

# 第二章 C++ 简单程序设计

2.1 C++语言概述

2.2 基本数据类型和表达式

2.3 数据的输入与输出

2.4 算法的基本控制结构

2.5 枚举类型

### 2.1.1 C++ 的产生

- C++是从C语言发展演变而来的，首先是一个更好的C
- 引入了类的机制，最初的C++被称为“带类的C”
- 1983年正式取名为C++
- 从1989年开始C++语言的标准化工作
- 于1994年制定了ANSI C++标准草案
- 于1998年11月被国际标准化组织（ISO）批准为国际标准，成为目前的C++

### 2.1.2 C++ 的特点

- 兼容C
  - 它保持了C的简洁、高效和接近汇编语言等特点
  - 对C的类型系统进行了改革和扩充
  - C++也支持面向过程的程序设计，不是一个纯正的面向对象的语言
- 支持面向对象的方法

## 2.1.3 C++ 程序实例—例2-1

```
//2_1.cpp
#include <iostream>
using namespace std;
int main() {
    cout << "Hello!" << endl;
    cout << "Welcome to this class!" << endl;
    return 0;
}
```

运行结果:

Hello!

Welcome to this class!

## 2.1.4 C++ 字符集

- 大小写的英文字母：A~Z, a~z
- 数字字符：0~9
- 特殊字符：

!	#	%	^	&	*	_	+
=	-	~	<	>	/	\	'
“	;	.	,	:	?	(	)
[	]	{	}				

## 2.1.5 词法记号

- **关键字** C++预定义的单词
- **标识符** 程序员声明的单词，它命名程序正文中的一些实体
- **文字** 在程序中直接使用符号表示的数据
- **操作符** 用于实现各种运算的符号
- **分隔符** `() {} , : ;`  
用于分隔各个词法记号或程序正文
- **空白符** 空格、制表符（TAB键产生的字符）、垂直制表符、换行符、回车符和注释的总称

## 标识符的构成规则

- 以大写字母、小写字母或下划线(\_)开始。
- 可以由以大写字母、小写字母、下划线(\_)或数字0~9组成。
- 字母是大小写敏感的，即：大写字母和小写字母代表不同的标识符。

### 2.2.1 基本数据类型

- C++能够处理的基本数据类型
  - 整数类型
  - 浮点数类型
  - 字符类型
  - 布尔类型
- 程序中的数据
  - 常量
    - 在源程序中直接写明的数据，其值在整个程序运行期间不可改变，这样的数据称为常量。
  - 变量
    - 在程序运行过程中允许改变的数据，称为变量。



## 2.2.1 基本数据类型

- 整数类型
  - 基本的整数类型
    - `int`
  - 按符号分
    - 符号的 (`signed`) 和无符号的 (`unsigned`)
  - 按照数据范围分
    - 短整型 (`short`) 和长整型 (`long`)，分别至少16位和32位；
    - 超越`long`(C99增加的2种类型)——长长整型(至少64位)：
      - ✓ `long long`和
      - ✓ `unsigned long long`。
  - 字符类型
    - `char`型，8位，实质上存储的也是整数（详见字符类型）

### 2.2.1 基本数据类型

- 浮点数类型
  - 单精度
    - float
  - 双精度
    - double
  - 扩展精度
    - long double

### 2.2.1 基本数据类型

- 字符类型
  - `char`类型
  - 容纳单个字符的编码
- 字符串类型（详见第6章）
  - C风格的字符串
    - 采用字符数组
  - C++风格的字符串
    - 采用标准C++类库中的String类
- 布尔类型
  - `bool`类型，只有两个值：`true`（真）、`false`（假）
  - 常用来表示关系比较、相等比较或逻辑运算的结果

## 2.2.1 基本数据类型

类型名	长度(byte)	取值范围
bool	1	false, true
char (signed char)	1	-128~127
unsigned char	1	0~255
short (signed short)	2	-32768~32767
unsigned short	2	0~65535
int (signed int)	4	-2147483648~2147483647
unsigned int	4	0~4294967295
long (signed long)	4	-2147483648~2147483647
unsigned long	4	0~4294967295
long long (signed long long)	8	-9223372036854775808~9223372036854775807
unsigned long long	8	0~18446744073709551615
float	4	$\pm 3.4\text{E}\pm 38$
double	8	$\pm 1.7\text{E}\pm 308$
long double	8	$\pm 1.7\text{E}\pm 308$

### 2.2.2 常量

- 所谓常量是指在程序运行的整个过程中其值始终不可改变的量，也就是直接使用符号（文字）表示的值。例如：12，3.5，'A'都是常量。

## 整数常量

- 以文字形式出现的整数，包括正整数、负整数和零。
- 十进制整数
  - [±]若干个0~9的数字，但数字部分不能以0开头，正数前边的正号可以省略。
- 八进制整数
  - [±]数字0若干个0~7的数字，注意数字部分要以数字0开头。
- 十六进制整数
  - [±]0x若干个0~9的数字及A~F的字母(大小写均可),注意数字部分要以数字0和x(或X)开头,不能以字母o(或O)和x(或X)开头。
- 后缀
  - L(或l)表示长整型；U(或u)表示无符号型。也可同时后缀L和U(大小写无关)。

