第二部分　C++语言规范

第 19 章　类型与表达式

本章将深入探讨 C++的强类型化系统概念，也会讲解如何求解并转换表达式。

首先将形式化地给出一些术语的定义。运算符是一种特殊类型的函数，它对操作数执行计算并返回结果，操作数是提供给运算符的实参。

可以将运算符看成常规的函数，但有一些运算符使用中缀运算符符号(例如，+，-，*，/等)。因此，除了可以使用更长的函数调用句法之外(例如，`str3 = operator+(str1, str2);`)，还可以使用更具可读性的中缀句法(例如，`str3 = str1 + str2;`)。

表达式可以由带有单一操作数、多个操作数的运算符或者包含实参的函数组成。每一个表达式都具有类型和值。值是通过将运算符(或者函数)的定义应用于操作数(或者实参)而求得的。

19.1　运算符

根据用途的不同，运算符被分成如下几个大类：

赋值运算符	=, +=, *=, …
算术运算符	+, -, *, /, %
关系运算符	<, <=, >, >=, ==, !=
逻辑运算符	&&, ||, !
位运算符	&, |, ^, ~, <<, >>
内存管理运算符	new, delete, sizeof
指针运算符和访问运算符	*, &, ., ->, [], ()
作用域解析运算符	::
其他运算符	条件(?:)，逗号(,)

正如表 19.1 中所示，C++标准定义的一些关键字可以充当某些运算符符号的别名。

表 19.1　运算符的别名

运　算　符	别　　名	运　算　符	别　　名
&&	and(与)	^	xor(异或)
&	bitand(位与)	^=	xor_eq(异或等于)
&=	and_eq(与等于)	!	not(非)
||	or(或)	!=	not_eq(非等于)
|	bitor(位或)	~	compl(补)
|=	or_eq(或等于)		

对于内置类型，运算符具有预先定义的含义，但并不是所有的运算符都是为内置类型而定义的。

运算符的特性

运算符具有下列几个特性：

- 优先级
- 结合性
- 所要求的操作数数量

表 19.2 中列出了全部 C++运算符以及它们的特性，按照优先级和用途分组，高优先级的先列出。

- “操作数”列包含运算符所要求的操作数数量。
- “描述”列包含针对内置类型该运算符的传统含义。
- “结合性”列给出的结合性，表示如果同一个运算符在一个表达式出现多次，则应该如何对其进行求值。
  - “左”表示结合性为从左到右。例如：

    ```
    d = a + b + c;     //首先求值 a + b，然后求值(a + b) + c
    ```

    赋值运算符是最后求值的，因为它具有更低的优先级。
  - “右”表示结合性为从右到左。例如：

    ```
    c = b = a;         //先将 a 赋值给 b，然后赋值给 c
    ```
- “重载性”列表示这个运算符是否可以针对定制类型被重载(重定义)。

  这个列可能的值包括：
  - Y。运算符可以被重载成全局函数或者成员函数。
  - M。运算符只可以被重载成类成员函数。
  - N。运算符不能被重载。

19.1.1　运算符表

表 19.2　C++运算符

运算符	操作数	描述	例子	结合性	重载性
::	1	全局作用域解析	:: name	右	N
::	2	类/命名空间作用域解析	*className::memberName*	左	N
->	2	通过指针的成员选择器	*ptr->memberName*	左	N
.	2	通过对象的成员选择器	*obj.memberName*	左	N
->	1	智能指针	*obj->member*	右	M
[ ]	2	下标运算符	*ptr[expr]*	左	M
()	任意个	函数调用	*function(argList)*	左	N
()	任意个	值构造	*className(argList)*	左	M
++	1	后递增	*varName++*	右	Y
--	1	后递减	*varName--*	右	Y
typeid	1	类型识别	typeid(*type*)或者typeid(*expr*)	右	N
dynamic_cast	2	运行时经检验的类型转换	dynamic_cast<*type*>(*expr*)	左	N
static_cast	2	编译时经检验的类型转换	static_cast<*type*>(*expr*)	左	N
reinterpret_cast	2	未检验的转换	reinterpret_cast<*type*>(*expr*)	左	N
const_cast	2	常量转换	const_cast<*type*>(*expr*)	左	N
sizeof	1	字节大小	sizeof *expr*或者sizeof(*type*)	右	N

(续表)

运算符	操作数	描述	例子	结合性	重载性
++	1	前递增	*++varName*	右	Y
--	1	前递减	*--varName*	右	Y
~	1	逐位取反	*~ expr*	右	Y
!	1	逻辑非	*! expr*	右	Y
+, -	1	一元加，一元减	*+expr* 或者 *-expr*	右	Y
*	1	指针解引用	** ptr*	右	Y
&	1	取址	*& lvalue*	右	Y
new	1	分配	new *type* 或者 new *type*(*exprlist*)	右	Y
new [ ]	2	分配数组	new *type*[*size*]	左	Y
delete	1	解除内存分配	delete *ptr*	右	Y
delete [ ]	1	解除数组分配	delete [ ] *ptr*	右	M
()	2	C 风格的类型转换	(*type*) *expr*	右	N[b]
->*	2	通过指针的成员指针选择器	*ptr->*ptrToMember*	左	M
.*	2	通过对象的成员指针选择器	*obj.*ptrToMember*	左	N
*	2	乘	*expr1 * expr2*	左	Y
/	2	除	*expr1 /expr2*	左	Y
%	2	求余	*expr1 % expr2*	左	Y
+	2	加	*expr1 + expr2*	左	Y
-	2	减	*expr1 expr2*	左	Y
<<	2	位左移	*expr << shiftAmt*	左	Y
>>	2	位右移	*expr >> shiftAmt*	左	Y
<	2	小于	*expr1 < expr2*	左	Y
<=	2	小于或者等于	*expr1 <= expr2*	左	Y
>	2	大于	*expr1 > expr2*	左	Y
>=	2	大于或者等于	*expr1 >= expr2*	左	Y
==	2	等于[c]	*expr1 == expr2*	左	Y
!=	2	不等于	*expr1 != expr2*	左	Y
&	2	位与	*expr1 & expr2*	左	Y
^	2	位异或	*expr1^e2*	左	Y
|	2	位或	*expr1 | expr2*	左	Y
&&	2	逻辑与	*expr1 && expr2*	左	Y
||	2	逻辑或	*expr1 || expr2*	左	Y
=	2	赋值	*expr1 = expr2*	右	Y
*=	2	乘并赋值	*expr1 *= expr2*	右	Y
/=	2	除并赋值	*expr1 /= expr2*	右	Y
%=	2	求余并赋值	*expr1 %= expr2*	右	Y
+=	2	加并赋值	*expr1 += expr2*	右	Y
-=	2	减并赋值	*expr1 -= expr2*	右	Y
<<=	2	左移并赋值	*expr1 <<= expr2*	右	Y
>>=	2	右移并赋值	*expr1 >>= expr2*	右	Y
&=	2	与并赋值	*expr1 &= expr2*	右	Y

