



C++ 基础

第 4 章：表达式

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者





目录



1. 表达式基础



2. 表达式详述



3. C++ 17 对表达式的求值顺序限定



表达式基础——引入

- 表达式由一到多个操作数组成，可以求值并（通常会）返回求值结果

- 最基本的表达式：变量、字面值
- 通常来说，表达式会包含操作符（运算符）
- 操作符的特性

- 接收几个操作数：一元、二元、三元
- 操作数的类型——类型转换
- 操作数是左值还是右值
- 结果的类型

- 结果是左值还是右值 https://en.cppreference.com/w/cpp/language/operator_precedence 处于同一个优先级的operator都有相同的associativity

- 优先级与结合性（cpp-reference），可以通过小括号来改变运算顺序

- 操作符的重载——不改变接收操作数的个数、优先级与结合性

- 操作数求值顺序的不确定性

右侧不同编译器输出的结果不同，原因是你写出了一个不安全的程序

warning: multiple unsequenced modifications to 'x'

原因：为了提高计算效率，c++和c可能会采取乱序执行的方式来执行代码

```
void fun(int p1, int p2)
{
    std::cout << p1 << ' ' << p2 << '\n';
}

int main()
{
    int x = 0;
    fun(x = x + 1, x = x + 1);
}
```



表达式基础——左值与右值

- 传统的左值与右值划分

把可以放在等号左边的称为左值。不能放在等号左边的称为右值。

- 来源于 C 语言：左值可能放在等号左边；右值只能放在等号右边

- 在 C++ 中，左值也不一定能放在等号左边；右值也可能放在等号左边

```
const int x = 3; // enmutable lvalue  
// x是一个左值
```

```
int() // prvalue  
  
struct Str {};  
int main()  
{ Str x = Str();  
  Str() = Str(); }
```

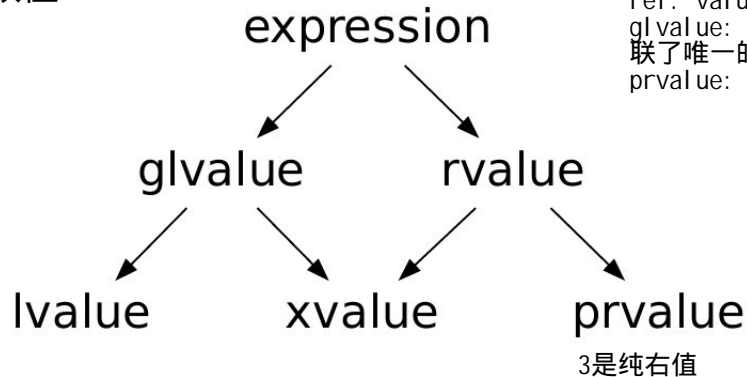
- 所有的划分都是针对表达式的，不是针对对象或数值

- glvalue：标识一个对象、位或函数

- prvalue：用于初始化对象或作为操作数

- xvalue：表示其资源可以被重新使用

将亡值





表达式基础——左值与右值（续）

- 左值与右值的转换

- 左值转换为右值（lvalue to rvalue conversion）

- 临时具体化（Temporary Materialization）prvalue \rightarrow xvalue
为了匹配左值和右值的关系而引入的这个概念