## 浮点数常量

- 以文字形式出现的实数。
- 一般形式：
  - 例如，12.5，-12.5等。
- 指数形式：
  - 例如，0.345E+2，-34.4E-3
  - 字母E可以大写或小写。
  - 整数部分和小数部分可以省略其一
- 实数常量默认为double型，如果实数常量带后缀F(或f)可以使其成为float型，例如：12.3f。

## 字符常量

- 字符常量

- 单引号括起来的一个字符，  
如：'a', 'D', '?', '\$'
- C++转义字符列表（用于在程序中表示不可显示字符）

字符常量形式	ASCII码（十六进制）	含义
\a	07	响铃
\n	0A	换行
\t	09	水平制表符
\v	0B	垂直制表符
\b	08	退格
\r	0D	回车
\f	0C	换页
\\	5C	字符“\”
\"	22	双引号
\'	27	单引号



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	<b>NUL</b> (null)	32	20	040	&#32;	<b>Space</b>	64	40	100	&#64;	<b>@</b>	96	60	140	&#96;	<b>`</b>
1	1	001	<b>SOH</b> (start of heading)	33	21	041	&#33;	<b>!</b>	65	41	101	&#65;	<b>A</b>	97	61	141	&#97;	<b>a</b>
2	2	002	<b>STX</b> (start of text)	34	22	042	&#34;	<b>"</b>	66	42	102	&#66;	<b>B</b>	98	62	142	&#98;	<b>b</b>
3	3	003	<b>ETX</b> (end of text)	35	23	043	&#35;	<b>#</b>	67	43	103	&#67;	<b>C</b>	99	63	143	&#99;	<b>c</b>
4	4	004	<b>EOT</b> (end of transmission)	36	24	044	&#36;	<b>\$</b>	68	44	104	&#68;	<b>D</b>	100	64	144	&#100;	<b>d</b>
5	5	005	<b>ENQ</b> (enquiry)	37	25	045	&#37;	<b>%</b>	69	45	105	&#69;	<b>E</b>	101	65	145	&#101;	<b>e</b>
6	6	006	<b>ACK</b> (acknowledge)	38	26	046	&#38;	<b>&amp;</b>	70	46	106	&#70;	<b>F</b>	102	66	146	&#102;	<b>f</b>
7	7	007	<b>BEL</b> (bell)	39	27	047	&#39;	<b>'</b>	71	47	107	&#71;	<b>G</b>	103	67	147	&#103;	<b>g</b>
8	8	010	<b>BS</b> (backspace)	40	28	050	&#40;	<b>(</b>	72	48	110	&#72;	<b>H</b>	104	68	150	&#104;	<b>h</b>
9	9	011	<b>TAB</b> (horizontal tab)	41	29	051	&#41;	<b>)</b>	73	49	111	&#73;	<b>I</b>	105	69	151	&#105;	<b>i</b>
10	A	012	<b>LF</b> (NL line feed, new line)	42	2A	052	&#42;	<b>*</b>	74	4A	112	&#74;	<b>J</b>	106	6A	152	&#106;	<b>j</b>
11	B	013	<b>VT</b> (vertical tab)	43	2B	053	&#43;	<b>+</b>	75	4B	113	&#75;	<b>K</b>	107	6B	153	&#107;	<b>k</b>
12	C	014	<b>FF</b> (NP form feed, new page)	44	2C	054	&#44;	<b>,</b>	76	4C	114	&#76;	<b>L</b>	108	6C	154	&#108;	<b>l</b>
13	D	015	<b>CR</b> (carriage return)	45	2D	055	&#45;	<b>-</b>	77	4D	115	&#77;	<b>M</b>	109	6D	155	&#109;	<b>m</b>
14	E	016	<b>SO</b> (shift out)	46	2E	056	&#46;	<b>.</b>	78	4E	116	&#78;	<b>N</b>	110	6E	156	&#110;	<b>n</b>
15	F	017	<b>SI</b> (shift in)	47	2F	057	&#47;	<b>/</b>	79	4F	117	&#79;	<b>O</b>	111	6F	157	&#111;	<b>o</b>
16	10	020	<b>DLE</b> (data link escape)	48	30	060	&#48;	<b>0</b>	80	50	120	&#80;	<b>P</b>	112	70	160	&#112;	<b>p</b>
17	11	021	<b>DC1</b> (device control 1)	49	31	061	&#49;	<b>1</b>	81	51	121	&#81;	<b>Q</b>	113	71	161	&#113;	<b>q</b>
18	12	022	<b>DC2</b> (device control 2)	50	32	062	&#50;	<b>2</b>	82	52	122	&#82;	<b>R</b>	114	72	162	&#114;	<b>r</b>
19	13	023	<b>DC3</b> (device control 3)	51	33	063	&#51;	<b>3</b>	83	53	123	&#83;	<b>S</b>	115	73	163	&#115;	<b>s</b>
20	14	024	<b>DC4</b> (device control 4)	52	34	064	&#52;	<b>4</b>	84	54	124	&#84;	<b>T</b>	116	74	164	&#116;	<b>t</b>
21	15	025	<b>NAK</b> (negative acknowledge)	53	35	065	&#53;	<b>5</b>	85	55	125	&#85;	<b>U</b>	117	75	165	&#117;	<b>u</b>
22	16	026	<b>SYN</b> (synchronous idle)	54	36	066	&#54;	<b>6</b>	86	56	126	&#86;	<b>V</b>	118	76	166	&#118;	<b>v</b>
23	17	027	<b>ETB</b> (end of trans. block)	55	37	067	&#55;	<b>7</b>	87	57	127	&#87;	<b>W</b>	119	77	167	&#119;	<b>w</b>
24	18	030	<b>CAN</b> (cancel)	56	38	070	&#56;	<b>8</b>	88	58	130	&#88;	<b>X</b>	120	78	170	&#120;	<b>x</b>
25	19	031	<b>EM</b> (end of medium)	57	39	071	&#57;	<b>9</b>	89	59	131	&#89;	<b>Y</b>	121	79	171	&#121;	<b>y</b>
26	1A	032	<b>SUB</b> (substitute)	58	3A	072	&#58;	<b>:</b>	90	5A	132	&#90;	<b>Z</b>	122	7A	172	&#122;	<b>z</b>
27	1B	033	<b>ESC</b> (escape)	59	3B	073	&#59;	<b>;</b>	91	5B	133	&#91;	<b>[</b>	123	7B	173	&#123;	<b>{</b>
28	1C	034	<b>FS</b> (file separator)	60	3C	074	&#60;	<b>&lt;</b>	92	5C	134	&#92;	<b>\</b>	124	7C	174	&#124;	<b> </b>
29	1D	035	<b>GS</b> (group separator)	61	3D	075	&#61;	<b>=</b>	93	5D	135	&#93;	<b>]</b>	125	7D	175	&#125;	<b>}</b>
30	1E	036	<b>RS</b> (record separator)	62	3E	076	&#62;	<b>&gt;</b>	94	5E	136	&#94;	<b>^</b>	126	7E	176	&#126;	<b>~</b>
31	1F	037	<b>US</b> (unit separator)	63	3F	077	&#63;	<b>?</b>	95	5F	137	&#95;	<b>_</b>	127	7F	177	&#127;	<b>DEL</b>