（续表）

运算符	操作数	描述	例子	结合性	重载性
|=	2	或并赋值	*expr1* |= *expr2*	右	Y
^=	2	异或并赋值	*expr1* ^= *expr2*	右	Y
?:	3	条件表达式	*bool* ? *expr* : *expr*	左	N
throw	1	抛出异常	*throw expr*	右	N
	2	顺序求值(逗号)	*expr* , *expr*	左	Y

a 可以将函数调用运算符声明成带任意数量的操作数。
b 类型转换运算符可以使用构造函数或者转换运算符来转换定制类型。
c 不能将这个运算符用于 float 或者 double 操作数。它要求的是精确匹配，这与体系结构有关且可能导致意料之外的结果。

19.2　语句与控制结构

语句是可执行的代码块。控制结构是控制其他语句的执行的语句。本章将形式化地定义一些语言元素，并会给出一些可用的控制结构类型。

19.2.1　语句

C++程序包含的语句可以改变由程序管理的存储状态，并且可以决定程序的执行流程。存在多种类型的 C++语句，其中的大多数都继承自 C 语言。首先，最简单的语句是以分号结尾的语句：

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

其次，是复合语句或者语句块，它由包含在一对大括号中的语句序列组成：

```
{
    int temp = x;
    x = y;
    y = temp;
}
```

上面是一个简单的复合语句，它包含三条简单语句，从上到下依次执行。变量 temp 为这个语句块的局部变量，当到达语句块的结尾处时，它会被销毁。复合语句可以包含其他的复合语句。

通常而言，可以将复合语句放于简单语句能够存在的任何位置。但是，反过来并不成立。特别地，函数定义

```
double area(double length, double width) {
    return length * width;
}
```

不能被替换成

```
double area(double length, double width)
   return length * width;
```

函数定义的语句体必须总是一个语句块。

19.2.2　选择语句

任何编程语言都至少存在一种控制结构，它使得程序的执行流程能够根据某个布尔条件的输出而变化。C 和 C++语言中都有 if 语句和 switch 语句。if 语句通常具有下列形式：

```
if(boolExpression)
  statement
```

它还可以具有一个可选的 else 分支：

```
if(boolExpression)
  statement1
else
  statement2
```

条件语句可以嵌套，这意味着它们可以变得相当复杂。需记住的一个重要原则是：如果紧挨在 else 或者 else if 子句前面的 if 条件中的 *boolExpression* 求值为 false，则会执行这个子句。如果程序逻辑允许省略某些 else 子句，则这个原则可能会使人迷惑。考虑下面这个故意设计错误的例子，其中 x 为 int 类型：

```
if (x>0)
    if (x > 100)
        cout << "x is over a hundred";
else
    if (x == 0)  // no! this cannot be true -the indentation is misleading
        cout << "x is 0";
else
    cout << "x is negative"; // no! x is between 1 and 100 inclusive!
```

利用大括号，就可以找出并修复这个逻辑错误：

```
if (x>0) {
    if (x > 100)
        cout << "x is over a hundred";
}
else
    if (x == 0)  // now this is possible.
        cout << "x is 0";
    else
        cout << "x is negative";
```

没有 else 分支的 if 语句，可以通过将 if 下面的语句放入一对大括号中，使其成为一个复合语句。

switch 语句

switch 是另一种分支结构，它允许根据参数的值执行不同的代码。

```
switch(integralExpression) {
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
        ...

    case valuen:
        statementn;
        break;
    default:
        defaultStatement;
}
nextStatement;
```

switch 语句是一种具有计算功能的 goto 语句。switch 语句中的每一个 case 后面都有一个唯一的标记值，它会与 *integralExpression* 比较。

如果某个 case 标记值等于 *integralExpression*，switch 就会跳到这个 case 后面的语句执行，直到到达 switch 语句块的结尾处或者遇到一条分支语句(例如，break)。

如果 *integralExpression* 没有与之相等的 case 标记值，就会跳到可选的 default 标记处执行。如果没有 default 标记且不存在匹配的 case 标记，则会跳到 *nextStatement* 去执行。

integralExpression 必须是一个能够求值为整数的表达式。除了 default 标记外，每一个 case 标记值都必须是一个整数常量[①]。

与上面相同的任何 switch 语句，都可以被重写为一个长的 if…else 语句。但是，switch 语句的运行时性能要好得多，因为它只需一次比较且只会执行一个分支。对应的 if…else 语句如下：

```
if(integralExpression == value1)
   statement1;
else if(integralExpression == value2)
   statement2;
...
else if(integralExpression == valuen)
   statementn;
else
   defaultStatement;
```

注意

长的复合条件语句和 switch 语句应当避免在面向对象编程中使用(除非它们只位于工厂代码中)，因为它们会使函数变得复杂并难以维护。

如果每一个 case 都能够被重写为不同类的方法，则可以使用策略模式(以及虚表)，而不必自己编写 switch 语句。

19.2.2.1　练习：选择语句

假设你是计算机，预测示例 19.1 的输出结果。然后，在计算机上运行它并将结果与预测值进行比较。

示例 19.1　src/early-examples/nestedif.cpp

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

void nestedif1 () {
    int m = 5, n = 8, p = 11;
    if (m > n)
        if (p > n)
            cout << "red" << endl;
        else
            cout << "blue"  << endl;
```

① case 标记与 goto 标记并不相同，后者被用作声名狼藉的 goto 语句的执行目标。goto 标记必须是标志符。特别地，这种标志符不能是整数。

```
}


void nestedif2() {
    int m = 5, n = 8,  p = 11;
    if (m > n) {
        if (p > n)
            cout << "red"  << endl;
    } else
        cout << "blue" << endl;
}


int main() {
    nestedif1();
    nestedif2();
    return 0;
}
```

迭代

C++提供了三种迭代结构：

1. while 循环

```
while ( loopCondition ) {
   loopBody
}
```

   a. 首先求值 *loopCondition*。
   b. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。

2. Do…while 循环

```
do {
   loopBody
} while ( loopCondition ) ;
```

   a. 首先执行 *loopBody*。
   b. 求值 *loopCondition*。
   c. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。

3. for 循环

```
for ( initStatement; loopCondition; incrStmt ) {
   loopBody
}
```

   a. 首先执行 *initStatement*。
   b. 不断执行 *loopBody*，直到 *loopCondition* 变为 false。
   c. 每次执行完 *loopBody* 后，执行 *incrStmt*。

对于这些迭代结构，只要 *loopCondition* 求值为 true，就会重复地执行 *loopBody* 代码。do 循环与另外两种不同，它的 *loopCondition* 在循环的底部进行检验，所以它的 *loopBody* 总是至少会执行一次。

一种常见的编程错误是在 while 的后面放置一个分号：

```
while (notFinished());
    doSomething();
```

第一个分号会终止整个 while 语句，并会导致一个具有空 loopBody 的循环。尽管 doSomething()被缩进了，但是它不会在循环中执行。*loopBody* 负责改变 *loopCondition* 的值。如果开始时 notFinished()不为 true，则空 loopBody 会导致无限循环。如果开始时 notFinished()为 false，则循环会立即终止，而 doSomething()会恰好执行一次。

为了对循环中执行的代码进行更好的控制，C++提供了 break 语句和 continue 语句：