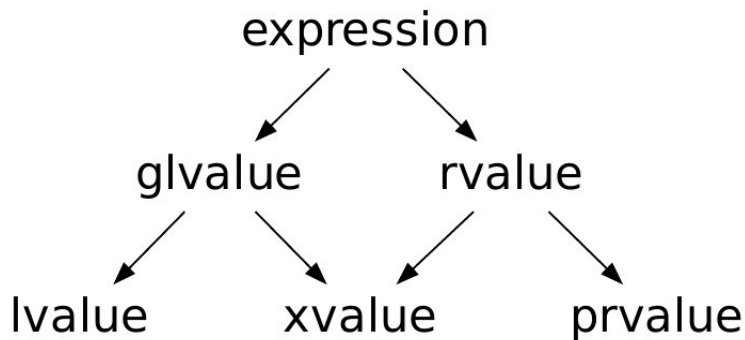
- 再论 decltype

- prvalue \rightarrow type
 - lvalue \rightarrow type&
 - xvalue \rightarrow type&&

```
int x = 3; // x is lvalue
int y = x; // 我们需要使用纯右值来对y进行初始化
y = x;
```

```
void fun(const int& par) {} // 引用必须绑定到
某个具体的对象上
```

```
int main() {
    fun(3); // 3是一个纯右值。这里就涉及到了
    temporary materialization, 把3转化为了xvalue
}
```





表达式基础—类型转换

- 一些操作符要求其操作数具有特定的类型，或者具有相同的类型，此时可能产生类型转换

- 隐式类型转换

- 自动发生

- 实际上是一个（有限长度的）转型序列

- https://en.cppreference.com/w/cpp/language/implicit_conversion

```
void fun(void* par, int t)
{
    if(t == 1) {
        int* ptr = static_cast<int*> (par);
    }
    else if (t == 2) {
        double* ptr = static_cast<double*> (par);
        // ...
    }
}
```

- 显式类型转换 不止下面这五种，但是下面这几种是用得比较多的。c++故意把类型转换做的这么麻烦，因为它尽量不想让人们使用这个功能。

- 显式引入的转换

- `static_cast` 可以在一定程度上把基类转换为派生类，但可能不安全. 比较安全的方法是用 `dynamic_cast` 不能去除常量性，会报错. `const int* ptr; static_cast<int*>(ptr);`

- `const_cast`

- `dynamic_cast` `int x = 3; double y = reinterpret_cast<double>(x); // 编译报错。`

- `reinterpret_cast` 大部分做的是有关指针的转换 `int x = 3; int* ptr = &x; double* ptr2 = reinterpret_cast<double*>(ptr); // 此时ptr2已经是一个未知的值了`

- C 形式的类型转换 `explicit_cast`. 一般在c语言中使用。不建议在c++中使用



表达式详述——算术操作符

- 共分为三个优先级

- + , - (一元) 就是代表正负号

- * , / , % % 表示求余

- + , - (二元)

- 均为左结合的

- 通常来说, 操作数与结果均为算数类型的右值; 但加减法与一元 + 可接收指针 引入+, 实现了强制的类型转换。+不能用于数组, 但是可以用于指针。

- 一元 + 操作符会产生 **integral promotion**

- 整数相除会产生整数, 向 0 取整**

- 求余只能接收**整数类型操作数**, 结果符号与第一个操作数相同

- 满足 $(m / n) * n + m \% n == m$

Source:

```
1 #include <stdio>
2
3 int main()
4 {
5     int a[3] = {1, 2, 3};
6     const auto& x = a;
7 }
```

Insight:

```
1 #include <stdio>
2
3 int main()
4 {
5     int a[3] = {1, 2, 3};
6     int *const &x = +a;
7 }
8
```

Source:

```
1 #include <stdio>
2
3 int main()
4 {
5     int a[3] = {1, 2, 3};
6     const auto& x = a;
7 }
```

Insight:

```
1 #include <stdio>
2
3 int main()
4 {
5     int a[3] = {1, 2, 3};
6     int const (&x)[3] = a;
7 }
```

Source:

```
1 #include <stdio>
2
3 int main()
4 {
5     short x = 3;
6     auto y = x;
7 }
```

Insight:

```
1 #include <stdio>
2
3 int main()
4 {
5     short x = 3;
6     short y = x;
7 }
```

Source:

```
1 #include <stdio>
2
3 int main()
4 {
5     short x = 3;
6     auto y = +x;
7 }
```

Insight:

```
1 #include <stdio>
2
3 int main()
4 {
5     short x = 3;
6     int y = +static_cast<int>(x);
7 }
8
```



