空行

　　回车的ASCII码10

输入一串字符+回车，输出：第一个字符及其ASCII码

（以输入abc为例）　　第二个字符及其ASCII码

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例5：cin.get(字符变量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while(cin.get(ch))
        cout << ch;
    cout << endl;

    return 0;
}
```

输入：连续多个一串字符+回车，串中可含CTRL+Z（能否结束？否）
（以输入123，回车，输入a+CTRL+Z+b，回车，输入CTRL+Z为例）
输出：第一个CTRL+Z之前的所有字符和一个不可见字符
输入：连续多个一串字符+回车，最后一行单独CTRL+Z
（以输入123，回车，输入abc，回车，输入CTRL+Z为例）
输出：第一个CTRL+Z之前的所有字符和一个不可见字符，最后一行单独CTRL+Z输入终止
问：右侧为get无参例子，左右两个程序的输出是否相同？相同。

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while( (ch=cin.get())!=EOF )
        cout << ch;
    cout << endl;

    return 0;
}

//不需要写结果
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例6：cin.get(字符变量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while((cin.get(ch))!='\n')
        cout << ch;
    cout << endl;

    return 0;
}
```

编译出错，为什么？

cin.get(字符变量)返回cin(流对象自身)，无法进行逻辑运算比较，因此编译出错。

```
error C2678: 二进制“!=”: 没有找到接受“std::basic_istream<char,std::char_traits<char>>”类型的左操作数的运算符(或没有可接受的转换)
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例7：cin.get(字符变量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while((cin.get(ch))!=EOF)
        cout << ch;
    cout << endl;

    return 0;
}
```

编译出错，为什么？

cin.get(字符变量)返回cin(流对象自身)，无法进行逻辑运算比较，因此编译出错。

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    while( (ch=cin.get())!=EOF )
        cout << ch;
    cout << endl;

    return 0;
}
```

编译正确，为什么？

cin.get()从输入流中读取一个字符并返回该字符，并将其赋值给char型变量ch，进行!=运算的是cin.get()从输入流中读取并返回的字符，因此编译正确。

error C2678: 二进制“!=”: 没有找到接受“std::basic_istream<char,std::char_traits<char>>”类型的左操作数的运算符(或没有可接受的转换)

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例8：cin.get(字符数组，字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch[10];

    cin.get(ch, 10, '*');
    cout << ch << endl;

    return 0;
}
```

输入多于10个的字符串，输出：前9个字符

（以abcdefghijklmn输入为例）

输入小于10个的字符串，输出：光标闪烁，继续等待输入

（以abcdefg输入为例）

输入字符串，第9个及以前位置有*，输出：*之前的所有字符

（以abc*defghijklmn输入为例）

输入字符串，第10个及以后位置有*，输出：前9个字符

（以abcdefghijklm*n输入为例）

Microsoft Visual Studio 调试控制台

```
abcdefghijklmn
abcdefghi
```

D:\Learning\高级语言程序设计\高级语言程序设计\Debug\C++ Demo.exe

```
abcdefg
```

Microsoft Visual Studio 调试控制台

```
abc*defghijklmn
abc
```

Microsoft Visual Studio 调试控制台

```
abcdefghijklm*n
abcdefghi
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例9：cin.get(字符数组，字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch[10];

    cin.get(ch, 10);  //省略第3个参数
    cout << ch << endl;

    return 0;
}
```

输入多于10个的字符串，输出：前9个字符

（以abcdefghijklmn输入为例）

输入小于10个的字符串，输出：输入的所有字符

（以abcdefg输入为例）

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例10：cin.getline(字符数组，字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch[10];
    cin.getline(ch, 10, '*');
    cout << ch << endl;
    return 0;
}
```

输入多于10个的字符串，输出：前9个字符

（以abcdefghijklmn输入为例）

输入小于10个的字符串，输出：光标闪烁，继续等待输入

（以abcdefg输入为例）

输入字符串，第9个及以前位置有*，输出：*之前的所有字符

（以abc*defghijklmn输入为例）

输入字符串，第10个及以后位置有*，输出：前9个字符

（以abcdefghijklm*n输入为例）

是否与三个参数的cin.get相同？相同。

§.基础知识题 – 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例11：三个参数的cin.get与cin.getline的使用区别