```
while ( moreWorkToDo ) {
    statement1;
    if ( specialCase ) continue;
    statement2;
    if ( noMoreInput ) break;
    statement3;
// continue jumps here
}
// break jumps here
```

break 会跳出当前的控制结构，不管这个控制结构是 switch，for，while 还是 do…while。

continue 只在循环内部执行，它会跳过当前迭代中剩下的语句而去检验 *moreWorkToDo* 条件。示例 19.2 中给出了 continue 语句和 break 语句的用法。

示例 19.2　src/continue/continue-demo.cpp

```
#include <QTextStream>
#include <cmath>

int main() {
   QTextStream cout(stdout);
   QTextStream cin(stdin);
   int num(0), root(0), count;
   cout << "How many perfect squares? "<< flush;
   cin >> count;
   for(num = 0;; ++num) {
      root = sqrt(num);                                    1
      if(root * root != num)
         continue;
      cout << num << endl;
      --count;
      if(count == 0)
         break;
   }
}
```

1 将 sqrt 转换成 int。

19.2.3.1　练习：迭代

1. 编写函数 isPrime(int n)，如果 n 为素数就返回 true，否则返回 false。提供一个交互式的 main()函数来测试这个函数。
2. 编写函数 primesBetween(int min, int max)，它在屏幕上显示位于 min 和 max 之间的全部素数。提供一个交互式的 main()函数来测试这个函数。

3. 编写函数 power2(int n)，它计算并返回 2 的 n 次幂。提供一个交互式的 main() 函数来测试这个函数。
4. 编写一个二进制算法函数 binLog(int n)，它计算并返回一个与 $\log_2 n$ 的整数部分相等的 int 值，其中 n 为正数。这等价于找出小于或者等于 n 的 2 的最大指数。例如，binLog(25)等于 4。至少存在两种简单的迭代方式可以执行这种计算。提供一个交互式的 main()函数来测试这个函数。

19.2.4 复习题

1. 复合语句与简单语句有什么不同?
2. 对于任意给定的 switch 值，如何保证至少有一个分支会被执行?
3. 三种迭代结构各自的优缺点是什么？对于每一种结构，在哪些情况下应该使用其中的一种而不是另外两种?

19.3 逻辑表达式的求值

在 C 和 C++语言中，一旦整个表达式的逻辑值已经确定，对逻辑表达式的求值过程就会停止。这种短路机制可能导致某些操作数不会被求值。当且仅当下面的全部操作数都为 true 时，表达式

$expr_1$ && $expr_2$ && ... && $expr_n$

的值才会被求值为 true。如果其中有一个或者多个操作数为 false，则表达式的值就为 false。对这个表达式的求值是从左到右依次进行的，且只要遇到某个操作数的值为 false，求值过程就会停止(且返回值 false)。

类似地，当且仅当下面的操作数全都为 false 时，表达式

$expr_1$ || $expr_2$ || ... || $expr_n$

的值才会被求值为 false。对这个表达式的求值是从左到右依次进行的，且只要遇到某个操作数的值为 true，求值过程就会停止(且返回值 true)。

程序员经常要开发包含类似如下语句的系统：

```
if( x != 0 &&  y/x < z) {
// do something ...
}
else {
// do something else ...
}
```

如果 x 等于 0，则第二个表达式会导致运行时错误。幸运的是，这不会发生。

逻辑表达式经常同时使用&&和||。需重点记住的是，&&的优先级比||高。换句话说，

$expr_1$ || $expr_2$ && $expr_3$

表示

$expr_1$ || ($expr_2$ && $expr_3$)

而不是

($expr_1$ || $expr_2$) && $expr_3$

19.4　枚举

第 2 章中简要探讨过如果通过定义类来在 C++语言中增加新的类型。C++中增加新类型的另一种途径也值得探讨。

关键字 enum(枚举)被用来向 C++标志符赋整数值。例如，当设计执行位操作的数据结构时，一种方便的做法是为各个位掩码命名。Qt 经常为此使用枚举。枚举的一个恰当例子位于 QFileDialog::Option 中。枚举的主要作用是使代码更具可读性，从而更易于维护。例如：

```
enum {UNKNOWN, JAN, FEB, MAR };
```

定义了四个常量标志符，它们从 0 开始按升序编号。这条语句等价于：

```
enum {UNKNOWN=0, JAN=1, FEB=2, MAR=3};
```

标志符 JAN，FEB，MAR 被称为枚举器(enumerator)。可以将它们初始化成任意的整数值。由于枚举器为 const 项，它们的名称经常以大写字母形式出现(但是，Qt 并没有遵循这个规范)。

```
enum Ages {MANNY = 10, MOE, JACK = 23,
           SCOOTER = JACK + 10};
```

如果第一个枚举器 MANNY 没有被初始化，则它会被自动赋予值 0。由于 MANNY 已经被初始化成 10 而 MOE 没有赋值，所以 MOE 的值为 11。各个枚举器的值可以相同。

为 enum 赋予一个名称，就定义了一个新类型。例如：

```
enum Winter {JAN=1, FEB, MAR, MARCH = MAR };
```

名称 Winter 被称为标记名(tag name)。现在，就可以声明 Winter 类型的变量了。

```
Winter m = JAN;
int i = JAN;    // OK - enum can be implicitly converted to int.
m = i;          // error - explicit cast is required.
m = static_cast<Winter>(i);  // OK
i = m;          // OK
m = 4;          // error
```

在各自的作用域内，标记名和枚举器必须具有不同的标志符。枚举器值可以被隐式地转换成普通的整数类型，但是如果没有进行显式地转型，反过来是不成立的。示例 19.3 演示了枚举的用法，并输出了几个枚举器的值。

示例 19.3　src/enums/enumtst.cpp

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

int main(int, char** ) {
    enum Signal { off, on } sig;                          1
    sig = on;
    enum Answer { no, yes, maybe = -1 };                  2
    Answer ans = no;                                      3
//    enum Neg {no,false} c;                              4
    enum { lazy, hazy, crazy } why;                       5

    int  i, j = on;                                       6
    sig = off;
```

```
    i = ans;
//  ans = s                                                    7
    ans = static_cast<Answer>(sig);                            8
    ans = (sig ? no : yes);
    sig = static_cast<Signal>(9);                              9
    Signal sig2(sig);                                          10
    why = hazy;
    cout << "sig2, ans, i, j, why "
         << sig2 << ans << i << j << why << endl;
    return 0;
}
```

1 在一行中给出了一个新类型，两个新枚举标志符以及一个变量定义。
2 为类型/枚举定义。
3 枚举的一个实例。
4 对标志符的非法重新定义。
5 一个未命名的枚举变量。
6 枚举总是可以被转换成 `int` 值。
7 枚举类型之间的转换不能隐式地进行。
8 可以用 `cast` 进行转换。
9 最好不要这样做。
10 添加了一个未命名的枚举器吗?

