《现代C++编程实现》电子版：https://leanpub.com/c01

运算符

YouTube频道： hwdong

博客： hwdong-net.github.io

运算符与表达式

- 运算符是用于对数据执行数学或逻辑运算的特殊符号。

  如：加法运算符 +　　乘法运算符 * 、　　逻辑或运算符 ||

  a+b
  a*b
  a||b

- 运算符用于操作数据，从而构成表达式。

运算符的分类

根据功能和用法，C++ 中的运算符可分为以下几类：

1. 算术运算符：用于数学计算，包括 +、-、*、/ 和 % 等。
2. 关系运算符：用于比较大小，包括 <、<=、>、>=、== 和 !=。
3. 逻辑运算符：用于布尔逻辑运算，包括 &&、|| 和 !。
4. 条件运算符：又称三元运算符，格式为 exp1 ? exp2 : exp3 。

运算符的分类

5. 逗号运算符：用于连接多个表达式，格式为 exp1, exp2 。
6. 位运算符：用于二进制位操作，包括 &、|、^、~、<< 和 >>。
7. 指针和成员访问运算符：包括 *、&、->、.、[] 等。
8. 类型转换运算符：用于类型转换，包括 C 风格的强制类型转换和 C++ 风格的 static_cast 、 dynamic_cast 、
9. 其他运算符：包括 const_cast 和 reinterpret_cast

运算符的分类

根据参与运算的运算数个数，运算符可分为：

1. 一元运算符：只作用于一个操作数，例如 !、++ 和 --。
2. 二元运算符：作用于两个操作数，例如 +、-、* 和 /。

```
int a = 2;
a++;  // a的值从2变为3，相当于a = a + 1
std::cout << a; // 输出3
```

3. 三元运算符只有一个，即条件运算符？：，其格式

exp1 ? exp2 : exp3

如:

a < b ? a : b

- 求3个数中的最小值:

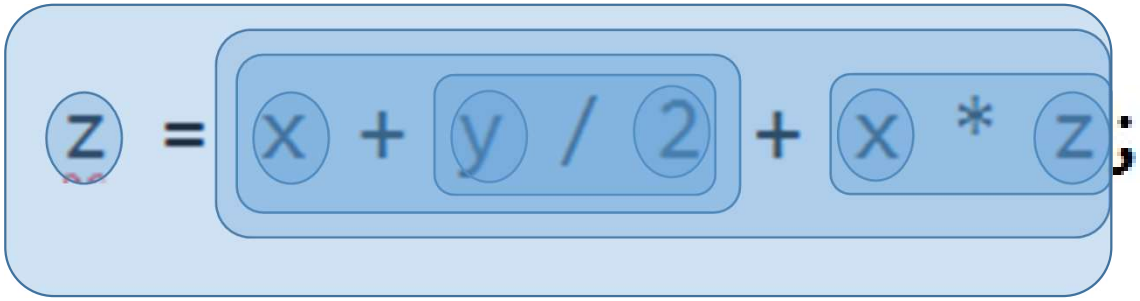
a < b ? (a < c ? a : c) : (b < c ? b : c

- 求3个数中的最小值

```cpp
#include <iostream>
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    int min = a < b ? (a < c ? a : c) : (b < c ? b : c);
    std::cout << a << ", " << b << ", " << c
              << " 这三个数的最小值是: " << min << std::endl;
}
```

表达式

- 运算符对数据(变量和常量)运算构成表达式
- 表达式：0个以上运算符和1个以上运算数
- 最简单的表达式仅由1个变量或常量构成，不含任何运算符
- 表达式都有运算结果，对它们可以继续运算，因此，表达式可以嵌套



先括号里后括号外
先乘除后加减
先左后右

先算术后赋值

优先级

- 运算符优先级决定了多个运算符在没有括号的情况下，表达式中不同运算符的计算顺序。优先级较高的运算符会先执行。

  3 + 4 * 5

- 例如，乘法 (*) 和除法 (/) 运算符的优先级高于加法 (+) 和减法 (-)，所以在没有括号的情况下，表达式 3 + 4 * 5 会先计算4*5，然后再加上3。