```
#include <iostream>
using namespace std;

int main()
{
    char ch[20];
    cout << "enter a sentence:"; //不需要endl
    cin >> ch; //直接cin，空格结束
    cout << "The string with cin is:" << ch << '#' << endl;
    cin.getline(ch, 20, '/');
    cout << "The second part is:" << ch << '#' << endl;
    cin.getline(ch, 20); //缺省是回车结束
    cout << "The third part is:" << ch << '#' << endl;
}
```

运行结果：

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:I study C++./I am h#
```

Microsoft Visual Studio 调试控制台

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:I study C++./I am h#
```

```
#include <iostream>
using namespace std;

int main()
{
    char ch[20];
    cout << "enter a sentence:"; //不需要endl
    cin >> ch; //直接cin，空格结束
    cout << "The string with cin is:" << ch << '#' << endl;
    cin.getline(ch, 20, '/');
    cout << "The second part is:" << ch << '#' << endl;
    cin.getline(ch, 20, '/');
    cout << "The third part is:" << ch << '#' << endl;
}
```

运行结果：

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:I study C++.#
```

Microsoft Visual Studio 调试控制台

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:I study C++.#
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例12：三个参数的cin.get与cin.getline的使用区别

```
#include <iostream>
using namespace std;

int main()
{
    char ch[20];
    cout << "enter a sentence:"; //不需要endl
    cin >> ch;            //直接cin，空格结束
    cout << "The string with cin is:" << ch << '#' << endl;
    cin.get(ch, 20, '/');
    cout << "The second part is:" << ch << '#' << endl;
    cin.get(ch, 20, '/');
    cout << "The third part is:" << ch << '#' << endl;
}
```

和上页的差别：两句蓝色语句从getline变为get，则结果：

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:#
```

getline：遇见终止字符，在缓冲区内清除终止字符

get　　：遇见终止字符，将终止字符保留在缓冲区中

Microsoft Visual Studio 调试控制台

```
enter a sentence:I like C++./I study C++./I am happy.
The string with cin is:I#
The second part is: like C++.#
The third part is:#
```

§.基础知识题 - 与cin有关的成员函数的基本使用

1.用于字符输入的流成员函数

例13：三个参数的cin.get与cin.getline的使用区别

```
#include <iostream>
using namespace std;
int main()
{   char ch1[10], ch2[10];
    cin.get(ch1, 10, '*');
    cout << ch1 << endl;
    cin.get(ch2, 10, '*');
    cout << ch2 << endl;
    return 0;
}
```

输入一串大于20个字符的字符串，输出：
前九个字符及之后九个字符
（以输入abcdefghijklmnopqrstuvwxyz为例）
输入一串字符串，每9个以内含*，输出：
第一个*之前的所有字符
（以输入abcd*efghijkl*mnopqrst*uvwx为例）
输入一串小于9的字符串，加回车，输出：
光标闪烁，继续等待输入
（以输入abcdefg为例）

```
abcdefghijklmnopqrstuvwxyz
abcdefghi
jklmnopqr
```

```
abcd*efghijkl*mnopqrst*uvwx
abcd
```

```
abcdefg
```

```
#include <iostream>
using namespace std;
int main()
{   char ch1[10], ch2[10];
    cin.getline(ch1, 10, '*');
    cout << ch1 << endl;
    cin.getline(ch2, 10, '*');
    cout << ch2 << endl;
    return 0;
}
```

输入一串大于20个字符的字符串，输出：
前九个字符
（以输入abcdefghijklmnopqrstuvwxyz为例）
输入一串字符串，每9个以内含*，输出：
第一个*之前的字符及两个*之间的前九个字符
（以输入abcd*efghijkl*mnopqrst*uvwx为例）
输入一串小于9的字符串，加回车，输出：
光标闪烁，继续等待输入
（以输入abcdefg为例）

```
abcdefghijklmnopqrstuvwxyz
abcdefghi
```

```
abcd*efghijkl*mnopqrst*uvwx
abcd
efghijkl
```

```
abcdefg
```

- 输入满：get满后 可继续输入
  getline满后 停止输入
- 遇中止字符：get遇中止字符，下一个 仍为中止字符
  getline遇中止字符，下一个 跳过中止字符
- 未满遇回车：get把回车当一个普通字符读入至满，下一个 继续读入回车之后字符
  getline把回车当一个普通字符读入至满，下一个 继续读入回车之后字符

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例14：cin.eof()