输出如下所示。

```
src/enums> ./enums
sig2, ans, i, j, why 91011
src/enums>
```

19.5 有符号整型类型与无符号整型类型

这一节讲解有符号(signed)整型类型与无符号(unsigned)整型类型的区别。

任何整型类型的对象 x 的底层二进制表示都是这样的(假设有 n 位存储空间):

$$d_{n-1}d_{n-2}\ldots d_2d_1d_0$$

其中每一个 d 为 0 或者 1。对 x 计算等价的十进制值，与 x 为有符号类型还是无符号类型相关。如果 x 为无符号类型，则其十进制值为

$$d_{n-1}*2^{n-1} + d_{n-2}*2^{n-2} +\ldots+ d_2*2^2 + d_1*2^1 + d_0*2^0$$

因此，能够用无符号整数表示的最大(正)值是:

$$2^n - 1 = 1*2^{n-1} + 1*2^{n-2} +\ldots+ 1*2^2 + 1*2^1 + 1*2^0$$

如果 x 为有符号类型，则其十进制值为

$$d_{n-1}*-(2^{n-1}) + d_{n-2}*2^{n-2} +\ldots+ d_2*2^2 + d_1*2^1 + d_0*2^0$$

因此，能够用有符号整数表示的最大(正)值是:

$$2^{n-1} - 1 = 0*-(2^{n-1}) + 1*2^{n-2} +\ldots+ 1*2^2 + 1*2^1 + 1*2^0$$

这被称为“2 的补码”表示。为了确定负的有符号整数的表示，需进行如下步骤:

1. 计算这个数的“1 的补码”(即将每一个位都变成它的补)。
2. 将上一步中得到的值加 1。

8 位整数举例

假设一个小型系统只使用 8 位来表示一个数。在这个系统中，最大的无符号整数是：

```
11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

但是对于同一个数，如果将其作为有符号整数，则将是：

```
11111111 = -128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -1
```

19.5.1　练习：有符号整型类型与无符号整型类型

假设有一台计算机，对于示例 19.4、示例 19.5 和示例 19.6，模拟计算机的动作执行给定的代码，并设想输出结果。然后，在计算机上编译并运行这些代码，检验你的设想是否正确。如果存在差异，请给出理由。

1. 示例 19.4　src/types/tctest1.cpp

```
#include <iostream>
using namespace std;
int main() {
  unsigned n1 = 10;
  unsigned n2 = 9;
  char *cp;
  cp = new char[n2 - n1];
  if(cp == 0)
    cout << "That's all!" << endl;
  cout << "bye bye!" << endl;
}
```

2. 示例 19.5　src/types/tctest2.cpp

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

int main() {
  int x(7), y = 11;
  char ch = 'B';
  double z(1.34);
  ch += x;
  cout << ch << endl;
  cout << y + z << endl;
  cout << x + y * z << endl;
  cout << x / y * z << endl;
}
```

3. 示例 19.6　src/types/tctest3.cpp

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

bool test(int x, int y)
{ return x / y; }

int main()
{ int m = 17, n = 18;
  cout << test(m,n) << endl;
```

```
  cout << test(n,m) << endl;
  m += n;
  n /= 5;
  cout << test(m,n) << endl;
}
```

4. 在任何计算或者比较中，如果混用有符号数和无符号数，则通常不是一种好做法。预测示例 19.7 的输出结果。

示例 19.7 src/types/unsigned.cpp

```
#include <iostream>
using namespace std;
int main() {
  unsigned u(500);
  int i(-2);
  if(u > i)
    cout << "u > i" << endl;
  else
    cout << "u <= i" << endl;
  cout << "i - u = " << i - u << endl;
  cout << "i * u = " << i * u << endl;
  cout << "u / i = " << u / i << endl;
  cout << "i + u = " << i + u << endl;
}
```

19.6 标准表达式转换

这一节探讨表达式转换，包括通过提升和降级进行的隐式类型转换，以及通过各种转换机制进行的显式转型。

假设 x 和 y 都为数字型变量。“x op y”形式的表达式同时具有值和类型。当求值这个表达式时，会使用 x 和 y 的临时副本。如果 x 和 y 具有不同的类型，则在执行操作之前，具有较短类型的那一个变量需要进行转换(变宽)。

对某个数进行的隐式转换如果能保持它的值，则被称为提升(promotion)。

x op *y* 的自动表达式转换规则

1. 任何 bool, char, signed char, unsigned char, enum, short int 或者 unsigned short int 类型，都可以被提升为 int 类型。这被称为整型提升(integral promotion)。
2. 上一步完成之后，如果表达式为混合类型，则较小类型的操作数会被提升为较大类型，而表达式的值就具有这种较大类型。
3. 类型的层次关系见图 19.1 中的箭头所示。

unsigned 与 long 之间的关系与它们的实现有关。例如，在将 int 实现成具有与 long 相同的字节的系统中，就不能将 unsigned 提升成 long，因此提升过程就会绕开 long 而将 unsigned 提升成 unsigned long。假定具有如下的声明：

```
double d;
int i;
```

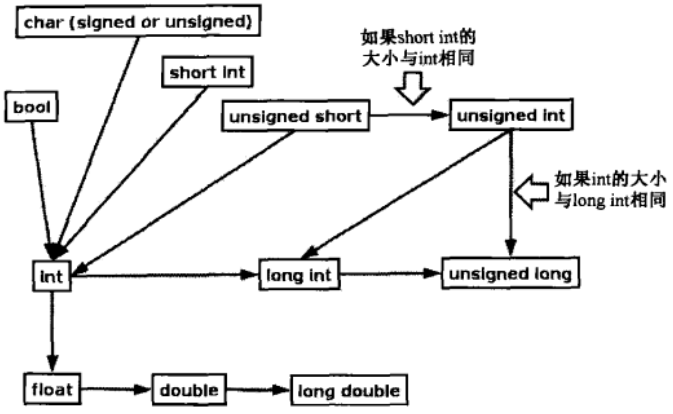

图 19.1　基本类型的层次关系

通常而言，执行语句“d = i;”时进行的提升不会存在什么问题。但是，赋值语句“i = d;”引起的降级会导致信息的丢失。如果编译器允许这种赋值，则 d 的小数部分将被丢弃。示例 19.8 中演示了前面讨论过的几种转换。

示例 19.8　src/types/mixed/mixed-types.cpp

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

int main() {
    int i, j = 88;
    double d = 12314.8723497;
    cout << "initially d = " << d
         << "  and j = " << j << endl;
    cout << "The sum is: " << j + d << endl;
    i = d;
    cout << "after demoting d,  i = " << i << endl;
    d = j;
    cout << "after promoting j,  d = " << d << endl;
}
```

以下是编译并运行的结果。

```
src> g++ mixed-types.cpp
mixed-types.cpp: In function `int main()':
mixed-types.cpp:10: warning: converting to `int' from `double'
src> ./a.out
initially d = 12314.9  and j = 88
The sum is: 12402.9
after demoting d,  i = 12314
after promoting j,  d = 88
src>
```

存在另一种可以根据需要随意执行的隐式转换。能够被求值成指针、整型值或者浮点值的表达式，都可以被转换成布尔值。如果表达式的值为 0，则布尔结果为 false，否则为 true。示例 19.9 演示了这种转换。