结合性

- 决定了当运算符具有相同优先级时，运算符是从左到右（左结合）还是从右到左 （右结合）进行计算。

  3 + 4 + 5

- **左结合**：大多数运算符是左结合的，意味着当多个同优先级的运算符出现在一个表达式中 时，会从左边开始依次计算。

  3 + 4 + 5 会首先计算 3 + 4 ，然后再加上 5，最终结果为 12

结合性

- **右结合**：一些运算符是右结合的，意味着计算顺序是从右到左。例如，赋值运算符（=）就是右结合的。

  a = b = c = 5

- 表达式 a = b = c = 5 会首先执行c=5，然后b=c，然后a=b

运算符优先级与结合性

- C++ 运算符的优先级和结合性可以参考如下网址：

https://en.cppreference.com/w/cpp/language/operator_precedence

运算符优先级与结合性

Precedence	Operator	Description	Associativity
1	`a::b`	Scope resolution	Left-to-right →
2	`a++` `a--`	Suffix/postfix increment and decrement	
	`type``(a)` *`type`*`{a}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`a.b` `a->b`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left ←
	`+a` `-a`	Unary plus and minus	
	`!a` `~a`	Logical NOT and bitwise NOT	
	`(`*`type`*`)a`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` - `new[]`	Dynamic memory allocation	
	`delete` - `delete[]`	Dynamic memory deallocation	

算术运算符

表 3-2 二元算术运算符含义及示例

运算符	含义	例子	结果
+	加	15+2	17
-	减	2-15	-13
*	乘	15*2	30
/	除	15/2	7
%	求余数	15%2	1

表 3-3 一元算术运算符含义及示例

运算符	含义	例子	例子解释
++	后置自增	a++	表达式的值是 a 原先值 2，然后 a 增加 1，变为 3
	前置自增	++a	a 先增加 1，表达式的值为增加后的 a 值 3
--	后置自减	a--	表达式的值是 a 原先值 2，然后 a 减少 1，变为 1
	前置自减	--a	a 先减去 1，表达式的值为减少后的 a 值 1
+	正号	+a	+a 就是 a 自身
-	负号	-a	a 的相反数

```
#include <iostream>
int main(){
    int a=3, b =3;
    std::cout<< "a++的值"<<a++<<", a 的值是"<<a<<std::endl;
    std::cout<< "++b 的值"<<++b<<", b 的值是"<<b<<std::endl;
}
```

输出结果将是：

```
a++的值 3, a 的值是 4
++b 的值 4, b 的值是 4
```

- 前置和后置的区别：

  前置++x，“先运算后结果”

  后置x++，“先结果后运算”

```
int main() {
    int x = 1;
    int a = ++x;    //表达式++x的结果是增加1后的x
    int b = x++;    //表达式x++的结果是原来x的值，x然后增加1
    int c = ++ ++x;
    int d = x + ++x; //++优先于+，但x和++x哪个先计算？
    int e = x++ ++;  //x++的结果是一个临时值，无法继续++
    std::cout << a << std::endl << b << std::endl
              << c << std::endl << d << std::endl;
}
```

error C2105: “++”需要左值

算术运算符需要注意的几个问题

- **溢出问题**

每种类型的变量在内存中占据一定大小的空间，因此其能表示的值的范围也不同。例如，short`类型占用 2 字节（16 位），其中最高位表示正负号，因此它的取值范围是 [-32768, 32767]。如果计算结果超出了该类型的表示范围，就会发生溢出（**overflow**），导致结果不可预期。

算术运算符需要注意的几个问题

- **溢出问题**

运算结果超出了该类型的表示范围，结果是不可预期的，即产生了溢出（overflow）。

```
#include <iostream>
int main() {
    short value = 32767;
    std::cout << value << std::endl;
    value = value + 1;  // 超出 short 类型的范围，结果不可预期!
    std::cout << value << std::endl;
}
```