表达式详述——逻辑与关系操作符

- 关系操作符接收算术或指针类型操作数；逻辑操作符接收可转换为 bool 值的操作数

- 操作数与结果均为**右值**（结果类型为 bool）

- 除逻辑非外，其它操作符都是左结合的

- 逻辑与、逻辑或具有**短路**特性

- 逻辑与的优先级高于逻辑或**

- 通常来说，不能将多个关系操作符串连

- 不要写出 `val == true` 这样的代码

- Spaceship operator: `<=>`**

- `strong_ordering`

中的`equal`表示二者具有可替换性

- `weak_ordering`

二者在某一方面相等的。但是二者不是相同的。

- `partial_ordering`

增加了个`unordered`. NaN values compare unordered with any other value.

NaN (not a number)

C++20 new feature

如果判断a和b的关系比较费时。就

可以采用这个方法，仅判断一次，

就等到a和b的关系。不用先判断a>b

，再判断a<b

`a <=> b;` // 会返回a和b的关系。

`auto res = (a <=> b);`

`if (res > 0)`

{

}

else if (res < 0)

{

}

else if (res == 0)

{

}

Source:

```
1 #include <stdio>
2 #include <compare>
3
4 int main()
5 {
6     auto res = ((3.0 <=> 5.0));
7 }
```

Insight:

```
1 #include <stdio>
2 #include <compare>
3
4 int main()
5 {
6     std::partial_ordering res = (3.0 <=> 5.0);
7 }
```

```
#include <compare>
#include <cmath>
#include <iostream>
```

```
int main()
{
    double f = 3.0;

    auto res = {sqrt(-1) <=> 5.0};

    std::cout << (res > 0) << std::endl;
    std::cout << (res < 0) << std::endl;
    std::cout << (res == 0) << std::endl;
    std::cout << (res == std::partial_ordering::unordered) << std::endl;
}
```

0 0 0 1



表达式详述——位操作符

- 接收右值，进行位运算，返回右值
- 除取反外，其它运算符均为左结合的
- 注意计算过程中可能会涉及到 integral promotion
- 注意这里没有短路逻辑
- 移位操作在一定情况下等价于乘（除）2 的幂，但速度更快
- 注意整数的符号与位操作符的相关影响
 - integral promotion 会根据整数的符号影响其结果
 - 右移保持符号，但左移不能保证

Source: `1 #include <stdio>`
`2`
`3 int main()`
`4 {`
`5 signed char x = 3;`
`6 signed char y = 5;`
`7 auto z = x & y;`
`8 }`

Insight: `1 #include <stdio>`
`2`
`3 int main()`
`4 {`
`5 signed char x = 3;`
`6 signed char y = 5;`
`7 int z = static_cast<int>(x) & static_cast<int>(y);`
`8 }`
`9`

原因：int 是在硬件系统中相对经常使用的类型。对 char 来做位操作，对硬件来说处理的没有那么好。因此我们通常会把 char 变成 int 的数据类型。然后再做位操作

```
int main()
{
    char x = 0xff; // 11111111
    //1111..1111111111
    //0000..0000000000
    auto y = ~x;
    std::cout << y << std::endl;
}
```

会在不同环境下的结果不同。因为你不知道编译器是解释为了 unsigned 还是 signed



表达式详述——赋值操作符

- 左操作数为可修改左值；右操作数为右值，可以转换为左操作数的类型
- 赋值操作符是右结合的 $x = y = 3$; // 因为赋值操作符是右结合的，所以3会先给y。然后把y=3的求值结果（求值结果是y）给x
- 求值结果为左操作数 $(x=5)=2$; // 1. 把5赋予x，然后返回求值结果x。然后再把2赋予x

- 可以引入**大括号（初始化列表）**以防止收缩转换（narrowing conversion）
`short x;`
`x = { 0x8000000 }; // 如果发生收缩转换会直接报错`
`// 不用大括号只有warning`

