> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Expressions

Contents

> C++ provides a rich set of operators and defines what these operators do when **applied to operands of built-in type**. It also allows us to **define the meaning** of most of the operators when **applied to operands of class types**.

Expression

- *composed:* **one or more operands**

- *simplest form:* **single literal or variable**
- *more complicated expression:* **an operator and one or more operands**

4.1 Fundamentals

4.1.1 Basic Concepts

- *unary operators*:address-of (**&**) and dereference(*****), **one operand.**
- *binary:* equality(**==**) and multiplication(*****),**two operands.**

Grouping Operators and Operands

- *precedence:* **start with what's operator**
- *assciativity:* **what's the operator's operands**
- *order of evaluation:* **the order to get result**

Operand Conversion

- *define:* **operands** are often **converted from one type to another**
- *promoted:* **small integral type** operands to **larger integral type**.(short,char,bool to int)

Overloaded Operators

- *when:* **operators applied to class types**
- *define:* **define the meaning** to an existing operator
- *wraning:* the meaning of the **operator including the type of operands** and **the result**

Lvalues and Rvalues

- *define:* **lvalues could stand on the left-hand** side of an assignment whereas **rvalues could not.**
- *Rvalue:* use it's **contents**
- *Lvalue:* use it's loacl **memory**
- *wraning:* **different Operators require different operands**,Lvalue or Rvalue.

- **wraning: *decltype(expression)***
    - *the expression is Lvalue:* decltype get the **reference of result type**
    - *the expression is Rvalue:*decltype get the **result type**

4.1.2 Precedence and Associativity

- *compound expression:* **A expression with two or more operators**
- *wraning:* **Precedence and associativity** dermine **how the operands are grouped**.

Parentheses Override Precedence and Associativity

```
cout << (6+3)*(4/2 + 2) << endl;   // 36
cout << 6 + 3*4 /(2+2)  << endl;   // 9
```

When Precedence and Associativity Matter

```
int ia[] = {0,2,4,6,8};
int last = *(ia + 4);
last = *ia + 4;          // *,+,= :precedence
cin >> v1 >> v2;         // left associativity:associativity
```

4.1.3 Order of Evaluation

```
cout << i << i++ << endl;  // undefined
```

Order of Evaluation,Precedence ,and Associativity

- *wraning:* **Order of evalution** is **independent of precedence and associativity**.
- *example:* ***f() + g() * h() + j()***
  - *Precedence guarantees:* **first:***, ***g() * h()***
  - *Associativity guarantees:* the result of f() is added to the product of g() and h() and that the result of that addition is added to the value of j().[1]
  - There are **no gurantees** as to **the order in which these functions are called**.
  - If any of **these functions do affect the same object**, then the expression is in error and has **undefined behavior.**

4.2 Arithmetic Operators

Table 4.1. Arithmetic Operators (Left Associative)

Operator	Function	Use
+	unary plus	+ expr
-	unary minus	- expr
*	multiplication	expr * expr
/	division	expr / expr
%	remainder	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

- *wraing:*
  - *evaluating:* **left to right**
  - *operands:* **Rvalue**
  - *result:* **Rvalue**
- *%:* The operands to **%** must have integral type

```
 21 % -5 =  1
-21 % -8 = -5
```

4.3 Logical and Relational Operators

Table 4.2. Logical and Relational Operators

Associativity	Operator	Function	Use
Right	!	logical NOT	!expr
Left	<	less than	expr < expr
Left	<=	less than or equal	expr <= expr
Left	>	greater than	expr > expr
Left	>=	greater than or equal	expr >= expr
Left	==	equality	expr == expr
Left	!=	inequality	expr != expr
Left	&&	logical AND	expr && expr
Left	||	logical OR	expr || expr

Logical AND and OR operators

- *short-circuit evaluation:*
  - *AND (&&):* **The right sibe is evaluated only if the left side is true**
  - *OR (||):* **The right sibe is evaluated only if the left side is false**

Euqality Tests and the bool Literals

```
if(val) {//true}
if(!val){//false}
```

4.4 Assignment Operators

- *left-hand operand:* **modifiable lvalue**[2]
- *result:* **left-hand**

```
int i = 0;          // initializations, not assignment
const int ci = i;   // initializations, not assignment
1024 = i;           // error,left-hand operand is rvalue

ci = 0;             // error,ci is a const lvalue
```

- *right-hand:*
  - *we can use initializer list:* **{ }**

```
int i;
i = {3.14};       // error,narrowing conversion[窄化转换]
vector<int> vi;
vi = {0,1,2,3,4}; // ok
```

  - *wraning:* when **left-hand** operand is **built-in type**,**{ } contain one value.**

Assignment Is Right Associative

```
int ival,jval;
ival = jval = 0;
```

- *associate:* **right to left**

Assignment Has Low Precedence

```
while( (i = get_value())  != 42){
    //...
}
```

Beware of Confusing Equality and Assignment Operators

```
if(a = b){/*...*/}
if(a == b){/*...*/}
```

Compound Assignment Operators

- a = a ***op*** b:
  - +=、-=、*=、/=、%=
  - <<=、>>=、&=、^=、|=
- *code-look:*

```
sum += 1; // sum = sum + 1;
```

4.5 Increment and Decrement Operators

```
int i = 0,j;
//case 1:j = ++i; result:j = 1,i = 1
//case 2:j = i++; result:j = 0,i = 1
//use the postfix operators(用前置++版本，除非必须使用后置的情况下才用后置)
```

```
cout << *iter++ << endl;
// 1.cout << iter << endl;
// 2.iter += 1;
```