算术运算符需要注意的几个问题

- **整数相除 /**

除法运算符 / 对两个整数进行运算时，结果总是整数。如果不能整除，小数部分会被截断。例 如：

```
int ival1 = 21 / 6; // ival1 is 3;
int ival2 = 21 / 7; // ival2 is 3;
21.0 / 6;  //3.5
```

算术运算符需要注意的几个问题

- **求余运算 %**

  求余运算符 % 只能用于整数，不能用于浮点数。例如

```
int ival = 42;
double dval = 3.14;
ival % 12; // ok: result is 6
ival % dval; // error: floating-point operand
```

算术运算符需要注意的几个问题

- **求余运算 %**

整数的求余运算 % 的结果符号与被除数相同，即 m % n 的符号与 m`的符号相同。例如：

```
21 % 6;     // 结果是 3
21 % 7;     // 结果是 0
-21 % (-8); // 结果是 -5
21 % (-5);  // 结果是 1
```

警告： 浮点数不能执行取模运算%

算术运算符需要注意的几个问题

- **整数和浮点数混合运算**

当运算中既有整数又有浮点数时，会先将整数转换为浮点数，再进行计算。

```
auto val = 21.0 / 6;
```

算术运算符需要注意的几个问题

如果希望整型数据按照浮点型进行运算，可以进行显式类型转换

21/6

```
static_cast<T>(var);  // 将变量 var 转换为类型 T
```

算术运算符需要注意的几个问题

static_cast<T>

```
#include <iostream>
int main() {
    int a = 3, b = 4;
    std::cout << "int / int = " << a / b << std::endl;
    std::cout << "double / int = " << static_cast<double>(a) / b << std::endl;
    std::cout << "int / double = " << a / static_cast<double>(b) << std::endl;
    std::cout << "double / double = " << static_cast<double>(a) / static_cast<double>
(b) << std::endl;
}
```

自增 ++ 和自减 --

自增 ++ 和自减 -- 运算符分为前置和后置两种形式：

- 前置 ++x：先对 x 加 1，返回的是自增后的 x自身
- 后置 x++：先返回 x 原来的值，然后 x 自增 1

```
#include <iostream>
int main() {
    int a = 3, b = 3;
    std::cout << "a++ 的值: " << a++ << ", a 的新值: " << a << std::endl;
    std::cout << "++b 的值: " << ++b << ", b 的新值: " << b << std::endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main() {
    int x = 1;
    int a = ++x;           // x 先加 1 为 2，a = 2
    int b = x++;           // b = x 的值 2，然后 x 自增为 3
    int c = ++ ++x;        // c 和 x 的值都是 5
    int d = x + ++x;       // x 和 ++x 的计算顺序不确定，可能是 5+6 或 6+6
    // int e = x++ ++;     // 错误：需要左值
    cout << a << endl << b << endl << c << endl << d << endl;
    return 0;
}
```

- 因为++x返回是x自身，所以对它可继续执行++，所以++ ++x的结果仍然是x，而 (++x)++ 也是合法的。
- 表达式 x++ ++ 会出现编译错误，因为x++ 的结果是临时值，不是左值，不能再对其应用++
- 实际编程中，应避免连续使用两个 ++ 运算符，以免造成混淆和错误。

数学计算函数库 cmath

- 对于一些复杂的运算，如求平方根或求指数运算，C++没有相应的运算符，需要借助于C++的函数库，比如从C语言继承来的数学函数库（头文件是cmath）来运算。

```
#include <cmath>
#include <iostream>
int main(){
    double d = 3.5;
    std::cout<<sqrt(3.5)<<std::endl;// 输出 3.5 的平方根
    return 0;
}
```

- pow(a,b)用于求a的b次方

```
#include <cmath>
#include <iostream>
int main(){
    double base = 3.5,exp =6.4;;
    std::cout<<base<<"^"<<exp<<"等于"<<pow(base,exp)std::endl;
    return 0;
}
```