示例 19.9 src/types/convert2bool.cpp

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

int main() {
   int j(5);
   int* ip(&j);
   int* kp(0);
   double y(3.4);
   if(y)
      cout << "y looks like true to me!" << endl;
   else
      cout << "y looks like false to me!" << endl;
   cout << "ip looks like " << (ip ? "true" : "false") << endl;
   cout << "kp looks like " << (kp ? "true" : "false") << endl;
   while(--j) {
      cout << j << '\t';
   }
   cout << endl;
}
```

输出如下所示。

```
src> ./a.out
y looks like true to me!
ip looks like true
kp looks like false
4       3       2       1
src>
```

19.6.1 练习：标准表达式转换

假设有如下的声明：

```
double d = 123.456;
int i = 789, j = -1;
uint k = 10;
```

- d + i 的类型和值分别是什么?
- j + k 的类型和值分别是什么?
- 如果执行类似“d = i;”的提升，会发生什么?
- 如果执行类似“i = d;”的降级，会发生什么?

19.7 显式转换

显式转换被称为转型(cast)。有时候，转型是需要的，但是它有过度使用的迹象，是某些错误的主要来源。C++的创立者 Bjarne Stroupstrup 公开表示要尽可能少使用它。

由于 C++起源于 C 语言，所以 C++支持老式的(不安全的)C 风格的转型：

```
(type)expr
```

例如：

```
double d=3.14;
int i = (int)d;
```

C++还支持另一种构造函数风格的转型语法：

```
Type t = Type(arglist)
```

转型会创建一个指定类型的临时值并将其压入程序栈中。如果 *Type* 为类，则创建的是一个临时对象，且它会被合适的转换构造函数初始化；如果 *Type* 为原始类型，则 *Type(arg)* 与 *(Type)arg* 等价。临时值被置于栈中，其存在时间只到其所在的表达式被求值为止。自此以后，它就会被销毁。

例如：

```
double d = 3.14;
Complex c = Complex(d);
```

19.8　用 ANSI C++类型转换进行更安全的类型转换

ANSI C++中增加了四个转型运算符（见表 19.3），它们具有临时风格的语法，能够更清晰地表达程序员的意图，并可使转型能容易在代码中体现出来。

表 19.3　ANSI 类型转换

`static_cast< type >(expr)`	用于相关类型之间的转换
`const_cast< type > expr`	用于转换掉 const（常量性）或者 volatile（易变性）
`dynamic_cast< type >(expr)`	用于在继承层次中安全地导航
`reinterpret_cast< type >(expr)`	用于不相关类型之间指针的类型转换

19.8.1　static_cast 与 const_cast

只要编译器知道从 *expr* 到 *DestType* 的隐式转换，*static_cast<DestType>(expr)* 就会将值 *expr* 转换成 *DestType* 类型。编译时会进行全部的类型检验。

```
static_cast<char>('A' + 1.0);
static_cast<double>(static_cast<int>(y) + 1);
```

static_cast 运算符会在相关类型之间进行转换，比如从一种指针类型转换成另一种指针类型，从枚举类型转换成整型类型，或者将浮点类型转换成整型类型。这些转换都被很好地定义了，且是可移植的、可逆的。编译器可以对每一个 static_cast 进行某些最小化的类型检验。

static_cast 不能转换掉常量性。如果要创建 *expr* 的一个非常量版本，则必须使用 const_cast<*DestType*>(*expr*)。

这种情况下，只有当存在或者不存在 const/volatile 时，*DestType* 的类型才可以与 *expr* 的类型不同。

对于 int i，static_cast<double>(i)会创建一个 double 类型的临时副本，它具有值 i。这个转型操作不会使变量 i 发生改变。示例 19.10 中包含了这两种转型操作。

示例 19.10　src/ansicast/m2k.cpp

```
// Miles are converted to kilometers.
#include <QTextStream>

QTextStream cin(stdin);
QTextStream cout(stdout);
```

```
QTextStream cerr(stderr);

const double  m2k = 1.609;     // conversion constant
inline double mi2km(int miles) {
    return (miles * m2k);
}

int main() {
    int  miles;
    double kilometers;
    cout << "Enter distance in miles: " << flush;
    cin >> miles ;
    kilometers = mi2km(miles);
    cout << "This is approximately "
         <<  static_cast<int>(kilometers)
         << "km."<< endl;
    cout << "Without the cast, kilometers = "
         << kilometers << endl;
    double* dp = const_cast<double*>(&m2k);
    cout << "m2k: " << m2k << endl;
    cout << "&m2k: " << &m2k << "  dp: " << dp << endl;
    cout << "*dp: " << *dp << endl;
    *dp = 1.892;                                                    1
    cout << "Can we reach this statement? " << endl;
    return 0;
}
```

1 这里的操作意图是什么?

输出如下所示。

```
Enter distance in miles: 23
This is approximately 37km.
Without the cast, kilometers = 37.007
m2k: 1.609
&m2k: 0x8049048  dp: 0x8049048
*dp: 1.609
Segmentation fault
```

根据示例 19.10 可以得出如下一些结论:

- 混合表达式 `miles * m2k` 被隐式地加宽成 `double` 类型。
- 安全的转型 `static_cast<int>(kilometres)` 会将 `double` 值截尾成 `int` 值。
- 转型操作不会改变 `kilometres` 变量的值。
- 对`*dp` 赋值的结果是未定义的。

转换掉常量性

通常而言，`const_cast` 只用在 `const` 引用和指向非 `const` 对象的指针上。用 `const_cast` 改变 `const` 对象的结果是未定义的，因为 `const` 对象可能被保存在只读内存中(操作系统会保护只读内存)。对于 `const int` 的情况，试图通过转换掉常量性来改变它的结果，与编译器的优化技术有关，经常的情况是将其优化成“消失了”(通过预编译值替换操作)。示例 19.11 中给出了可能发生的一些奇怪行为的结果。

示例 19.11　src/casts/constcast1.cpp

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

int main() {
    const int N = 22;
    int* pN = const_cast<int*>(&N);
    *pN = 33;
    cout << N << '\t' << &N << endl;
    cout << *pN << '\t' << pN << endl;
}
```

输出如下所示。

```
22      0xbf91cfa0
33      0xbf91cfa0
```

前面的输出是用 gcc 4.4.5 获得的，它可能与你的系统上的输出有所不同，因为程序的行为是未定义的。

这个示例中使用了 const_cast 来获得一个 const int 的常规指针。由于 const int 位于栈存储类中，试图改变它的内存不会导致段错误。编译器不能“优化掉” int，const_cast 会告诉它不要这样做。