```
#include <iostream>
using namespace std;

int main()  //P.430 例13.5
{
    char c;

    while (!cin.eof())
       if ((c=cin.get())!=' ')
          cout.put(c);

    return 0;
}
```

输入：连续多个字符串(含空格及CTRL+Z)+回车，最后一行单独CTRL+Z

（以输入123空格456，回车，输入abc+CTRL+Z+def，回车，输入CTRL+Z为例）

输出：第一个CTRL+Z之前的所有字符和一个不可见字符

最后一行单独CTRL+Z输入终止

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例14：cin.peek()

```
#include <iostream>
using namespace std;

int main()
{
    char ch;

    ch = cin.peek();
    cout << ch << int(ch) << endl;
    ch = cin.get();
    cout << ch << int(ch) << endl;

    return 0;
}
```

输入：ab　　输出为：

```
Microsoft Visual Studio 调试控制台
ab
a97
a97
```

输入：CTRL+Z　输出为：

```
Microsoft Visual Studio 调试控制台
^Z
 -1
 -1
```

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例15：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();    //get()一次

    cin.putback('H'); //putback()一次

    while((ch=cin.get())!='\n')
        cout.put(ch);

    return 0;
}
```

输入：abc　　输出：

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例16：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();    //get()两次
    ch = cin.get();
    cin.putback('H'); //putback()两次
    cin.putback('i');
    while((ch=cin.get())!='\n')
        cout.put(ch);

    return 0;
}
```

输入：abc　　输出：

§.基础知识题 – 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例17：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();      //get()一次
    cin.putback('H');    //putback()两次
    cin.putback('i');
    while((ch=cin.get())!='\n')
        cout.put(ch);
    return 0;
}
```

输入：abc

输出：VS　：

Dev ：

光标一直在动，什么意思？

持续输出不可见字符。

上两页的正确情况，本页的错误情况，综合起来，putback使用时要注意什么问题？

putback使用时要注意插入的字符数不要超过原输入流长度。

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例18：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();
    cin.putback('H');
    cin.putback('i');
    while((ch=cin.get())!='\n')
        cout << int(ch) << ' '; //输出换为int
    return 0;
}
```

输入：abc

输出：VS ：

Microsoft Visual Studio 调试控制台

```
abc
105 72 98 99
```

Dev ：

选择 C:\Users\lenovo\Desktop\demo.exe

```
abc
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例19：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();
    cin.putback('H');
    cin.putback('i');
    while((ch=cin.get())!=EOF)   //判断条件换为!=EOF
        cout.put(ch);
    return 0;
}
```

输入：abc

输出：VS ：

Dev ：

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例20：cin.putback(字符变量/字符常量)

```
#include <iostream>
using namespace std;

int main()
{   char c[20];
    int ch;
    cout << "please enter a sentense:" << endl;
    cin.getline(c, 15, '/');
    cout << "The first part is:" << c << endl;
    ch = cin.peek();
    cout << "The next char(ASCII):" << ch <<endl;
    cin.putback(c[0]);
    cin.getline(c, 15, '/');
    cout << "The second part is:" << c << endl;
    return 0;
}
```

运行结果(红色为输入)：

```
please enter a sentense:I am a boy./ am a student.
The first part is:I am a boy.
The next char(ASCII):32
The second part is:I am a student
```

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例21：cin.ignore(字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();
    cout << ch;
    cin.ignore(5, 'A');
    ch = cin.get();
    cout << ch;

    return 0;
}
```

输入：abcdefghijk　　输出：

```
Microsoft Visual Studio 调试控制台
abcdefghijk
ag
```

输入：abcdAfghijk　　输出：

```
Microsoft Visual Studio 调试控制台
abcdAfghijk
af
```

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例22：cin.ignore(字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch;
    ch = cin.get();
    cout << ch;
    cin.ignore();    //缺省1个字符，中止字符为EOF
    ch = cin.get();
    cout << ch;

    return 0;
}
```

输入：abcdefghijk　　输出：

输入：abcdAfghijk　　输出：

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例23：cin.ignore(字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch[20];
    cin.get(ch, 20, '/');  //指针停留在'/'处
    cout << "The first part is:" << ch << endl;

    cin.get(ch, 20, '/');  //从'/'处取，为空
    cout << "The second part is:" << ch << endl;
    return 0;
}
```

输入：I like C++./I study C++./I am happy.

