Module03-03
C++ 语言基础：表达式

语言基础 - 表达式

- 表达式 (Expressions)
  - 左值与右值 (lvalues and rvalues)
  - 类型转换 (Type Conversions)
  - 常量表达式 (Constant Expressions)
  - volatile 限定符
  - 操作符 (Operators)
  - 表达式求值 (Expression Evalution)

表达式 - 关于表达式

- 表达式是什么
  - 表达式是用一系列的操作符和操作数来具体描述一次计算，一个表达式会产生一个值或带来一些副作用。
  - 示例

```
'a';     12; // 文字常量也是表达式
int a = 0;
a;  // 名字（对象/变量）也是表达式
++a;     // 一元操作符的表达式
a + 2;  // 二元操作符的表达式
a > 3 ? 8 : 2;  // 三元操作符的表达式
(a > 9 || a < 3);    // 复合表达式
int n = 0, i;
(i = n, ++n, i);     // 逗号表达式
int* np = new int[sizeof(long)]; // new和sizeof操作符的表达式
```

表达式 - 左值与右值

- 左值 (lvalue)
  - 左值是产生对象引用 (包括指针) 的表达式，简单而言，左值就是能出现在赋值操作符左边的对象或表达式；
- 右值 (rvalue)
  - 右值是产生对象的表达式，相对于左值，右值是只能出现在赋值操作符右侧的对象或表达式。
- 关于左值与右值的一些规则：
  - 数组是左值，但地址是右值
  - 语言内建的数组下标 [] 、去引用 * 、赋值 (=, +=, 等) 、前置 ++ 、—操作符产生左值，其它操作符产生右值
  - 返回对象引用的函数产生左值，其它情况产生右值
  - 左值可以隐式的转换为右值，反之则不可

表达式 - 左值与右值

- 左值与右值的一些示例

```
int& leftVal(int& n) { return n; }
int rightVal() { return 12; }

int main() {
    int ar[8];
    ar;  // 左值
    ar[0];  // 左值
    &ar[0]; // 右值
    int* p = ar;        p;  // p左值
    "Tiger"; 12;  'k'; // 文字常量为右值
    const int n = 0;    n;  // 常量n为右值
    int(9); // 右值
    int a = 8;
    rightVal();     // 右值
    leftVal(a); // 左值
    ++a;     // 左值
    a++;     // 右值
    ar[1] = ++a;     // 左值可以隐式转为右值
}
```

表达式 - 类型转换

- 算术类型
  - 可以描述为基本类型，如布尔型、字符型、整型和浮点型
- 关于类型转换
  - 算术表达式的二元操作符两边左、右操作数的类型须保持一致
  - 在操作符两边的操作数类型不一致的情况下，通常会通过下面操作来保持类型的一致性：
    - 算术类型的自动提升
    - 类型的隐式转换
    - 显式的类型转换

表达式 - 类型转换

- 类型提升：
  - 从小的算术类型向较大的算术类型自动转换，如 short 向 int 转换
  - 算术类型的类型提升
    - 如果 int 类型能完整的容纳较小的整型类型的右值，将较小整型类型的右值转换成 int 型，否则转换成 unsigned int （所谓小的整型类型指： char, unsigned char, short, unsigned short ）
    - wchar_t 类型、枚举类型通常会转换成： int, unsigned int, long, unsigned long
    - bool 型的右值，可以转换成 int ， true 为 1 ， false 为 0
    - float 可以转换为 double

表达式 - 类型转换

- 类型转换的规则及次序：
  - 如果其中一个操作数是 long double，则其它操作数也转换成 long double
  - 如果其中一个操作数是 double，则其它操作数也转换成 double
  - 如果其中一个操作数是 float，则其它操作数也转换成 float
  - 接下来是整型类型的自动提升
  - 在自动提升后，如果一个操作数是 unsigned long，其它操作数也转换成 unsigned long
  - 如果一个操作数是 long，另一个是 unsigned int，则
    - 如果 unsigned int 型变量的值能完整被 long 容纳，则转换成 long
    - 否则，两个操作数均转换成 unsigned long

表达式 - 类型转换

- 类型转换的规则及次序（续）：
  - 如果一个操作数是 long，其它操作数也转换成 long
  - 如果一个操作数是 unsigned （统指无符号数），其它的操作数也转换成无符号数
  - 最后，操作数都为 int
- 其它类型向 bool 型转换：
  - 任意类型向 bool 型的隐式转换，各类型的非 0 值对象转换成 ture， 0 值对象转换成 false