19.8.1.1　练习：static_cast 与 const_cast

1. 示例 19.11 中，将语句

   ```
   const int N = 22;
   ```

   移到

   ```
   int main() {
   ```

   的上面或者下面，输出有什么不同？请解释。

2. a. 预测示例 19.12 的输出。

   示例 19.12　src/casts/constcast2.cpp

   ```
   #include <iostream>

   void f2(int& n) {
       ++n;
   }

   void f1(const int& n, int m) {
       if (n < m)
           f2(const_cast<int&>(n));
   }

   int main() {
       using namespace std;
       int num1(10), num2(20);
       f1(num1, num2);
       cout << num1 << endl;
   }
   ```

   b. 将 f1() 中调用 f2() 的语句里移除 const_cast，再次预测输出。

19.8.2 reinterpret_cast

reinterpret_cast 被用于与表示相关或者与系统相关的转型中。这类例子包括不相关类型之间的转换(例如，int 到指针)，或者不相关指针类型之间的转换(例如，int*到double*)它不能转换掉常量性。

reinterpret_cast 是危险的，且通常是不可移植的，所以应避免使用。

考虑下面的情形。

```
Spam spam;
Egg* eggP;
eggP = reinterpret_cast<Egg*>(&spam);
eggP->scramble();
```

reinterpret_cast 具有一些 spam 且提供一个 Egg 类型的指针，而没有考虑类型兼容性。eggP 将会把猪肉(spam)重新解释成鸡蛋。

转型的真正用途是什么

有时，C 函数会返回一个 void*指针，指向开发人员已知的某种类型。这种情况下，有必要用类型转换将 void*转换成实际类型。如果确定它指向的是一个 Egg，则合适的转型操作是 reinterpret_cast<Egg*>。对于这种转型，编译器不会进行类型检验，运行时系统也不会。

例如，示例 12.15 的 QMetaType 中，就对 void*使用了 reinterpret_cast。

19.8.3 为什么不使用 C 风格的转型

C 风格的转型操作已经被弃用，不应该再使用。考虑下面的语句：

```
Apple apple;
Orange* orangeP;
// other processing steps ...
orangeP = (Orange*) &apple;
orangeP->peel();
```

这些语句存在的问题是：从这些代码中无法看出开发人员是否知道苹果(Apple)是否与橘子(Orange)“兼容”。这里看不出它是合适的类型转换还是不可移植的指针转换。尽管它们可能都有 peel()函数，但是你能像剥橘子皮一样给苹果剥皮吗?

由这种转型导致的错误，比较难以理解和改正。如果有必要使用依赖于系统的转型操作，则推荐的做法是使用 reinterpret_cast 而不是 C 风格的转型，这样当出现问题时，会更容易从源代码中找出原因。

19.8.4 关于 explicit 转换构造函数的更多说明

2.12 节中讲解过 explicit 转换构造函数，下面是使用它的另一个例子。

假设出于某个理由，需要编写一个自己的 String 类[①]，则可能会编写出带有如下几个构造函数的一个类定义：

① 来自 Sgt. Major 的声音：“QString 还不够吗？”

```
class String {
  String();                  // Creates an empty string of length 0
  String(const char* str); // Converts a char array to a String
  explicit String(int n);  // Creates length n string, filled
                             // with spaces
  ... other member functions - but not constructors ...
};
```

描述 String 对象 str1 和 str2 会得到如下的客户代码行：

```
...
String str1, str2;         // construct two empty strings
str1 = "A";
str2 = 'A';
...
```

如果让这两个 String 对象都只包含一个字符'A'，使 str1 和 str2 非常相似，这样做没有什么不合法的。但是，根据实现转换构造函数方式的不同(即如何处理终止的 null 字符)，str1 可能会包含字符'A'和 null 字符。对 str2 的赋值可能是另外一种完全不同的情况。如果移除 explicit 关键字，则 str2 将成为一个长度为 65 的 String 对象，其值为空。这是因为，字符'A'将被提升为 int 值 65('A'的 ASCII 码值)，这样就会隐式地调用第三个构造函数。

为了避免这种误解，一种好的做法是将第三个构造函数指定成 explicit。这样做会导致 str2 赋值产生一个编译错误，即第三个构造函数不能被隐式地调用，只能显式地调用。因此，String 类就无法处理这一行代码，编译器会报告错误。

19.9 重载特殊的运算符

本节探讨如何将运算符重载作为“句法糖果”的一种形式，这样就能够将复杂对象当成简单类型，即具有方括号的数组和带有圆括号的函数。

19.9.1 转换运算符

2.12 节中讲过，利用转换构造函数，可以在编译器的类型系统中实现类型转换功能。这样就使得从一种已有类型转换成一种新类型成为可能。为了定义反方向的转换，需要使用转换运算符。5.3 节中讲解过如何定义非简单类型上的运算符的行为。转换运算符使用与类型转换重载相同的机制。

转换运算符看起来与成员函数大不相同。它没有返回类型(甚至没有 void)，也没有任何参数。当使用转换运算符时，会返回一个具有它所命名的类型的对象：在函数体中指定的主对象的转换对象。转换运算符通常是隐式使用的，这样在需要时可以进行自动转换(例如确定函数调用时，见 5.1 节)。

当运行示例 19.13 中的程序时，可以看到它使用了从 Fraction 到 double 和从 Fraction 到 QString 的用户定义转换。

示例 19.13 src/operators/fraction/fraction-operators.cpp

```
#include <QString>
#include <QTextStream>

QTextStream cout(stdout);
```

```
class Fraction {
public:
    Fraction(int n, int d = 1)                                    1
        : m_Numerator(n), m_Denominator(d) {}

    operator double() const {                                     2
        return (double) m_Numerator / m_Denominator;
    }
    operator QString () const {
        return  QString("%1/%2").arg(m_Numerator).arg(m_Denominator);
    }
private:
    int m_Numerator, m_Denominator;
};

QTextStream& operator<< (QTextStream& os, const Fraction& f) {
    os << static_cast<QString> (f);                               3
    return os;
}

int main() {

    Fraction frac(1,3);
    Fraction frac2(4);                                            4
    double d = frac;                                              5
    QString fs = frac;                                            6
    cout << "fs= " << fs << "  d=" << d << endl;
    cout << frac << endl;                                         7
    cout << frac2 << endl;
    return 0;
}
```

1 转换构造函数。

2 转换运算符。

3 显式的转型操作调用转换运算符。

4 转换构造函数调用。

5 调用转换运算符。

6 另一个转换运算符调用。

7 直接调用 operator <<()。

以下是程序的输出。

```
src/operators/fraction> ./fraction
fs= 1/3  d=0.333333
1/3
4/1
src/operators/fraction>
```

19.9.2 下标运算符 operator[]

许多列表和与数组相似的类都提供与数组一致的接口，但还具有更多的功能。下标运算符 operator[]被限制成只使用一个参数。它通常被用来提供对容器中某个元素的访问，如

示例 5.16 所示。示例 19.14 定义的一个容器类使用了下标运算符 operator[]()，它能够防止数组下标越界。

示例 19.14　src/operators/vect1/vect1.h

```
[ . . . . ]
class Vect {
public:
    explicit Vect(int n = 10);
    ~Vect() {
        delete []m_P;
    }
    int& operator[](int i) {                                    1
        assert (i >= 0 && i < m_Size);
        return m_P[i];
    }
    int  ub() const {
        return (m_Size - 1);
    }                                                           2
private:
    int*  m_P;
    int   m_Size;
};

Vect::Vect(int n) : m_Size(n) {
    assert(n > 0);
    m_P = new int[m_Size];
}
[ . . . . ]
```