## Extended ASCII Codes

128	Ç	144	É	160	á	176	░	192	Ł	208	Ш	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	ł	209	̐	225	β	241	±
130	é	146	Æ	162	ó	178	▓	194	ṽ	210	̑	226	Γ	242	≥
131	â	147	ô	163	ú	179		195	ṽ	211	Ш	227	π	243	≤
132	ä	148	ö	164	ñ	180	└	196	—	212	↳	228	Σ	244	∫
133	à	149	ò	165	Ñ	181	┐	197	+	213	ℱ	229	σ	245	┐
134	å	150	û	166	²	182	┌	198	┐	214	ℙ	230	μ	246	÷
135	ç	151	ù	167	°	183	▯	199	┌	215	▯	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	▯	200	↳	216	≠	232	Φ	248	°
137	ë	153	Ö	169	┐	185	▯	201	ℙ	217	┐	233	⊙	249	.
138	è	154	Ü	170	┐	186	▯	202	Ш	218	┐	234	Ω	250	.
139	ï	155	©	171	½	187	▯	203	̐	219	■	235	δ	251	√
140	î	156	£	172	¼	188	▯	204	┐	220	■	236	∞	252	π
141	ì	157	¥	173	¡	189	▯	205	=	221	▯	237	φ	253	²
142	Ä	158	£	174	«	190	▯	206	▯	222	▯	238	ε	254	■
143	Å	159	ƒ	175	»	191	▯	207	▯	223	■	239	∩	255	

## 2.2.3 变量

- 变量的声明和定义
  - 声明语句形式如下：
    - 数据类型 变量名1, 变量名2, ..., 变量名n;
  - 在定义一个变量的同时，也可以给它赋以初值
    - 例如： `int a = 3;`
- 变量的存储类型
  - **auto**: 采用堆栈方式分配内存空间属于暂时性存储，其存储空间可以被若干变量多次覆盖使用。
  - **register**: 存放在通用寄存器中。
  - **extern**: 在所有函数和程序段中都可引用。
  - **static**: 在内存中是以固定地址存放的，在整个程序运行期间都有效。

## 2.2.4 符号常量

- 符号常量在声明时一定要赋初值，而在程序中间不能改变其值。
  - `const` 数据类型说明符 常量名=常量值;  
或:  
数据类型说明符 `const` 常量名=常量值;