函数名	含义
abs(x)	绝对值函数：返回 x 的绝对值。注意：x 如果是整数，返回的也是整数。如 abs(-2)的结果是整数 2。abs(-2.0)的结果是 2.0
ceil(x)	天花板函数：返回大于等于 x 的最小整数对应的浮点值。如 ceil(-2.5)的结果是-2.0 ，ceil(2.6)的结果是 3
floor(x)	地板函数：返回小于等于 x 的最大整数对应的浮点值。如 ceil(-2.5)的结果是-3.0 ，ceil(2.6)的结果是 2
round(x)	返回最接近 x 的整数对应的浮点值。如 round(-2.3)的结果是-2.0，而 round(-2.6)的结果是-3
pow(x,y)	底数是 x 指数是 y 的指数函数的值。如 pow(2.3,4.5)
exp(x)	底数是自然数，指数是 x 的指数函数的值。exp(2.3)
log(x)	底数是自然数，值是 x 的对数函数的值。
log10(x)	底数是 10，值是 x 的对数函数的值。
sqrt(x)	平方根函数，如 sqrt(2)的值是 1.414..

此外，cmath 还包含了各种三角函数和反三角函数。如 sin()、tan()、acos()、atan()等。

演示三角函数

```cpp
#include <cmath>
#include <iostream>

int main() {
    float angle_deg{ 60.0f };  // 角度
    const float pi{ 3.14159265f };
    const float pi_degrees{ 180.0f };

    float tangent{ std::tan(pi * angle_deg / pi_degrees) };
    float angle = std::atan(tangent);
    float angle_deg2{ angle * pi_degrees / pi };

    std::cout << angle << '\t' << angle_deg2 << '\t' << angle_deg << '\n';
}
```

编写程序，输入一元二次方程的系数a,b,c（假设满足b^2-4ac>=0），输出该方程的2个根.

注：求平方根函数是sqrt(x)形式，其定义在头文件cmath中

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
#include <iostream>
#include <cmath>
int main() {
    double a, b, c;
    std::cin >> a >> b >> c;
    double delta = b*b - 4 * a*c;
    double x1 = (-b + sqrt(delta)) / (2 * a);
    double x2 = (-b + sqrt(delta)) / (2 * a);
    std::cout << x1 << '\t' << x2 << std::endl;
}
```

位运算

- 在计算机中，所有的量都以二进制位的形式存储。假设有两个整数 a = 37 和 b = 22，它们在内存中的二进制表示如下：

```
a = 00100101
b = 00010110
```

显示二进制

- 可以使用 <bitset> 头文件中的 bitset 模板将其转换为二进制表示：

```cpp
#include <iostream>
#include <bitset>
#include <iomanip>

int main() {
    char a{ 37 }, b{ 22 };
    std::cout << std::setw(5) << "a:" << std::bitset<8>(a) << '\n'
              << std::setw(5) << "b:" << std::bitset<8>(b) << '\n';
}
```

- bitset<8> 中的 8 表示以 8 位二进制形式输出。对于 4 字节（32 位）的 int类型，可以使用bitset<32> 来转换为 32 位二进制表示。例如：

```
std::cout << std::bitset<32>(a) << '\n';
```

输出结果为：

```
00000000000000000000000000100101
```

位运算

- 假设 p 和 q 分别是 a 和 b 两个数的对应位置的二进制位

表 3-5 二元位运算符的运算规则

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

```
a = 00100101
b = 00010110
```

因此，`a` 和 `b` 的 `&`、`|` 和 `^` 运算结果如下：

```
a & b = 00000100
a | b = 00110111
a ^ b = 00110011
```

一元位运算符（补运算 `~`、左移 `<<`、右移 `>>`）的规则如下：

- `~` **（补运算）：** 返回一个数的补，即对每一位取反。结果相当于 `-x - 1`。
- `<<` **（左移）：** 各二进位全部左移若干位，高位丢弃，低位补 0。
- `>>` **（右移）：** 各二进位全部右移若干位。对于无符号数，高位补 0；对于负数，高位补 1。