表达式 - 类型转换

- 显式类型转换：
  - (Type) expr 或 Type (expr)
    - 旧式的显式类型转换，前者是 C 风格的、后者是 C++ 早期的风格
  - static_cast<Type>(expr)
    - 一般用于基本类型之间或枚举类型与整型之间、 void* 向确切类型指针之间的转换
  - dynamic_cast<Type>(expr)
    - 一般用于将基类的指针转换成派生类的指针
  - const_cast<Type>(expr)
    - 用于将 const 指针转换为非 const 对象，或相反的操作
  - reinterpret_cast<Type>(expr)
    - 一种有潜在危险的转换。如将一个地址转换成一个整型数。

表达式 - 类型转换

- 类型转换的结果：
  - 较大类型的整型向较小类型的整型转换，如 int 型向 char 型转换，可能出现值溢出的现象，结果是未定义的
  - 浮点型向整型转换：
    - 小数部分将被丢弃（注意：不是舍入）
    - 整数部分也有可能出现值溢出的现象，结果是未定义的

表达式 - 常量表达式

- const 限定符
  - const 限定的类型，其对象的值不可修改
  - const 指针： T* const
  - const 引用： const T&
- const 指针的书写格式：
  - const 出现在最左边位置，限定其随后相邻的类型
    - const char* 限定的是 char（不是 char*）
  - const 不是出现在最左边的位置，则限定其左侧相邻的类型
    - char const * 限定 char
    - char* const 限定 char*
    - const char* const ：指针不可修改，也不能通过该指针修改指向的对象

表达式 - 常量表达式

- 常量表达式的重要性
  - 对象状态的保护
  - 编译期检查

表达式 - 常量表达式

- 示例：

```
const int cn;    // 错误，必须同时初始化
cn = 0; // 错误，必须同时初始化
const int m = 0;
m = 9;  // 错误，m不可更改

const char* cp = "Tiger"; // 注意不是const 指针
// char const* cp = "Tiger"; // 同上效果
cp[2] = 'm';     // 错误，cp指向的对象不可更改
++cp;     // 可以，指针本身不是const变量
int k = 0;
int* const kp = &k;
const int* const ckp = &k; // 指针和其指向的对象均不可通过ckp更改
++kp; // 错误，const 指针不能改变

const int& kr = k;
++kr;    // 错误，不能通过const 引用去改变k的值
++k;     // 没问题
int& jr = 1;     // 错误，非const引用不能引用文字常量
const int& jr2 = 1; // 或int const& jr2 = 1; // OK
int& const jr3 = 1; // 没有对T&类型的const修饰
```

表达式 - volatile 限定符

- volatile 限定修饰符：
  - 用于告诉编译器被其限定的类型的对象不在编译器控制之下，提醒编译器不要对含有该限定符的对象作优化
  - 声明方式同 const 限定符

表达式 - 操作符

- 操作符列表（按优先级的高低排列）

表达式格式	结合性	描述
class::member	从左到右	域操作符，类成员
namespace::name		域操作符，名字空间成员
::name		域操作符，全局名字
object.member	从左到右	访问对象的成员
pointer->member		访问对象的成员
pointer[expr]		下标操作
expr(expr-list)		函数调用
type(expr-list)		对象构造
lvalue++		后置 ++，递增
lvalue--		后置 --，递减
typeid(type)		类型识别

表达式 - 操作符

- 操作符列表（按优先级的高低排列）（续 1 ）

表达式格式	结合性	描述
typeid(expr)	从左到右	运行期类型识别
dinamic_cast<type>(expr)		运行期检查的转换
static_cast<type>(expr)		编译器检查的转换
reinterpret_cast<type>(expr)		不检查的转换
const_cast<type>(expr)		const 转换
sizeof expr	从右到左	对象的大小
sizeof(type)		类型的大小
++lvalue		前置 ++ ，增量
--lvalue		前置 -- ，递减
~expr		求补
!expr		逻辑非