1 访问 m_P[i]。

2 上界。

示例 19.15 中的客户代码定义了一个 Vect 对象的数组，它提供一种与矩阵类似的结构，其中一个是固定的一维数组，另一个根据 Vect 实现的不同可能是变量或者稀疏矩阵。main() 中还为数字输出提供了一个简单的右对齐方法。

示例 19.15　src/operators/vect1/vect1test.cpp

```
#include "vect1.h"

int main() {
    Vect a(60), b[20];

    b[1][5] = 7;
    cout << " 1 element 5 = "<< b[1][5] << endl;
    for (int i = 0; i <= a.ub(); ++i)
        a[i] = 2 * i + 1;
    for (int i = a.ub(); i >= 0; --i)
        cout << ((a[i] < 100) ? " " : "" )
        << ((a[i] < 10) ? " " : "" )
        << a[i]
        << ((i % 10) ? "  " : "\n");
```

```
    cout << endl;
    cout << "Now try to access an out-of-range index"
    << endl;
    cout << a[62] << endl;
}
```

以下是程序的输出。

```
src/operators/vect1> ./vect1
 1 element 5 = 7
119  117  115  113  111  109  107  105  103  101
 99   97   95   93   91   89   87   85   83   81
 79   77   75   73   71   69   67   65   63   61
 59   57   55   53   51   49   47   45   43   41
 39   37   35   33   31   29   27   25   23   21
 19   17   15   13   11    9    7    5    3    1

Now try to access an out-of-range index
vect1: vect1.h:16: int& Vect::operator[](int):
Assertion `i >= 0 && i < m_Size' failed.
Aborted
src/operators/vect1>
```

19.9.3 函数调用运算符

可以将函数调用运算符 operator()重载成非静态成员函数。它经常被用来提供可调用的接口、迭代器或者一个具有多个索引下标的运算符。它比 operator[]更为灵活，因为可以用不同的签名进行重载。示例 19.16 中，就为 Matrix 类提供了一个多下标运算符。

示例 19.16 src/operators/matrix/matrix.h

```
[ . . . . ]
class Matrix {
public:
    Matrix(int rows, int cols);                          1
    Matrix(const Matrix& mat);                           2
    ~Matrix();
    double& operator()(int i, int j);
    double operator()(int i, int j) const;
    // Some useful Matrix operations
    Matrix& operator=(const Matrix& mat);                3
    Matrix operator+(const Matrix& mat) const;           4
    Matrix operator*(const Matrix& mat) const;           5
    bool operator==(const Matrix& mat) const;
    int getRows() const;
    int getCols() const;
    QString toString() const;
private:
    int m_Rows, m_Cols;
    double  **m_NumArray;
    //Some refactoring utility functions
    void sweepClean();                                   6
    void clone(const Matrix& mat);                       7
```

```
    double rcprod(int row, const Matrix& mat, int col) const;
                                                /* Computes dot product of the
                                                host's row with  mat's col. */
};
[ . . . . ]
```

1 分配单元格并清空。
2 复制构造函数；复制 mat。
3 删除主矩阵内容；复制 mat。
4 矩阵加法。
5 矩阵乘法。
6 清除主矩阵全部单元格的值。
7 用新内存位置保存主矩阵的副本。

示例 19.17 中实现了所需要的多下标运算符的两个版本，一个用于获取 Matrix 的值，另一个用于设置它的值。注意，这个实现中使用了(不受保护的)数组下标用法，它与 C/C++ 数组中的下标用法相同。这一段用于正确地处理底层数组的警告代码，有助于产生安全而可靠的公共接口，它对 Matrix 类使用(具有范围检查的)函数调用下标标注法。

示例 19.17　src/operators/matrix/matrix.cpp

```
[ . . . . ]

double Matrix::operator()(int r, int c) const {
   assert (r >= 0 && r < m_Rows && c >= 0 && c < m_Cols);
   return m_NumArray[r][c];
}

double& Matrix::operator()(int r, int c) {
   assert (r >= 0 && r < m_Rows && c >= 0 && c < m_Cols);
   return m_NumArray[r][c];
}
```

示例 19.18 中给出的构造函数实现，需要知道希望为这个 Matrix 使用多少行和多少列。

示例 19.18　src/operators/matrix/matrix.cpp

```
[ . . . . ]

Matrix:: Matrix(int rows, int cols):m_Rows(rows), m_Cols(cols) {
    m_NumArray = new double*[rows];
    for (int r = 0; r < rows; ++r) {
        m_NumArray[r]  = new double[cols];
        for(int c = 0; c < cols; ++c)
           m_NumArray[r][c]  = 0;
    }
}
```

构造函数在 Matrix 的每一个单元格中为 double 值分配了空间，所以示例 19.19 中实现的析构函数必须删除每一个单元格。如果需要实现其他的成员函数，则需要去除这些删除代码。

示例 19.19 src/operators/matrix/matrix.cpp

```
[ . . . . ]

void Matrix::sweepClean() {
   for (int r = 0; r < m_Rows; ++r)
      delete[] m_NumArray[r] ;
   delete[] m_NumArray;
}

Matrix::~Matrix() {
   sweepClean();
}
```

19.9.3.1 练习：函数调用运算符

完成 Matrix 类的成员函数的实现，并编写全面测试这个类的客户代码。

19.10 运行时类型识别

这一节讲解 dynamic_cast 和 typeid，它们是能够执行运行时类型识别(RTTI)的两个运算符。

基类指针到派生类指针的转换，被称为向下转型(downcasting)，因为从基类到派生类的转型被认为是沿类层次“向下”移动。

当操作这种类型层次时，有时必须将指针“向下转型”为一种更具体的类型。如果没有向下转型，则只能使用指针类型(基类)的接口。用到向下转型的一种常见情形是接受基类指针的函数内部。

RTTI 使得程序员能够安全地将指针和引用从基类型转换成派生类型的对象。

dynamic_cast<*D**>(ptr)具有两个操作数：指针类型 *D**和多态类型 *B**的指针 ptr。如果 *D* 是 *B* 的基类(或者 *B* 与 *D* 具有相同的类)，则 dynamic_cast<D*>(ptr)是一个向上转型(或者根本不是一个转型)，它等价于 static_ cast<D*>(ptr)。但是，如果 ptr 具有类型为 *D* 的对象的地址，其中 *D* 派生自 *B*，则这个运算符返回一个类型为 *D**的向下转型指针，指向同一个对象。如果无法使用转型操作，则会返回一个空指针。

dynamic_cast 执行运行时检查，以判断指针/引用转换是否是有效的。例如，假设是在处理一个 QWidget*的集合。示例 19.20 给出了对 QWidget 集合的操作。进一步假设只对按钮和 spinbox 进行操作，而不理睬其他的窗件。