例: `const float PI = 3.1415926;`

## 程序举例

```
#include <iostream>
using namespace std;
int main() {
    const int PRICE = 30;
    int num, total;
    double v, r, h;
    num = 10;
    total = num * PRICE;
    cout << total << endl;
    r = 2.5;
    h = 3.2;
    v = 3.14159 * r * r * h;
    cout << v << endl;
    return 0;
}
```

## 算术运算符与算术表达式

- 基本算术运算符

$+$   $-$   $*$   $/$  (若整数相除，结果取整)

$\%$  (取余，操作数为整数)

- 优先级与结合性

先乘除，后加减，同级自左至右

- $++$ ,  $--$  (自增、自减)

例:  $i++$ ;  $--j$ ;

## 赋值运算符和赋值表达式

### ——简单的赋值运算符“=”

- 举例

$n = n + 5$

- 表达式的类型

赋值运算符左边对象的类型

- 表达式的值

赋值运算符左边对象被赋值后的值



## 赋值运算符和赋值表达式

### ——复合的赋值运算符

- 有10种复合运算符:

$+=, -=, *=, /=, \%=,$

$<<=, >>=, \&=, ^=, |=$

- 例

$a += 3$  等价于  $a = a + 3$

$x *= y + 8$  等价于  $x = x * (y + 8)$



## 逗号运算和逗号表达式

- 格式

表达式1, 表达式2

- 求解顺序及结果

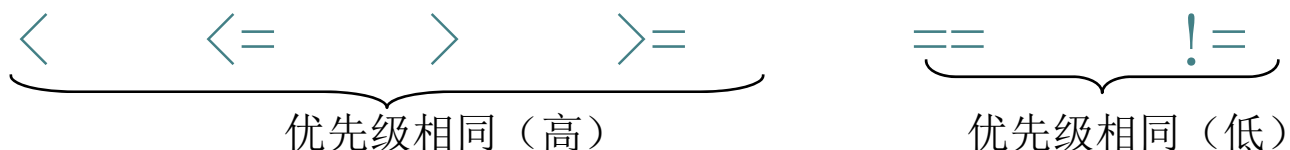
先求解1, 再求解2, 最终结果为表达式2的值

- 例

a = 3 \* 5 , a \* 4      最终结果为60

## 关系运算与关系表达式

- 关系运算是一种比较简单的一种逻辑运算，优先次序为：



- 关系表达式是一种最简单的逻辑表达式  
其结果类型为 `bool`，值只能为 `true` 或 `false`。
- 例如：`a > b`，`c <= a + b`，`x + y == 3`

## 逻辑运算与逻辑表达式

- 逻辑运算符

!(非)   &&(与)   ||(或)

优先次序： 高      →      低

- 逻辑表达式

例如：(a > b) && (x > y)

其结果类型为 `bool`，值只能为 `true` 或 `false`

## 逻辑运算与逻辑表达式（续）

- “&&”的“短路特性”

表达式1 && 表达式2

- 先求解表达式1
- 若表达式1的值为false，则最终结果为false，不再求解表达式2
- 若表达式1的结果为true，则求解表达式2，以表达式2的结果作为最终结果

- “||”也具有类似的特性

## 条件运算符与条件表达式

- 一般形式

表达式1? 表达式2: 表达式3

表达式1 必须是 bool 类型

- 执行顺序

- 先求解表达式1,

- 若表达式1的值为true, 则求解表达式2, 表达式2的值为最终结果

- 若表达式1的值为false, 则求解表达式3, 表达式3的值为最终结果

- 例: `x = a > b ? a : b;`

## 条件运算符与条件表达式（续）

- 注意：
  - 条件运算符优先级高于赋值运算符，低于逻辑运算符
  - 表达式2、3的类型可以不同，条件表达式的最终类型为 2 和 3 中较高的类型。

• 例：  $x = a > b ? a : b;$

①

②

## sizeof 运算符

- 语法形式  
`sizeof (类型名)`  
或 `sizeof 表达式`
- 结果值：  
“类型名”所指定的类型或“表达式”的结果类型所占的字节数。
- 例：  
`sizeof(short)`  
`sizeof x`

## 位运算——按位与 (&)

- 运算规则
  - 将两个运算量的每一个位进行逻辑与操作
- 举例：计算 3 & 5

```
3:      0 0 0 0 0 0 1 1
5: (&)  0 0 0 0 0 1 0 1
-----
3 & 5:  0 0 0 0 0 0 0 1
```
- 用途：
  - 将某一位置0，其他位不变。例如：  
将char型变量a的最低位置0: `a = a & 0xfe;`
  - 取指定位。  
例如：有 `char c; int a;`  
取出a的低字节，置于c中: `c = a & 0xff;`



## 位运算——按位或（|）

- 运算规则
  - 将两个运算量的每一个位进行逻辑或操作
- 举例：计算  $3 \mid 5$

3:		0	0	0	0	0	0	1	1
5:	( )	0	0	0	0	0	1	0	1
<hr/>									
3   5:		0	0	0	0	0	1	1	1