```
b << 2 = 01011000
~b = 11101001
```

```
a = 00100101
b = 00010110
```

```cpp
#include <iostream>
#include <bitset>

int main() {
    char a{ 37 }, b{ 22 };
    std::cout <<"a:"<< '\t'<<(short)a<<'\t'<< std::bitset<8>(a) << '\n'
         << "b:"<< '\t'<<(short)b<<'\t'<< std::bitset<8>(b) << '\n';

    std::cout << "a&b" << '\t'<< (a&b) << '\t'<< std::bitset<8>(a&b) << '\n';
    std::cout << "a|b" << '\t'<< (a|b) << '\t'<< std::bitset<8>(a|b)  << '\n';
    std::cout << "a^b" << '\t'<< (a^b) << '\t'<< std::bitset<8>(a^b)  << '\n';

    std::cout << "~a" << '\t'<< (~a) << '\t'<<std::bitset<8>(~a)<< '\n';
    std::cout << "a<<2" << '\t'<< (a<<2)<< '\t'<<std::bitset<8>(a<<2)  << '\n';
    std::cout << "a>>2" << '\t'<< (a>>2) << '\t'<<std::bitset<8>(a>>2) << '\n';
}
```

```
a:      37      00100101
b:      22      00010110
a&b     4       00000100
a|b     55      00110111
a^b     51      00110011
~a      -38     11011010
a<<2    148     10010100
a>>2    9       00001001
```

赋值运算符

- 最基本的赋值运算符是 =，用于将右边表达式的值赋给左边的变量。例如，a = b 表示将 b 的值复制到变量 a 的内存中。
- 在 C++ 中，基本赋值运算符 = 可以与算术运算符或位运算符组合，形成复合赋值运算符。例如，

```
a += b
```

等价于a = a + b，即将 b 的值加到 a 上。

赋值运算符

运算符	例子	含义
=	a=b	将 b 的值赋值给变量 a
+=	a+=b	相当于 "a = a+b"
-=	a-=b	相当于 "a = a-b"
*=	a*=b	相当于 "a = a*b"
/=	a/=b	相当于 "a = a/b"
%=	a%=b	相当于 "a = a%b"
&=	a&=b	相当于 "a = a&b"
|=	a|=b	相当于 "a = a|b"

赋值运算符

^=	a^=b	相当于 "a = a^b"
<<=	a<<=2	相当于 "a = a<<2"
>>=	a>>=2	相当于 "a = a>>2"

关于赋值运算符的几个说明：

- 右结合性，赋值运算符从右到左的次序计算。如 a=b=c 是先计算 b=c，然后 a=b。
- 多个赋值运算符连用时，类型必须一致。
- 赋值运算符的左边必须是一个左值(可修改的具有独立内存的变量)，不能是文字量或 const 变量，也不能是表达式。如 34=a 或 a+b=c 都是错误的。

```
int a,b,c;
int *p = &a;   // p是int *类型变量，保存a的地址，&a表示获得a的地址
a = b = 3;     //右结合性
a = p;         //错！类型不一致，不能赋值
1024 = a;      //错！左操作数必须是左值
a + b = 10;    //错！左操作数必须是左值
const int ci = c; //const表示不能被修改
ci = c;              // 错！  ci是const变量(对象)，不能被修改
int d = 0; // 对象初始化时的=不是赋值运算符
```

关系运算符

传统比较运算符

表 3-7 关系运算符

运算符	例子	含义
==	a==b	a、b 相等时，返回 true，否则 false
!=	a!=b	a、b 不等时，返回 true，否则 false
<	a < b	a 小于 b 时， 返回 true，否则 false
<=	a <= b	a 小于等于 b 时，返回 true，否则 false
>	a > b	a 大于 b 时， 返回 true，否则 false
>=	a >= b	a 大于等于 b 时，返回 true，否则 false

```cpp
#include <iostream>
using namespace std;
int main() {
    int a = 4, b = 5;
    bool b1 = a < b;     // b1 为 true，因为 4 < 5
    bool b2 = a == b;    // b2 为 false，因为 4 != 5
    // boolalpha 会以 "true" 或 "false" 输出布尔值，而不是 1 或 0
    cout << boolalpha << b1 << '\t' << b2 << endl;
    return 0;
}
```