- 小心区分 = 与 ==

- 复合赋值运算符

异或操作：1. 有交换律。2. 两个数相同的，异或出来是0
3. 任何东西和0异或都是它本身

```
int x = 2;
int y = 3;
x^=y^=x^=y; // 交换了x和y,
// 优点节省了一个内存空间
// ^= 右结合。所以会先算
// 最右边的两个表达式
// 其实也并不推荐上面的方法。
// 仅用作简单的思维训练
```

// 典型的方法：

```
int tmp;
tmp = x;
x = y;
y = tmp;
```

```
x = 2, y = 3
x^=y    x = 2^3, y = 3
y^=x    x = 2^3, y = 3^2^3 = 3^3^2 = 0^2 = 2
x^=y    x = 2^3^2 = 3, y = 2
```

```
int y = 3;
short x;
x = { y };
std::cout << x << std::endl;
```

```
constexpr int y = 3;
short x;
x = { y };
std::cout << x << std::endl;
```

左侧情况会报错，右侧情况不会报错。因为左侧编译器不知道y运行到此处的值是多少。而右侧程序知道y还是3。不会收缩



表达式详述——自增与自减运算符

- ++; --
- 分前缀与后缀两种情况
- 操作数为左值；前缀时返回左值；后缀时返回右值
返回变化后的值 返回x的原始值，因为x已经加1了，所以返回的是一个临时变量。所以返回的是一个右值
- 建议使用前缀形式 后缀：先构建tmp=x；再x+1，再返回tmp。付出了更多的成本。

```
++++x; //合法的  
(x++)++; //不合法。因为x++返回是右值。
```

x++; //如果你不使用临时变量。那么编译器一般会优化，省去临时变量。但是这个并不是标准，不一定会优化。



表达式详述——其它操作符

- 成员访问操作符：. 与 ->

- `->` 等价于 `(*)`.

```
Str* ptr = &a;
(*ptr).x; //和下面的语句等价
ptr->x;
```
- `.` 的左操作数是左值（或右值），返回左值（或右值 xvalue）
- `->` 的左操作数指针，返回左值

- 条件操作符

- **唯一的三元操作符** `true ? 3 : 5;`
- 接收一个可转换为 `bool` 的表达式与**两个类型相同**的表达式，**只有一个**表达式会被求值
- **如果表达式均是左值，那么就返回左值，否则返回右值**

```
int x = 2;
false ? 1 : x; //表达式返回右值
int y = 2;
false ? x : y; // 表达式返回左值
```
- 右结合


```
int score = 100;
int res = (score > 0) ? 1 : (score == 0) ? 0 : -1;
// 右结合，所以先算后半部分 (score == 0) ? 0 : -1
// 不过这样的代码看起来太复杂了
```



表达式详述——其它操作符（续）

- 逗号操作符 2, 3 // 逗号是个二元操作符，会先求2，再求3。求值的顺序是确定的。这个表达式返回的结果是3。
 - 确保操作数会被从左向右求值
 - 求值结果为右操作数 fun(2, 3); // 这里面逗号并不是一个操作符。这里面对于2还是3哪个先求值是不确定的
 - 左结合
- sizeof 操作符
 - 操作数可以是一个类型或一个表达式 int* ptr = nullptr;
 *ptr; // 非法的，我们不能对一个空指针解引用
 - 并不会实际求值，而是返回相应的尺寸 sizeof(*ptr); // 合法的
- 其它操作符
 - 域解析操作符 ::
 - 函数调用操作符 ()
 - 索引操作符 []
 - 抛出异常操作符 throw
 - ...



C++ 17 对表达式的求值顺序限定

- 以下表达式在 C++17 中，可以确保 e1 会先于 e2 被求值
 - e1[e2] c++之前a会在b之前被执行的情况：
a, b
 - e1.e2 a ? b : c
a && b
a || b
 - e1.*e2
 - e1→*e2
 - e1<<e2 左移右移操作符
 - e1>>e2
 - e2 = e1 / e2 += e1 / e2 *= e1... （赋值及赋值相关的复合运算）
- new Type(e) 会确保 e 会在分配内存之后求值

感谢聆听 !
Thanks for Listening