- 用途：
  - 将某些位置1，其他位不变。  
例如：将 **int** 型变量 **a** 的低字节置 **1**：  
 **$a = a \mid 0xff;$**

## 位运算——按位异或 ( ^ )

- 运算规则
  - 两个操作数进行异或：  
若对应位相同，则结果该位为 0，  
若对应位不同，则结果该位为 1，
- 举例：计算  $071 \wedge 052$

071:	0	0	1	1	1	0	0	1	
052:	(^)	0	0	1	0	1	0	1	0
<hr/>									
$071 \wedge 052$ :	0	0	0	1	0	0	1	1	

## 位运算——按位异或（ $\wedge$ ）（续）

- 用途：

- 使特定位翻转（与0异或保持原值，与1异或取反）

例如：要使 **01111010** 低四位翻转：

$$\begin{array}{r} 0\ 1\ 1\ 1\ 1\ 0\ 1\ 0 \\ (\wedge)\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\ \hline 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1 \end{array}$$

## 位运算——取反（ $\sim$ ）

单目运算符，对一个二进制数按位取反。


例： 025: 00000000000010101

$\sim$ 025: 11111111111101010

## 位运算——移位

- 左移运算 (<<)  
左移后，低位补0，高位舍弃。
- 右移运算 (>>)  
右移后，  
低位：舍弃  
高位：无符号数：补0  
有符号数：补“符号位”

## 运算符优先级：

( )	 <div>高</div>
++, --, sizeof	
*, /, %	
+, -	
==, !=	
位运算	
&&	
?:, 赋值运算 (结合性: 右→左)	
逗号运算 (结合性: 左→右)	
	低

## 混合运算时数据类型的转换

### ——隐含转换

- 一些二元运算符（算术运算符、关系运算符、逻辑运算符、位运算符和赋值运算符）要求两个操作数的类型一致。
- 在算术运算和关系运算中如果参与运算的操作数类型不一致，编译系统会自动对数据进行转换（即隐含转换），基本原则是将低类型数据转换为高类型数据。

char, short, int, unsigned, long, unsigned long, float, double

低 —————→ 高

## 混合运算时数据类型的转换 ——隐含转换（续）

- 当参与运算的操作数必须是bool型时，如果操作数是其它类型，编译系统会自动将**非0**数据转换为true，0转换为false。
- **位运算**的操作数必须是**整数**，当二元位运算的操作数是不同类型的整数时，也会自动进行类型转换，
- 赋值运算要求左值与右值的类型相同，若类型不同，编译系统会自动将右值转换为左值的类型。



## 混合运算时数据类型的转换 ——显式转换

- 语法形式（3种）：
  - 类型说明符(表达式)
  - (类型说明符)表达式
  - 类型转换操作符<类型说明符>(表达式)
    - 类型转换操作符可以是：  
`const_cast`、`dynamic_cast`、  
`reinterpret_cast`、`static_cast`
- 显式类型转换的作用是将表达式的结果类型转换为类型说明符所指定的类型。
- 例：`int(z)`, `(int)z`, `static_cast<int>(z)`  
三种完全等价
- 类型转换操作符在后续章节中会陆续介绍。

# 【 1 】 static\_cast

- 用法: `static_cast < type-id > (expression)`

允许执行任意的隐式转换和相反转换动作 (即使它是不允许隐式的)。该运算符把`expression`转换为`type-id`类型, 但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法:

- ①用于类层次结构中基类和子类之间指针或引用的转换。
  - 进行上行转换(把子类的指针或引用转换成基类表示)是安全的;
  - 进行下行转换(把基类指针或引用转换成子类表示)时, 由于没有动态类型检查, 所以是不安全的。
- ②用于基本数据类型之间的转换, 如把`int`转换成`char`, 把`int`转换成`enum`。这种转换的安全性也要开发人员来保证。
- ③把空指针转换成目标类型的空指针。
- ④把任何类型的表达式转换成`void`类型。
- 注意: `static_cast` 不能转换掉`expression`的`const`、`volatile`、或者`__unaligned`属性。

## 【 2 】dynamic\_cast

- 用法: `dynamic_cast < type-id > (expression)`

该运算符把expression转换成type-id类型的对象。Type-id必须是类的指针、类的引用或者void\*。

它只用于对象的指针和引用。当用于多态类型时,它允许任意的隐式类型转换以及相反过程。不过,与static\_cast不同,在后一种情况里(注:即隐式转换的相反过程),dynamic\_cast会检查操作是否有效。也就是说,它会检查转换是否会返回一个被请求的有效的完整对象。检测在运行时进行。如果被转换的指针不是一个被请求的有效完整的对象指针,返回值为NULL。

- 如果type-id是类指针类型,那么expression也必须是一个指针,如果type-id是一个引用,那么expression也必须是一个引用。
- dynamic\_cast主要用于类层次间的上行转换和下行转换,还可以用于类之间的交叉转换。
- 在类层次间进行上行转换时,dynamic\_cast和static\_cast的效果是一样的;
- 在进行下行转换时,dynamic\_cast具有类型检查的功能,比static\_cast更安全。