```
true    false
```

注意事项

1. **右结合性**：关系运算符的结合性从左到右，例如 `a < b < c` 会被解析为 `(a < b) < c`，而不是 `a < (b < c)`，容易引起误解，应避免这种用法。
2. **类型一致性**：比较的两个值通常应为同类型，否则会发生类型转换，可能影响结果。

3.5.2 太空船运算符 `operator<=>`

太空船运算符 `operator<=>` 是 `C++20` 开始引入的一种**比较运算符**，用于统一处理 `<`、`<=`、`>` 和 `>=`。它返回一个**比较类别**（comparison category）对象，而不是 `bool` 值。根据返回的类型，编译器可以自动推导出 `<`、`<=`、`>` 和 `>=` 的结果。例如，`a <=> b` 的返回值可以用于判断 `a` 是否小于、大于或等于 `b`。

C++ 提供了以下几种**标准比较类别**：

- `std::strong_ordering`（强序）
- `std::weak_ordering`（弱序）
- `std::partial_ordering`（部分序）

```cpp
#include <iostream>
#include <compare> // 引入比较支持

int main() {
    int a = 10, b = 20;

    auto result = a <=> b; // 使用太空船运算符进行比较

    if (result < 0) {
        std::cout << "a 小于 b\n";
    } else if (result > 0) {
        std::cout << "a 大于 b\n";
    } else {
        std::cout << "a 等于 b\n";
    }
}
```

太空船运算符与传统比较运算符的区别

特性	**传统比较运算符（<, >, <=, >=）**	**太空船运算符 `<=>`**
运算结果	`bool`	`std::strong_ordering` 等
需要重载的运算符数量	4 个（`<`，`>`，`<=`，`>=`）	1 个（`<=>`）
是否支持 `default` 生成	否	是
编译器是否可自动推导其他比较运算符	否	是

使用 `<=>` 可以减少手动编写多个比较运算符的负担，同时提供更强大的自动推导能力。

- 对于2个浮点数，不能用==判断它们是否相等

```
#include <iostream>
using namespace std;
int main(){
    double d1(100-99.99);
    double d2(10-9.99);
    bool b = d1==d2; // bool b = (d1==d2)
    cout << boolalpha << b << endl;
}
```

输出结果：

```
false
```

```
#include <iostream>
using namespace std;
int main(){
    double d1(100-99.99);
    double d2(10-9.99);
    bool b = d1==d2; // bool b = (d1==d2)
    cout << boolalpha << b << endl;
    cout<<setprecison(17); //浮点数输出格式修改为精度 17 位
    cout<<d1<<endl;
    cout<<d2<<endl;
}
```

输出结果：

```
false
0.010000000000005116
0.009999999999997868
```

- 判断2个浮点数是否相等，通常是看它们差的绝对值是否足够小

```
#include <iostream>
#include<cmath>      // 绝对值函数fabs 函数在 cmath 头文件中
using namespace std;
int main(){
    double d1(100-99.99);
    double d2(10-9.99);
    bool b = fabs(d1-d2)<1e-10;    // 误差是否足够小？
    cout << boolalpha  << b << endl;
    cout<<setprecison(17);
    cout<<d1<<endl;
    cout<<d2<<endl;
}
```

输出结果：

```
true
0.01000000000000516
0.009999999999997868
```

逻辑运算符

表 3-8 逻辑运算符

运算符	含义	运算规则
&&	与	2 个都是 true 或非 0 值时，结果才为 true
||	或	有 1 个是 true 或非 0 值时，结果就为 true
!	非	一元运算符，true 或非 0 值变为 false，false 或 0 值变为 true

```
#include <iostream>
using namespace std;
int main(){
    int a = 4,b = 0;
    cout << boolalpha;
    cout << (a||b) << endl
         << (a&&b) << endl
         << (!a&&b) << endl;
}
```

运行程序的结果为：

```
true
false
false
```

- &&只有当左操作数为真时才去检查右操作数。
- || 只有当左操作数为假时才去检查右操作数。

C++逻辑运算符 !、&&、||分别有等价的关键字 not、and、or。例如 a&&b 也可以写成 a and b。

- 下列逻辑运算的结果分别是?

1) (true && true) || false
2) (false && true) || true
3) (false && true) || false || true
4) (5 > 6 || 4 > 3) && (7 > 8)
5) !(7 > 6 || 3 > 4)

- 下列表达式的结果是什么?

