【C++】Day twelve

Class	C++
Date	@November 30, 2021
Material	
Series Number	
Summary	

【Ch4】Expressions

4.1 Fundamentals

4.1.1 Basic Concepts

There are both unary operators and binary operators.

- Unary Operators: Such as address-of(`&`) and dereference(`*`), act on one operand.
- Binary operators: Such as euqality (`==`) and multiplication(`*`), act on two operands.

Some symbols, such as *, are used as both a unary(dereference) and a binary(multiplication) operator. The context in which a symbol is used determines whether the symbol represetns a unary or binary operator.

Operand Conversions

As part of evaluating an expression, operands are often converted from one type to another.

For example, the binary operators usually expect operands with the same type. These operators can be used on operands with differing types so long as the operands can be converted to a common type.

Overloaded Operators

The language defines what the operators mean when applied to built-in and compound types. We can also define what most operators mean when applied to class types.

Because such definitions give an alternative meaning to an existing operator symbol, we refer to them as overloaded operators.

When we use an overloaded operator, the meaning of the operator-including the type of its operands and the result-depend on how the operator is defined.

However, the number of operands and the precednece and the associativity of the operator cannot be changed.

Lvalues and Rvalues

Every expression in C++ is either an rvalue or an lvalue.

In C++, an lvalue expression yields an object or a function. However, some lvalues, such as const objects, may not be the left-hand operand of an assignment. Moreoever, some expression yield objects but return them as rvalues, not lavlues.

When we use an objet as an rvalue, we use the object's value(its contents). When we use an objects as an lvalue, we use the object's identity(its location in memeory).

4.1.2 Precedence and Associativity

An expression with two or more operators is a compound expression. Evaluating a compound expression involves grouping the operands to the operators. Precedence and associativity determine how the operands are grouped.

Operands of operators with higher precedence group more tightly than operands of operators at lower precednece. Associativity determines how to group operands with the same precedence.

4.1.3 Order of Evaluation

Precedence specifies how the operands are grouped. It says nothing about the order in which the operands are evaluated. In most cases, the order is largely unspecified.

For operators that do not specify evalution order, it is an error for an expression to refer to and chagne the same object. Expressions that do so have undefined behaviour. As a

simple example, the << operator makes no guarantees about when or how its operands are evaluated. As a result, the following touput expression is undefined:

```
int i = 0;
cout << i << " " << ++i << endl; //undefined
```

There are four operators that guarantee the order of evaluation: AND(&&), OR(||), the conditional operator(?:), and the comma(,) operator.

Advice: Managing Compound Expressions

1. When in doubt, parenthesize expressions to force the grouping that the logia of your program requires
2. If you change the value of an operand, don't use that operand elsewhere in the same expression

   Except for the case:

   ```
   *++ptr; //ptr will be added one and then dereferenced
   ```

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

The table groups the operators by their precedence. The unary arithmetic operators have higher precedence than the multiplication and division operators, which in turn have higher precedence than the binary addition and sbutraction operators.

The modulus operator % returns the result of `m % n` that has the same sign as m.

```
-21 % 4 = -1;
-21 % -5 = -1
21 % -5 = 1
```