## 【 3 】 reinterpret\_cast

- 用法: `reinterpret_cast<type-id> (expression)`

它转换一个指针为其它类型的指针,也可将一个指针转换为整数类型。反之亦然。

`reinterpret_cast`运算符是用来处理无关类型之间的转换;它会产生一个新的值,这个值会有与原始参数(`expression`)有完全相同的比特位。按照`reinterpret`的字面意思“重新解释”,即对数据的比特位重新解释。

- [IBM的C++指南](#) 里明确告诉我们: `reinterpret_cast`可以,或者说应该在什么地方用来作为转换运算符:
  - 从指针类型到一个足够大的整数类型
  - 从整数类型或者枚举类型到指针类型
  - 从一个指向函数的指针到另一个不同类型的指向函数的指针
  - 从一个指向对象的指针到另一个不同类型的指向对象的指针
  - 从一个指向类函数成员的指针到另一个指向不同类型的函数成员的指针
  - 从一个指向类数据成员的指针到另一个指向不同类型的数据成员的指针
  - 总结来说: `reinterpret_cast`用在任意指针(或引用)类型之间的转换,以及指针与足够大的整数类型之间的转换,从整数类型(包括枚举类型)到指针类型,无视大小。
- 注意: `static_cast` 不能转换掉`expression`的`const`、`volatile`、或者`__unaligned`属性。

## 【 4 】 const\_cast

- 用法: `const_cast<type-id> (expression)`  
该运算符用来修改类型的`const`、`volatile`、`__unaligned`属性。  
除了`const`、`volatile`、`__unaligned`修饰之外，`type_id`和`expression`的类型是一样的。
  - 常量指针被转化成非常量指针,并且仍然指向原来的对象;
  - 常量引用被转换成非常量引用,并且仍然指向原来的对象;
  - 常量对象被转换成非常量对象。
- 这个转换类型操纵传递对象的`const`属性, 或者是设置或者是移除:

## 2.2.6 语句

- C++中的语句有如下种类
  - 空语句
    - 只有一个语句结束符 “;”
  - 声明语句
    - 例如：变量声明
  - 表达式语句
    - 在表达式末尾添加语句结束符便构成表达式语句
    - 例如：a=3;
  - 流程控制语句（详见2.4节）
    - 选择语句、循环语句、跳转语句
  - 标号语句
    - 在语句前附加标号，通常用来与跳转语句配合
- 复合语句
  - 用 “{}” 括起来的多条语句

### 2.3.1 I/O 流

- 在C++中，将数据从一个对象到另一个对象的流动抽象为“流”。流在使用前要被建立，使用后要被删除。
- 从流中获取数据的操作称为提取操作，向流中添加数据的操作称为插入操作。
- 数据的输入与输出是通过I/O流来实现的，**cin**和**cout**是预定义的流类对象。**cin**用来处理标准输入，即键盘输入。**cout**用来处理标准输出，即屏幕输出。

## 2.3.2 预定义的插入符和提取符

- “<<”是预定义的插入符，作用在流类对象cout上便可以实现最一般的屏幕输出。
  - `cout << 表达式 << 表达式...`
- 键盘输入是将提取符作用在流类对象cin上。
  - `cin >> 表达式 >> 表达式...`
- 在输入语句中，提取符可以连续写多个，每个后面跟一个表达式，该表达式通常是用于存放输入值的变量。例如：
  - `int a, b;`
  - `cin >> a >> b;`



## 2.3.3 简单的I/O格式控制

### 常用的I/O流类库操纵符

操纵符名	含 义
dec	数值数据采用十进制表示
hex	数值数据采用十六进制表示
oct	数值数据采用八进制表示
ws	提取空白符
endl	插入换行符，并刷新流
ends	插入空字符
setprecision(int)	设置浮点数的小数位数(包括小数点)
setw(int)	设置域宽

例: `cout << setw(5) << setprecision(3) << 3.1415;`

- 使用`setprecision(n)`可控制输出流显示浮点数的小数部分的数字个数(包括小数点)。C++默认的值`n=6`。

```
#include <iostream>
#include <iomanip> //格式控制
using namespace std;

int main()
{
    double PI= 3.141592653;
    cout<<"when default PI= "<<PI<<endl; //(1)
    for(int i=0;i<13;i++)
    {
        cout<<"when setprecision("<<i<<") PI= "<<setprecision(i) <<PI<<endl;
    }
}
```

- 程序运行的结果：
  - `/******`
  - when default PI= 3.14159
  - when setprecision(0) PI= 3
  - when setprecision(1) PI= 3
  - when setprecision(2) PI= 3.1
  - when setprecision(3) PI= 3.14
  - when setprecision(4) PI= 3.142
  - when setprecision(5) PI= 3.1416
  - when setprecision(6) PI= 3.14159
  - when setprecision(7) PI= 3.141593
  - when setprecision(8) PI= 3.1415927
  - when setprecision(9) PI= 3.14159265
  - when setprecision(10) PI= 3.141592653
  - when setprecision(11) PI= 3.141592653
  - when setprecision(12) PI= 3.141592653
- C++还提供了一种在输出中进行换行的旧式方式：C语言符号“\n”或‘\n’。