输出：

§.基础知识题 - 与cin有关的成员函数的基本使用

2.与字符输入有关的其它成员函数

例24：cin.ignore(字符个数n，中止字符)

```
#include <iostream>
using namespace std;

int main()
{
    char ch[20];
    cin.get(ch, 20, '/');   //指针停留在'/'处
    cout << "The first part is:" << ch << endl;
    cin.ignore();           //跳过'/'
    cin.get(ch, 20, '/');   //从'/'后取，非空
    cout << "The second part is:" << ch << endl;
    return 0;
}
```

输入：I like C++./I study C++./I am happy.

输出：

Microsoft Visual Studio 调试控制台

```
I like C++./I study C++./I am happy.
The first part is:I like C++.
The second part is:I study C++.
```

§.基础知识题 - 与cin有关的成员函数的基本使用

3.输入错误处理

例25：C++方式cin输入错误处理

```
int x;
while (1) { //输入错误
    cout << "请输入x的值[0-100] : ";
    cin >> x;
    if (cin.fail()) {
        cin.clear();
        /* 方式1：清除缓冲区中字符，到'\n'为止，最大65536(可改) */
        cin.ignore(65536, '\n');
        /* 方式2：#include <limits>: INT_MAX */
        cin.ignore(INT_MAX, '\n');
        /* 方式3：numeric_limits<streamsize>::max(): 返回流缓冲区的最大容量 */
        /*        VS ：64位整数时直接可用，当 #include <Windows.h> 时不可用 */
        /*        Dev：32位整数且 #include <limits> 时可用                  */
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        continue;
    }
    else if (x >= 0 && x <= 100) { //输入正确且满足条件
        break;
    }
    else { //输入正确但不满足条件
        continue;
    }
}
```

§.基础知识题 - 与cin有关的成员函数的基本使用

3.输入错误处理

例26：C方式scanf输入错误处理

```
int x;
while (1) {
    printf("请输入x的值[0-100] : ");
    if (scanf("%d", &x) != 1) { //输入错误
        while (getchar() != '\n')
            continue;
        continue;
    }
    else if (x >= 0 && x <= 100) { //输入正确且满足条件
        break;
    }
    else { //输入正确但不满足条件
        continue;
    }
}
```

§.基础知识题 - 与cout有关的成员函数的基本使用

1.用于字符输出的流成员函数

例1：cout.put()

```
#include <iostream>
using namespace std;

int main()
{
    char str[] = "Hello";
    int i;

    for (i = 0; i < 5; i++)
        cout.put(str[i]);
    cout.put('\n');
    cout.put('H').put('e').put('l').put('l').put('o').put(0x0A);

    return 0;
}
```

运行结果：

§.基础知识题 – 与cout有关的成员函数的基本使用

1.用于字符输出的流成员函数

例2：cout.write()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";
    cout.write(s1, 5);
    cout.put('\n');
    cout.write(s1, 10);
    cout.put('\n');

    char s2[] = { 'H', 'e', 'l', 'l', 'o' };
    cout.write(s2, 5);
    cout.put('\n');
    cout.write(s2, 10);
    cout.put('\n');

    return 0;
}
```

运行结果：

当write的参数是字符串(s1)，且write要写的长度超过字符串长度时的表现：

cout.write()成员函数不会判断字符串是否结尾，指定显示几个字符就显示几个，即使数组越界，当write要写的长度超过字符串长度时，则继续输出后续内存中未初始化的随机值。

当write的参数非字符串(s2)，且write要写的长度超过字符串长度时的表现：

cout.write()成员函数不会判断字符串是否结尾，指定显示几个字符就显示几个，即使数组越界，当write要写的长度超过字符串长度时，则继续输出后续内存中未初始化的随机值。

结论：用write向标准输出设备输出指定个数的字符时，输出缓冲区<u>不要求</u>是字符串

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例3：cout.width()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";

    cout << "01234567890123456789" << endl;
    cout.width(10);
    cout << s1 << '*' << endl;
    cout << s1 << '#' << endl;

    return 0;
}
```

运行结果：

结论：

1、cout.width(10) 等价于 cout << setw(10)