```
unsigned long ul1 = 3, ul2 = 7;
```

(a) `ul1 & ul2`

(b) `ul1 | ul2`

(c) `ul1 && ul2`

(d) `ul1 || ul2`

条件运算符Conditional Operator

- 条件cond为真，结果是expr1的结果，否则结果是expr2的结果
- expr1 和expr2的类型必须相同或能转化为同一种类型

$$cond\ ?\ expr1\ :\ expr2;$$

```
m = a < b ? a : b;
```

如何求a, b, c中的最大值?

```
m = a > b ? (a>c?a:c) : (b>c?b:c) ;
```

```
int a,b,c, m;
cin >> a >> b >> c;
m = a > b ? (a>c?a:c) : (b>c?b:c) ;
cout << m << endl;
```

逗号运算符,

- 逗号运算符(,)也是二元运算符，按照从左到右对2个表达式进行运算，并返回右表达式的值作为整个“逗号表达式”的值。

expr1, expr2

```
#include <iostream>
int main() {
    int a = 1, b = 2, c = 3;
    int x = a, b;    //错:
    int y = (a, b);
    int z = (a, b, c);
    std::cout << z << std::endl;
}
```

变量定义，不是逗号表达式

(a, b)是逗号表达式

逗号运算符,

```
z = (a, b);
z = a, b;
```

先计算逗号表达式(a, b),其结果就是b的结果,然后将这个结果赋值给z

逗号表达式是 "(z = a), b", 因此z的值就是z的值，而整个表达式的结果并没有赋值给其他变量

sizeof运算符

- 返回某类型或表达式结果的占用内存字节数，结果是size_t类型的值
- 两种形式

```
sizeof (type)
sizeof expr
```

```
int i, j;
cout << sizeof(i) << '\t' << sizeof(i+j)
     << '\t' << sizeof(int) << endl;
```

```cpp
#include <iostream>
using namespace std;
int main() {
    cout << "bool: " << sizeof(bool) << "字节" << endl;
    cout << "char: " << sizeof(char) << "字节" << endl;
    cout << "wchar_t: " << sizeof(wchar_t) << "字节" << endl;
    cout << "char16_t: " << sizeof(char16_t) << "字节" << endl;
    cout << "char32_t: " << sizeof(char32_t) << "字节" << endl;
    cout << "short: " << sizeof(short) << "字节" << endl;
    cout << "int: " << sizeof(int) << "字节" << endl;
    cout << "long: " << sizeof(long) << "字节" << endl;
    cout << "long long: " << sizeof(long long) << "字节" << endl;
    cout << "float: " << sizeof(float) << "字节" << endl;
    cout << "double: " << sizeof(double) << "字节" << endl;
    cout << "long double: " << sizeof(long double) << "字节" << endl;
}
```

成员和指针相关运算符

array subscript	`a[b]`
indirection	`*a`
address of	`&a`
member selection	`a.b`
member selection	`a->b`
member selection	`a.*b`
member selection	`a->*b`

其他运算符

function call	`a(...)`
comma	`a, b`
ternary conditional	`a ? b : c`
scope resolution	`a::b`
sizeof	`sizeof(a)`
parameter-pack sizeof	`sizeof...(a)`
alignof	`alignof(T)`
allocate storage	`new T`
allocate storage (array)	`new T[a]`
deallocate storage	`delete a`
deallocate storage (array)	`delete[] a`

关注我

博客： hwdong-net.github.io

Youtube频道

hwdong

@hwdong · 5.01K subscribers · 558 videos

博客：https://hwdong-net.github.io >

youtube.com/c/4kRealSound and 4 more links