## 2.4.1 用if语句实现选择结构

```
//2_2.cpp
#include <iostream>
using namespace std;
int main() {
    int year;
    bool isLeapYear;

    cout << "Enter the year: ";
    cin >> year;
    isLeapYear = ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0));

    if (isLeapYear)
        cout << year << " is a leap year" << endl;
    else
        cout << year << " is not a leap year" << endl;

    return 0;
}
```

例2-2 输入一个年份，判断是否闰年

## If语句的语法形式

if (表达式) 语句

例: if (x > y) cout << x;

if (表达式) 语句1 else 语句2

例: if (x > y) cout << x;  
else cout << y;

if (表达式1) 语句1

else if (表达式2) 语句2

else if (表达式3) 语句3

...

else 语句 n

## 2.4.2 多重选择结构——嵌套的if结构

```
#include<iostream>
using namespace std;
int main() {
    int x, y;
    cout << "Enter x and y:";
    cin >> x >> y;
    if (x != y)
        if (x > y)
            cout << "x > y" << endl;
        else
            cout << "x < y" << endl;
    else
        cout << "x = y" << endl;
    return 0;
}
```

例2-3：输入两个整数，比较两个数的大小。

### 例 2-3 ( 续 )

运行结果1:

Enter x and y:5 8

$x < y$

运行结果2:

Enter x and y:8 8

$x = y$

运行结果3:

Enter x and y:12 8

$x > y$

## 嵌套的if结构（续）

- 语法形式

if( )

    if( ) 语句 1

    else 语句 2

else

    if( ) 语句 3

    else 语句 4

- 注意

语句 1、2、3、4 可以是复合语句，每层的 if 与 else 配对，或用 { } 来确定层次关系。



### switch 语句

- 例2-4： 输入一个0~6的整数，转换成星期输出。

## 例 2-4 ( 续 )

```
#include <iostream>
using namespace std;
int main() {
    int day;
    cin >> day;
    switch (day) {
    case 0: cout << "Sunday" << endl; break;
    case 1: cout << "Monday" << endl; break;
    case 2: cout << "Tuesday" << endl; break;
    case 3: cout << "Wednesday" << endl; break;
    case 4: cout << "Thursday" << endl; break;
    case 5: cout << "Friday" << endl; break;
    case 6: cout << "Saturday" << endl; break;
    default:
        cout<<"Day out of range Sunday .. Saturday"
            << endl;
        break;
    }
    return 0;
}
```

## switch语句（续）

- 一般形式

switch (表达式)

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

- 执行顺序

以case中的常量表达式值为入口标号，由此开始顺序执行。因此，每个case分支最后应该加break语句。

## switch语句（续）

- case分支可包含多个语句，且不用{ }。
- 表达式、判断值都是int型或char型。
- 若干分支执行内容相同可共用一组语句。

### 2.4.3 循环结构 ——while语句

- 例2-5 求自然数1~10之和

分析：本题需要用累加算法，累加过程是一个循环过程，可以用while语句实现。

## 例 2-5 ( 续 )

```
#include <iostream>
using namespace std;
int main() {
    int i = 1, sum = 0;
    while (i <= 10) {
        sum += i; //相当于sum = sum + i;
        i++;
    }
    cout << "sum = " << sum << endl;
    return 0;
}
```

运行结果:  
sum = 55

## while 语句（续）

- 形式

while (表达式) 语句

↑  
可以是复合语句，其中必须含有改变条件表达式值的语句。

- 执行顺序

先判断表达式的值，若为 **true** 时，执行语句。

## do-while 语句

```
#include <iostream>
using namespace std;
int main() {
```

例2-6：输入一个数，将各位数字翻转后输出

```
    int n, right_digit, newnum = 0;
    cout << "Enter the number: ";
    cin >> n;

    cout << "The number in reverse order is  ";
    do {
        right_digit = n % 10;
        cout << right_digit;
        n /= 10; //相当于n=n/10
    } while (n != 0);
    cout << endl;
    return 0;
}
```



### 例 2-6 ( 续 )

运行结果:

Enter the number: 365

The number in reverse order is 563

## do-while 语句（续）

- 一般形式

do 语句

while (表达式)

← 可以是复合语句，其中必须含有改变条件表达式值的语句。

- 执行顺序

先执行循环体语句，后判断条件。

表达式为 true 时，继续执行循环体

- 与while语句的比较：

while 语句执行顺序

先判断表达式的值，为true时，再执行语句

## 例 2-7 用do-while语句编程，求自然数1~10之和

```
//2_7.cpp
#include <iostream>
using namespace std;
int main() {
    int i = 1, sum = 0;
    do {
        sum += i;
        i++;
    } while (i <= 10);
    cout << "sum = " << sum << endl;
    return 0;
}
```