2、cout.width()设置后 仅1次 有效

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例4：cout.width()与cout.fill()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";

    cout << "01234567890123456789" << endl;
    cout.width(10);
    cout.fill('$');
    cout << s1 << '*' << endl;
    cout.width(15);
    cout << s1 << '#' << endl;
    cout.width(12);
    cout.fill(' ');
    cout << s1 << '*' << endl;

    return 0;
}
```

运行结果：

结论：

1、cout.fill() 等价于 cout << setfill()

2、cout.fill()设置后 始终 有效

3、默认的cout.fill()设置是哪个字符？
ASCII码为32的空格字符(即' ')

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例5：cout.width()与cout.setf()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";

    cout << "01234567890123456789" << endl;
    cout.width(10);
    cout.setf(ios::left);
    cout << s1 << '*' << endl;
    cout.width(15);
    cout << s1 << '#' << endl;

    return 0;
}
```

运行结果：

结论：

1、cout.setf(ios::left) 等价于
cout << setiosflags(ios::left)

2、cout.setf()设置后<u>始终</u>有效

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例6：cout.width()与cout.setf()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";

    cout << "01234567890123456789" << endl;
    cout.width(10);
    cout.setf(ios::left);
    cout << s1 << '*' << endl;
    cout.setf(ios::right);
    cout.width(15);
    cout << s1 << '#' << endl;
    cout.width(10);
    cout.setf(ios::left);
    cout << s1 << '*' << endl;
    return 0;
}
```

运行结果：

结论：

1、cout.setf(ios::left) 等价于
cout << setiosflags(ios::left)

2、cout.setf(ios::right) 等价于
cout << setiosflags(ios::right)

3、cout.setf()设置后 始终 有效

4、不设置默认是 右 对齐

5、left后设置right是 有效 的

6、right后设置left是 无效 的

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例7：cout.width()与cout.setf()、cont.unsetf()

```
#include <iostream>
using namespace std;

int main()
{
    char s1[] = "Hello";

    cout << "01234567890123456789" << endl;
    cout.width(10);
    cout.setf(ios::left);
    cout << s1 << '*' << endl;
    cout.setf(ios::right);
    cout.width(15);
    cout << s1 << '#' << endl;
    cout.width(10);
    cout.unsetf(ios::right);//使用cout.函数名
    cout.setf(ios::left);
    cout << s1 << '*' << endl;
    return 0;
}
```

将程序补充完整，得到期望的运行结果：

所用的 cout.unsetf(ios::right) 等价于 cout << resetiosflags(ios::right)

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例8：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;

    cout << d << '*' << endl;
    cout.precision(10);
    cout << d << '*' << endl;
    cout.precision(3);
    cout << d << '*' << endl;
    cout.precision(5);
    cout << d << '*' << endl;
    cout.precision(20);
    cout << d << '*' << endl;
    return 0;
}
```

运行结果：

结论：

1、不做任何设置的情况下，浮点数默认为 小数 方式；不设precision的输出宽度默认为 6

2、默认情况下，precision设定的宽度是 全部数据

3、宽度 不包含 小数点

4、如果宽度超过有效位数，则 可以 显示，超出有效位数 不可信

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例9：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::fixed);
    cout << d << '*' << endl;
    cout.precision(10);
    cout << d << '*' << endl;
    cout.precision(3);
    cout << d << '*' << endl;
    cout.precision(5);
    cout << d << '*' << endl;
    cout.precision(20);
    cout << d << '*' << endl;
    return 0;
}
```

运行结果：

结论：

1、加ios::fixed后，precision默认的宽度为 6 ，设定的宽度是 小数部分

2、宽度 不包含 小数点

3、如果宽度超过有效位数，则 可以 显示，超出有效位数 不可信

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例10：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::scientific);
    cout << d << '*' << endl;
    cout.precision(10);
    cout << d << '*' << endl;
    cout.precision(3);
    cout << d << '*' << endl;
    cout.precision(5);
    cout << d << '*' << endl;
    cout.precision(20);
    cout << d << '*' << endl;
    return 0;
}
```