示例 19.20 src/rtti/dynamic_cast.cpp

```
[ . . . . ]
int processWidget(QWidget* wid) {

    if (wid->inherits("QAbstractSpinBox")) {                    1
        QAbstractSpinBox* qasbp =
            static_cast <QAbstractSpinBox*> (wid);
        qasbp->setAlignment(Qt::AlignHCenter);
    }
    else {
        QAbstractButton* buttonPtr =
```

```
            dynamic_cast<QAbstractButton*>(wid);
        if (buttonPtr) {                                          2
            buttonPtr->click();
            qDebug() << QString("I clicked on the %1 button:")
                .arg(buttonPtr->text());
        }
        return 1;
    }
    return 0;
}
[ . . . . ]
    QVector<QWidget*> widvect;

    widvect.append(new QPushButton("Ok"));
    widvect.append(new QCheckBox("Checked"));
    widvect.append(new QComboBox());
    widvect.append(new QMenuBar());
    widvect.append(new QCheckBox("With Fries"));
    widvect.append(new QPushButton("Nooo!!!!"));
    widvect.append(new QDateTimeEdit());
    widvect.append(new QDoubleSpinBox());
    foreach (QWidget* widpointer, widvect) {
        processWidget(widpointer);
    }
   return 0;
}
```

1 只针对由元对象编译器(moc)处理的 QObject。

2 如果非空，则为有效的 QAbstractButton。

注意

qobject_cast(参见 12.2 节)要比 dynamic_cast 运行得快，但它只能用于 QObject 派生的类型。

就运行时成本而言，dynamic_cast 要昂贵得多，也许是 static_cast 的 10～50 倍。这两种转换是不可互换的操作，它们用在不同的环境中。

19.10.1　typeid 运算符

RTTI 中另一种运算符是 typeid()，它返回关于实参的类型信息。例如：

```
void f(Person& pRef) {
   if(typeid(pRef) == typeid(Student) {
     // pRef is actually a reference to a Student object.
     // Proceed with Student-specific processing.
   }
   else {
     // Nope! The object referred to by pRef is not a Student.
     // Proceed to do whatever alternative stuff is required.
   }
}
```

typeid()返回的 type_info 对象与实参的类型相对应。

如果两个对象具有相同的类型，则它们的 type_info 对象应当等价。可以将 typeid() 运算符用于多态类型或者非多态类型，也可以将它用于基本类型和定制类型。此外，typeid() 的实参可以是类型名称或者对象名称。

以下是 g++ 4.4 中 type_info 类的一种实现方式：

```
class type_info {
 private:
    type_info(const type_info& );
    // cannot be copied by users
    type_info& operator=(const type_info&);
    // implementation-dependent representation
 protected:
    explicit type_info(const char *name);
 public:
    virtual ~type_info();
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    bool before(const type_info& rhs) const;
    const char* name() const;
    // returns a pointer to the name of the type
    // [...]
}
```

19.11 成员选择运算符

有两种形式的成员选择运算符：

- *pointer->memberName*
- *object->memberName*

它们貌似相同，但存在两个重要的不同点：

- 第一个是二元的，第二个是一元的。
- 第一个是全局的且不可重载，第二个是可重载的成员函数。

当用于类定义时，一元 operator->()应返回一个指针，指向其成员名称为 *memberName* 的一个对象。

实现了 operator->的对象，通常被称为智能指针。之所以这样称呼，是因为在程序中它会比常规的指针更“聪明”。例如，QSharedPointer 就是一个维护 QObject 引用计数指针的智能指针。如果删除了最后一个共享的指针，则共享的 QObject 也会被删除。

智能指针的例子包括：

- STL 风格的迭代器。
- QPointer, QSharedDataPointer, QSharedPointer, QWeakPointer, QScopedPointer, QExplicitlySharedDataPointer。
- auto_ptr，STL 智能指针。

这些智能指针都是模板类。

示例 19.21 中给出了 QSharedPointer 的部分定义。

示例 19.21　src/pointers/autoptr/qsharedpointer.h

```
template <class T>
class QSharedPointer {
public:
    QSharedPointer();
    explicit QSharedPointer(T* ptr);
    T& operator*() const;
    T* operator->() const;

    bool isNull() const;
    operator bool() const;
    bool operator!() const;
    // [ ... ]
};
```

如果指针<*T*>的对象被销毁时，它自动设置成 0，则称指针<*T*>为 *T* 类型的 QObject 的监控指针(guarded pointer)。QSharedPointer，QPointer 和 QWeakPointer 同样也提供了这种功能性。以下的代码段给出了智能指针是如何被当作常规指针使用的：

```
[. . .]
QPointer<QIntValidator> val = new QIntValidator(someParent);
val->setRange(20, 60);
[. . .]
```

第二行代码中，val->()返回一个指向新分配的对象的指针，然后被用来访问 setRange()成员函数。

19.12　练习：类型与表达式

1. 假设要求你使用一个编写得很不规范的类库，而且没有办法改进这些类的实现方式(但愿这不是真的)。示例 19.22 中给出了这样一个编写得很差的类样本，还给出了一个使用它的小程序。这个程序中创建了几个对象，并将它们作为 const 引用(隐含的意思是不会改变它们)传递给了一个函数，然后输出算术结果。遗憾的是，由于类编写得很差，运行过程中出现了一些错误。

示例 19.22　src/const/cast/const.cc

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

class Snafu {
public:
    Snafu(int x) : mData(x) {}
    void showSum(Snafu & other) const {
        cout << mData + other.mData << endl;
    }

private:
    int mData;
```

```
};

void foo(const Snafu & myObject1,
         const Snafu & myObject2) {
    // [ . . . ]
    myObject1.showSum(myObject2);
}

int main() {

    Snafu myObject1(12345);
    Snafu myObject2(54321);

    foo(myObject1, myObject2);

}
```

回答下列问题。

a. 这些错误是什么?

b. 对类做哪些修改可改正这些错误?

不巧的是,无法对类进行修改。在不改变类定义的前提下,至少给出修复这个程序的两种方法。哪一个方法更好?为什么?

c. 示例 19.23 是一个尚未完成的类定义,它对每一个实例计算每一个对象的数据被输出的次数。请查看并改正这个程序。

示例 19.23 src/const/cast/const2.cc

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

class Quux {
public:
    Quux(int initializer) :
        mData(initializer), printcounter(0) {}
    void print() const;
    void showprintcounter() const {
      cout << printcounter << endl;
    }

private:
    int mData;
    int printcounter;
};

void Quux::print() const {
    cout << mData << endl;
}

int main() {
    Quux a(45);
    a.print();
    a.print();
```

```
    a.print();
    a.showprintcounter();
    const Quux b(246);
    b.print();
    b.print();
    b.showprintcounter();
    return 0;
}
```

19.13　复习题

1. 语句与表达式有什么不同?
2. 重载运算符与函数有什么不同?
3. 在 C++中引入新类型有哪些方法?
4. 对于数字值，最合适的转型运算符是什么?
5. 向 int 变量赋予一个 double 值时，会发生什么?
6. 对于通过多态层次进行的向下转型操作，最合适的转型运算符是什么?
7. 为什么推荐使用 ANSI 转型操作而不是 C 风格的转型操作?
8. 在什么情况下适合使用 reinterpret_cast?