## 对比下面两个程序的不同

程序1:

```
#include <iostream>
using namespace std;
int main() {
    int i, sum = 0;
    cin >> i;
    while (i <= 10) {
        sum += i;
        i++;
    }
    cout<< "sum= " << sum
        << endl;
    return 0;
}
```

程序2:

```
#include <iostream>
using namespace std;
int main() {
    int i, sum = 0;
    cin >> i;
    do {
        sum += i;
        i++;
    } while (i <= 10);
    cout << "sum=" << sum
        << endl;
    return 0;
}
```

## for 语句

### 语法形式

for (初始语句; 表达式1; 表达式2) 语句

循环前先求解      |      每次执行完循环体后求解  
为true时执行循环体

**例2-8：输入一个整数，求出它的所有因子。**

## 例 2-8 （续）

```
#include <iostream>
using namespace std;
int main() {
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;
    cout << "Number " << n << " Factors ";

    for (int k = 1; k <= n; k++)
        if (n % k == 0)
            cout << k << " ";
    cout << endl;
    return 0;
}
```

### 例 2-8 （ 续 ）

运行结果1:

Enter a positive integer: 36

Number 36 Factors 1 2 3 4 6 9 12 18 36

运行结果2:

Enter a positive integer: 7

Number 7 Factors 1 7

## 2.4.4 循环结构与选择结构的嵌套

```
#include <iostream>
using namespace std;
int main() {
    for (int n = 100; n <= 200; n++) {
        if (n % 3 != 0)
            cout << n;
    }
    return 0;
}
```

举例



## 2.4.4 循环结构与选择结构的嵌套

### 例2-10

- 入一系列整数，统计出正整数个数*i*和负整数个数*j*，读入0则结束。
- 分析：
  - 需要读入一系列整数，但是整数个数不定，要在每次读入之后进行判断，因此使用while循环最为合适。循环控制条件应该是 $n \neq 0$ 。由于要判断数的正负并分别进行统计，所以需要在循环内部嵌入选择结构。

## 例 2-10 ( 续 )

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

int main() {
    int i = 0, j = 0, n;
    cout << "Enter some integers please (enter 0 to
quit): " << endl;
    cin >> n;
    while (n != 0) {
        if (n > 0) i += 1;
        if (n < 0) j += 1;
        cin >> n;
    }
    cout << "Count of positive integers: " << i << endl;
    cout << "Count of negative integers: " << j << endl;
    return 0;
}
```

## 2.4.5 其他控制语句

- **break**语句

使程序从循环体和**switch**语句内跳出，继续执行逻辑上的下一条语句。不宜用在别处。

- **continue** 语句

结束本次循环，接着判断是否执行下一次循环。

- **goto** 语句

**goto**语句的作用是使程序的执行流程跳转到语句标号所指定的语句。

## 2.5 枚举类型

- 只要将需要的变量值一一列举出来，便构成了一个枚举类型。
- 枚举类型的声明形式如下：  
`enum 枚举类型名 {变量值列表};`
- 例如：  
`enum Weekday  
{SUN, MON, TUE, WED, THU, FRI, SAT};`

## 枚举类型应用说明

- 对枚举元素按常量处理，不能对它们赋值。例如，不能写：`SUN = 0;`
- 枚举元素具有默认值，它们依次为：`0,1,2,.....`。
- 也可以在声明时另行指定枚举元素的值，如：  
`enum Weekday{SUN=7,MON=1,TUE,WED,THU,FRI,SAT};`
- 枚举值可以进行关系运算。
- 整数值不能直接赋给枚举变量，如需要将整数赋值给枚举变量，应进行强制类型转换。

## 例 2-11

- 设某次体育比赛的结果有四种可能：胜（**WIN**）、负（**LOSE**）、平局（**TIE**）、比赛取消（**CANCEL**），编写程序顺序输出这四种情况。
  - ▣ 分析：由于比赛结果只有四种可能，所以可以声明一个枚举类型，声明一个枚举类型的变量来存放比赛结果。

## 例 2-11 ( 续 )

```
#include <iostream>
using namespace std;
enum GameResult {WIN, LOSE, TIE, CANCEL};
int main() {
    GameResult result;
    enum GameResult omit = CANCEL;
    for (int count = WIN; count <= CANCEL;
        count++) {
        result = GameResult(count);
        if (result == omit)
            cout << "The game was cancelled" <<
endl;
        else {
            cout << "The game was played ";
            if (result == WIN)
                cout << "and we won!";
            if (result == LOSE)
                cout << "and we lost.";
            cout << endl;
        }
    }
    return 0;
}
```

### 例 2-11 ( 续 )

运行结果

The game was played and we won!

The game was played and we lost.

The game was played

The game was cancelled



# 小结

- 主要内容
  - C++语言概述、基本数据类型和表达式、数据的输入与输出、算法的基本控制结构、自定义数据类型
- 达到的目标
  - 掌握C++语言的基本概念和基本语句，能够编写简单的程序段。