运行结果：

结论：

1、加ios::scientific后，precision默认的宽度为 6 ，设定的宽度是 小数部分

2、宽度 不包含 小数点

3、如果宽度超过有效位数，则 可以 显示，超出有效位数 不可信

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例11：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::fixed);
    cout.precision(10);
    cout << d << '*' << endl;

    cout.setf(ios::scientific);
    cout.precision(10);
    cout << d << '*' << endl;

    return 0;
}
```

运行结果：

结论：

先设ios::fixed后，再设ios::scientific，则输出显示 错误

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例12：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::scientific);
    cout.precision(10);
    cout << d << '*' << endl;

    cout.setf(ios::fixed);
    cout.precision(10);
    cout << d << '*' << endl;

    return 0;
}
```

运行结果：

结论：

先设ios::scientific后，再设ios::fixed，则输出显示 错误

§.基础知识题 - 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例13：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::fixed);
    cout.precision(10);
    cout << d << '*' << endl;
    cout.unsetf(ios::fixed);//使用cout.函数名
    cout.setf(ios::scientific);
    cout.precision(10);
    cout << d << '*' << endl;

    return 0;
}
```

将程序补充完整，得到期望的运行结果：

```
123.4567891235*
1.2345678912e+02*
```

§.基础知识题 – 与cout有关的成员函数的基本使用

2.用于字符输出控制的流成员函数

例14：cout.precision()

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double d = 123.456789123456;
    cout.setf(ios::scientific);
    cout.precision(10);
    cout << d << '*' << endl;
    cout.unsetf(ios::scientific);//使用cout.函数名
    cout.setf(ios::fixed);
    cout.precision(10);
    cout << d << '*' << endl;

    return 0;
}
```

将程序补充完整，得到期望的运行结果：

§.基础知识题 - sscanf与sprintf函数的基本使用

1.将格式化输出的内容放入字符串中

例1：

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

int main()
{
    char str[80];
    int k=123, ret;
    double pi=3.1415925;

    ret = sprintf(str, "k=%-4d*pi=%.2f#", k, pi);
    printf("ret : %d\n", ret);
    printf("str : %s\n", str);

    return 0;
}
```

输出结果：

§.基础知识题 - sscanf与sprintf函数的基本使用

1.将格式化输出的内容放入字符串中

例2：

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

int main()
{
    char str[80];
    int k=123, ret;
    double pi=3.1415925;

    ret = sprintf(str, "k=%6dpi=%10.2f", k, pi);
    printf("ret : %d\n", ret);
    printf("str : %s\n", str);

    return 0;
}
```

输出结果：

结合例1和例2，sprintf的返回值是：输出字符的个数

§.基础知识题 - sscanf与sprintf函数的基本使用

1.将格式化输出的内容放入字符串中

例3：

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

int main()
{
    char str[15];
    int k=123, ret;
    double pi=3.1415925;

    ret = sprintf(str, "k=%-4d*pi=%.2f#", k, pi);
    printf("ret : %d\n", ret);
    printf("str : %s\n", str);

    return 0;
}
```

输出结果：VS：

Microsoft Visual Studio 调试控制台

```
ret : 15
str : k=123 *pi=3.14#
```

Dev：

C:\Users\lenovo\Desktop\demo.exe

```
ret : 15
str : k=123 *pi=3.14#
```

VS编译弹窗报错

结合例1/2/3，sprintf使用时对字符数组的要求是：

字符数组要有足够空间容纳输出的数据(否则越界错)

§.基础知识题 - sscanf与sprintf函数的基本使用

2.从字符串中进行格式化输入

例4：

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

int main()
{
    char str[80] = "Hello 123 11.2", s[10];
    int i, ret;
    double d;

    ret = sscanf(str, "%s %d %lf", s, &i, &d);
    printf("ret : %d\n", ret);
    printf("s=%s i=%d d=%f\n", s, i, d);

    return 0;
}
```

输出结果：

§.基础知识题 - sscanf与sprintf函数的基本使用

2.从字符串中进行格式化输入

例5：

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

int main()
{
    char str[80] = "123Hello";
    int i, j, ret;

    ret = sscanf(str, "%d%d", &i, &j);
    printf("ret : %d\n", ret);
    printf("i=%d j=%d\n", i, j);

    return 0;
}
```

输出结果：

结合4例和例5，sscanf的返回值是：正确读入的输入数据的个数

§.基础知识题 - sscanf与sprintf函数的基本使用

2.从字符串中进行格式化输入

例6：

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

int main()
{
    char str[80] = "123 456";
    int i, j, ret;

    ret = sscanf(str, "%d%d", &i, &j);
    printf("ret : %d\n", ret);
    printf("i=%d j=%d\n", i, j);

    ret = sscanf(str, "%d%d", &j, &i); //顺序反
    printf("ret : %d\n", ret);
    printf("i=%d j=%d\n", i, j);

    return 0;
}
```