4.6 The Member Access Operators

- *dot operator:* **.**
- *arrow operator:* **->**

```
string s1 = "a string",*p = &s1;
auto n = s1.size();  //1 direct
n = (*p).size();     //2 indirect
n = p->size();       //3 indirect
```

- *wraing:* allow operator **require a pointer operand,result is lvalue.**

4.7 The Conditional Operator

- cond **?** expr1 **:** expr2

Nesting Conditional Operations

```
finalgrade = (grade > 90) ? "high pass"
                          : (grade < 60) ? "fail" : "pass";
```

4.8 The Bitwise Operators

Operator	Function	Use
~	bitwise NOT	~expr
<<	left shift	expr1 << expr2
>>	right shift	expr1 >> expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
|	bitwise OR	expr1 | expr2

- *wraing:* we should **using unsigned type** with the bitwise operators

Bitwise Shift Operators

- right-hand operand must more than 0

<< Operator

- *wraning:* **lower than** the **arithmetic** operators but **higher than** the **relational, assignment, and conditional operators**.

4.9 The sizeof Operator

- *return type:* **size_t** ， the size of the type
- two forms:
  - **sizeof (type)**
  - **sizeof expr**
- *wraning:* it does not evaluate its operand
- *use for member of class:* **sizeof classname::member (C++11)**

4.10 Comma Operator(，)

- *evaluate order:* **left to right**
- *result:* **return right expression**

4.11 Type Conversions

- *implicit conversion:* for example : **3.14** + **3**

- double + **int** = > double + **double**

4.1.1 The Arithmetic Conversions

Integral Promotions

- *promoted int or unsigned int:* **bool 、char 、signed char、 unsigned char 、short 、unsigned short**

4.1.2 Other Implicit Conversions

```
int ia[10];
int *p = ia;   // conver ia to a pointer to the first element: &ia[0]
```

- **The IO library defines a conversion from istream to bool.**

```
while(cin >> s)
{
    // ...
}
// The resulting bool value depends on the state of the stream.
```

4.11.3 Explicit Conversions

Named Casts

- ***cast-name*<*type*>*(expression)***
  - *type:* **the target type to conver**
    - if type is a reference,the result is an lvalue.
  - *expression:* **the value to be cast**
  - cast-name: **static_cast、dynamic_cast、const_cast、reinterpret_cast**

static_cast

- cannot use const

const_cast

- use to cast away the const[3]

reinterpret_cast

- *use:*a low-level reinterpretation of the bit pattern of its operands.

```
int *p = reinterpret_cast<int*>(0xffffffff);
```

Old-Style Casts

- two forms:
  - **type (expr); // function style**
  - **(type) expr; // C style**

4.12 Operator Precedence Table

Table 4.4. Operator Precedence

	Associativity and Operator		Function	Use	See Page
1	L	::	global scope	::name	286
	L	::	class scope	class::name	88
	L	::	namespace scope	namespace::name	82
2	L	.	member selectors	object.member	23
	L	->	member selectors	pointer->member	110
	L	[]	subscript	expr[expr]	116
	L	()	function call	name(expr_list)	23
	L	()	type construction	type(expr_list)	164
3	R	++	postfix increment	lvalue++	147
	R	--	postfix decrement	lvalue--	147
	R	typeid	type ID	typeid(type)	826
	R	typeid	run-time type ID	typeid(expr)	826
	R	explicit cast	type conversion	*cast_name*<type>(expr)	162
4	R	++	prefix increment	++lvalue	147
	R	--	prefix decrement	--lvalue	147
	R	~	bitwise NOT	~expr	152
	R	!	logical NOT	!expr	141
	R	-	unary minus	-expr	140
	R	+	unary plus	+expr	140
	R	*	dereference	*expr	53
	R	&	address-of	&lvalue	52
	R	()	type conversion	(type) expr	164
	R	sizeof	size of object	sizeof expr	156
	R	sizeof	size of type	sizeof(type)	156
	R	sizeof...	size of parameter pack	sizeof...(name)	700
	R	new	allocate object	new type	458
	R	new[]	allocate array	new type[size]	458
	R	delete	deallocate object	delete expr	460
	R	delete[]	deallocate array	delete[] expr	460
	R	noexcept	can expr throw	noexcept(expr)	780

5	L	->*	ptr to member select	ptr->*ptr_to_member	837
	L	.*	ptr to member select	obj.*ptr_to_member	837
6	L	*	multiply	expr * expr	139
	L	/	divide	expr / expr	139
	L	%	modulo (remainder)	expr % expr	139
7	L	+	add	expr + expr	139
	L	-	subtract	expr - expr	139
8	L	<<	bitwise shift left	expr << expr	152
	L	>>	bitwise shift right	expr >> expr	152
9	L	<	less than	expr < expr	141
	L	<=	less than or equal	expr <= expr	141
	L	>	greater than	expr > expr	141
	L	>=	greater than or equal	expr >= expr	141
10	L	==	equality	expr == expr	141
	L	!=	inequality	expr != expr	141
11	L	&	bitwise AND	expr & expr	152
12	L	^	bitwise XOR	expr ^ expr	152
13	L	|	bitwise OR	expr | expr	152
14	L	&&	logical AND	expr && expr	141
15	L	||	logical OR	expr || expr	141
16	R	?:	conditional	expr ? expr : expr	151
17	R	=	assignment	lvalue = expr	144
	R	*=, /=, %=,	compound assign	lvalue += expr, etc.	144
	R	+=, -=,			144
	R	<<=, >>=,			144
	R	&=, |=, ^=			144
18	R	throw	throw exception	throw expr	193
19	L	,	comma	expr , expr	157

1. 三部分相加↩

2. 可修改的左值↩

3. 只用于 const 相关转换（例如去除const属性）↩