- 操作符列表（按优先级的高低排列）（续 2 ）

表达式格式	结合性	描述
-expr	从右到左	
+expr		
&lvalue		取址
*expr		去引用
new type		创建新对象（分配空间）
new type(expr-list)		创建新对象（分配空间、初始化）
new(expr-list) type		创建新对象（使用空间）
new(expr-list) type(expr-list)		创建新对象（使用空间、初始化）
delete pointer		销毁对象、释放空间
delete[] pointer		销毁数组、释放空间
(type) expr		类型转换

表达式 - 操作符

- 操作符列表（按优先级的高低排列）（续 3 ）

表达式格式	结合性	描述
object.*pointer-to-member	从右到左	成员指针
object->*pointer-to-member		成员指针
expr * expr	从左到右	乘
expr / expr		除
expr % expr		求模
expr + expr		加
expr - expr		减
expr << expr	从左到右	左移
expr >> expr		右移

- 操作符列表（按优先级的高低排列）（续 4 ）

表达式格式	结合性	描述
expr < expr	从左到右	小于
expr <= expr		小于等于
expr > expr		大于
expr >= expr		大于等于
expr == expr		等于
expr != expr		不等于
expr & expr		按位与
expr ^ expr		按位异或
expr | expr		按位或
expr && expr		逻辑与
expr || expr		逻辑或

- 操作符列表（按优先级的高低排列）（续 5 ）

表达式格式	结合性	描述
expr ? expr : expr	从右到左	条件表达式
lvalue = expr		赋值
lvalue *= expr		乘并赋值
lvalue /= expr		除并赋值
lvalue %= expr		模并赋值
lvalue += expr		加并赋值
lvalue -= expr		减并赋值
lvalue <<= expr		左移并赋值
lvalue >>= expr		右移并赋值
lvalue &= expr		与并赋值
lvalue |= expr		或并赋值
lvalue ^= expr		异或并赋值

表达式 - 操作符

- 操作符列表（按优先级的高低排列）（续 6 ）

表达式格式	结合性	描述
`throw expr`	从右到左	抛出（异常或对象）
`expr, expr, ...`	从左到右	逗号表达式

表达式 - 操作符

- 操作符的优先级
  - 见上节：操作符列表的次序

表达式 - 表达式求值

- 副作用 (Side effects)

  任何表达式有可能产生下面副作用中的一个或多个：

  - 访问 volatile 对象
  - 修改一个对象
  - 调用一个标准库函数
  - 调用其它产生副作用的函数

表达式 - 表达式求值

- 序列点 (Sequence Points)
  - 在程序执行期间，有这么一些时间点：已执行的表达式产生的副作用已完毕，未执行的表达式的副作用尚未产生，这些时间点称为序列点
  - 在个序列点之间，编译器在不违背原始语句的语义的前提下，可以自由重排表达式的执行次序：
    - 如： a() + b() + c() 三个表达式的执行顺序是未定的
  - 在多个序列点之间多次修改或在修改后访问同一个标量对象 (scalar object)，结果是未定义的，如：

```
int ar[] = { 12, 4, 5, 89, 23 };
int n = 1;
ar[n] = n++; // n==2? n==1?
```

表达式 - 表达式求值

- 序列点 (Sequence Points)（续）
  - 一些确切的序列点位置
    - 一个完整的表达式的结束处，如：

```
i++; // 是完整的表达式
i++ - 2; // 其中的 i++ 不是完整的表达式，整个语句才是
```

    - 函数的所有参数已求值完毕，但尚未调用函数时
    - 在获取（拷贝）函数的返回值，且没有对被调用的函数体外的任何表达式求值之前
    - 下面表达式的 expr1 求值结束后（对基本类型而言）
      - `expr1 && expr2`
      - `expr1 || expr2`
      - `expr1 ? expr2 : expr3`
      - `expr1 , expr2` // 逗号表达式

表达式 - 表达式求值

- 短路求值 (Short-Circuit Evaluation)
  - 逻辑运算操作符 && 和 || 有短路求值的特性，即如果操作符左边的表达式结果决定了整个表达式的结果，则操作符右边的表达式将不会求值，如：

```
if (false && 0 < n)        // 0 < n 将不会被求值
if (true || m < 9)         // m < 9 将不会被求值
```

表达式 - Bjarne's Advices

- Bjarne's Advices
  - 避免过于复杂的表达式
  - 如果不确定操作符的优先级，加 ()
  - 尽量避免显式类型转换
  - 如果一定要使用类型转换，用 C++ 风格的 cast
  - 避免带有未定义求值次序的表达式