输出结果：

本例说明，str中的内容<u>不可以</u>被重复读取

§.基础知识题 - sscanf与sprintf函数的基本使用

3.综合应用

例7：

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

int main()
{
    char str[80] = "123 456";
    int i, j, ret;

    ret = sscanf(str, "%d%d", &i, &j);
    printf("ret : %d\n", ret);
    printf("str=%s\ni=%d j=%d\n", str, i, j);

    ret = sprintf(str, "i=%d j=%d", i, j);
    printf("ret : %d\n", ret);
    printf("str=\"%s\"\n", str);

    return 0;
}
```

输出结果：

本例说明，str中的内容<u>可以</u>被替换

§.基础知识题 - sscanf与sprintf函数的基本使用

3.综合应用

例8：

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

int main()
{
    int x, w;
    printf("请输入[1..99999]间的整数及显示宽度[6..10]\n");
    scanf("%d %d", &x, &w); //不考虑输入错误
    printf("01234567890123456789\n"); //标尺

    char fmt[16];
    sprintf(fmt, "%%%dd*\n", w);
    printf(fmt, x);

    return 0;
}
```

1、输入3 6，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
3 6
01234567890123456789
     3*
```

2、输入123 6，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
123 6
01234567890123456789
   123*
```

3、输入12345 6，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
12345 6
01234567890123456789
 12345*
```

4、输入3 9，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
3 9
01234567890123456789
        3*
```

5、输入123 9，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
123 9
01234567890123456789
      123*
```

6、输入12345 9，输出：

```
Microsoft Visual Studio 调试控制台
请输入[1..99999]间的整数及显示宽度[6..10]
12345 9
01234567890123456789
    12345*
```

§.基础知识题 - sscanf与sprintf函数的基本使用

3.综合应用

例9：键盘输入一个长度[3..12]间字符串，再输入显示宽度[长度+1..20]，左对齐输出这个字符串（最后加*分辨空格）

注：输入宽度小于等于串长则置为串长+1，不考虑其它输入错误

```
//给出相应的代码
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>

int main()
{
    char str[20] = { 0 };
    int w;

    printf("请输入长度[3..12]间的字符串及显示宽度[len+1..20]\n");
    scanf("%s %d", str, &w);
    printf("01234567890123456789\n");
    if (w <= (int)strlen(str))
        w = strlen(str) + 1;
    char fmt[8];

    sprintf(fmt, "%%-%ds*", w);
    printf(fmt, str);

    return 0;
}
```

1、输入abc 12，输出：

2、输入abc 2，输出：

3、输入abcdefg 3，输出：

4、输入abcdefg 13，输出：

§.基础知识题 - sscanf与sprintf函数的基本使用

3.综合应用

例10：键盘输入一个double型数据，再输入总显示宽度及小数点后的位数，右对齐输出这个字符串（最后加*分辨空格）

注：不考虑其它输入错误

```
//给出相应的代码
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int main()
{
    double x;
    int  w, h;

    printf("请输入double型数据及显示总宽度、小数点后位数\n");
    scanf("%lf%d%d", &x, &w, &h);
    printf("01234567890123456789\n");
    char fmt[20];

    sprintf(fmt, "%%%d.%dlf*", w, h);
    printf(fmt, x);

    return 0;
}
```

1、输入12.34 9 5，输出：

```
Microsoft Visual Studio 调试控制台
请输入double型数据及显示总宽度、小数点后位数
12.34 9 5
01234567890123456789
 12.34000*
```

2、输入123.456789 12 2，输出：

```
Microsoft Visual Studio 调试控制台
请输入double型数据及显示总宽度、小数点后位数
123.456789 12 2
01234567890123456789
       123.46*
```

3、输入12345678.9 5 2，输出：

```
Microsoft Visual Studio 调试控制台
请输入double型数据及显示总宽度、小数点后位数
12345678.9 5 2
01234567890123456789
12345678.90*
```

4、输入12345678.9 5 0，输出：

```
Microsoft Visual Studio 调试控制台
请输入double型数据及显示总宽度、小数点后位数
12345678.9 5 0
01234567890123456789
12345679*
```