C/C++: Lecture 2

Vorobev D.V

11.09.2020

Operators

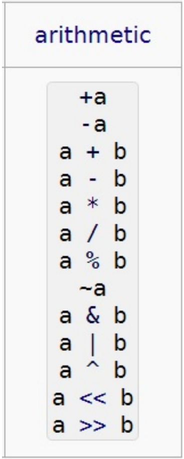
Precedence

It is a property that determines the order of calling operator.

Associativity

It is a property that determines the order of placing brackets.

Arithmetic



arithmetic
+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b

Arithmetic

Right associative

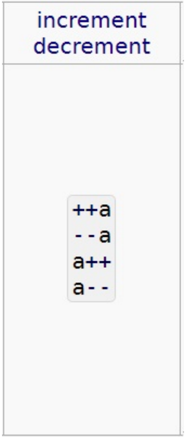
+ (unary) - (unary)

Left associative

+ (binary) - (binary) * / % & | ˆ« »

```
#include <iostream>

int main() {
    // ((7 - 7) - 7) = -7
    std::cout << 7 - 7 - 7;

    // ((24 / 4) / 2) = 3
    std::cout << 24 / 4 / 2;
    return 0;
}
```

Increment / decrement



increment decrement
++a --a a++ a--

Increment / decrement

Prefix

1. increase the value
2. return the value

```
#include <iostream>

int main() {
    int x = 10;
    // 11
    std::cout << ++x;
    // 11
    std::cout << x;
}
```

Increment / decrement

Postfix

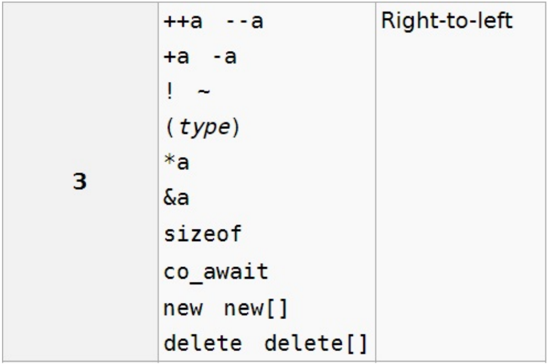
1. copy the value
2. increase the value
3. return the copy

```cpp
#include <iostream>

int main() {
    int x = 10;
    // 10
    std::cout << x++;
    // 11
    std::cout << x;
}
```

Let’s consider the combination of ”- -” and ”-”

```
#include <iostream>

int main() {
    int x = 10;
    std::cout << ---x;
    return 0;
}
```

We can see that - - and - have precedence 3 and both right-associative



3	++a --a +a -a ! ~ (*type*) *a &a sizeof co_await new new[] delete delete[]	Right-to-left

Consequently the order of placing brackets is the following

```
#include <iostream>

int main() {
    int x = 10;
    std::cout << --(-x);
    return 0;
}
```

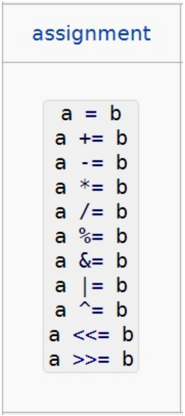
Yes, it raises an error, but the error is the same as it was at the beginning. If we have placed brackets in this way ”-(- -x)” than there would be no error. These two facts prove that the brackets were placed correctly.

Let's consider another example

```cpp
#include <iostream>

int main() {
    int x = 10;
    // everything's fine here
    // output: 9
    std::cout << -(--x);
    return 0;
}
```

```cpp
#include <iostream>

int main() {
    int x = 10;
    // there is no error, thus
    // brackets placing is
    // the following -(x--)
    std::cout << -x--;
    return 0;
}
```

Assignment



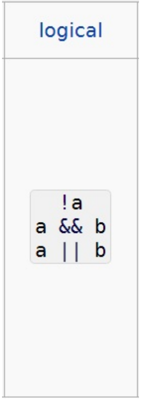
assignment
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b

Assignment

```cpp
#include <iostream>

int main() {
    int x = 10;
    int y = 6;

    x &= y;
    // 2
    std::cout << x;
    return 0;
}
```

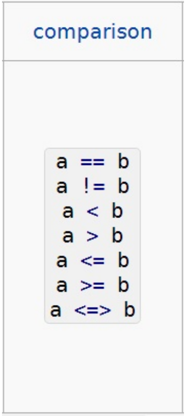
```cpp
#include <iostream>

int main() {
    int x = 10;
    int y = 6;

    x = x & y;
    // 2
    std::cout << x;
    return 0;
}
```

Logical



logical
!a a && b a || b

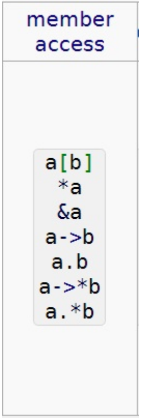
Comparison



comparison

```
a == b
a != b
a < b
a > b
a <= b
a >= b
a <=> b
```

Access



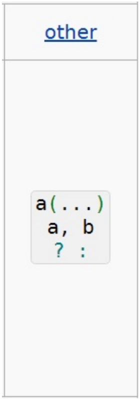
member access
a[b] *a &a a->b a.b a->*b a.*b

Access

```cpp
#include <iostream>

struct A {
    int x = 10;
};

int main() {
    A* p = new A;
    std::cout << p->x;
    std::cout << (*p).x;

    return 0;
}
```

```cpp
#include <iostream>

int main() {
    int* p = new (5);
    // 5
    std::cout << *p;
    int x = 10;
    p = &x;
    // 10
    std::cout << p;

    return 0;
}
```

Other



other
a(...) a, b ? :

Ternary conditional

```
#include <iostream>

int main() {
    int x = 0;
    int y = 1;

    std::cout << a > b ? a : b;
    return 0;
}
```

Comma

```cpp
#include <iostream>

int main() {
    int n = 1;
    int m = (++n, std::cout << "n = " << n << '\n', ++n, 2*n);
    std::cout << "m = " << (++m, m) << '\n';

    return 0;
}
```

sizeof, alignof

- sizeof - yields the size in bytes for the given type
- alignof - yields the alignment in bytes

```
#include <iostream>

struct C {
    char x;
    int y;
};
int main() {
    // 1
    std::cout << alignof(char);
    // 4
    std::cout << alignof(int);
    // an alignment by int
    std::cout << sizeof(C);
    return 0;
}
```

lvalue and rvalue

Definition

- **lvalue** is an **expression** such that we can assign a value
- rvalue is an **expression** that is not lvalue expression

Note

Is is a naive definition. More accurately in the 2nd part of the course.

Function overloading

Definition

It is a definition of at least two functions in the same scope with the same name, different parameter lists and different cv-qualifiers.

Function overloading

Allowable

```
#include <iostream>

void func(double a) {}
void func(int a) {}

int main() {
    return 0;
}
```

Not allowable

```
#include <iostream>

void func(int a) {}
int func(int a) {}

int main() {
    return 0;
}
```

Default arguments

Only the trailing arguments can have default values

Allowable

```cpp
#include <iostream>

void func(int a, int b = 0) {}

int main() {
    return 0;
}
```

Not allowable

```cpp
#include <iostream>

void func(int b = 0, int a) {}

int main() {
    return 0;
}
```

Explicit type conversion

Problem

On the left is a pointer to double, which is assigned a the address of the float value.
We read 8 bytes (ptr to double), but the value is stored in 4 bytes.

C-cast

```
#include <iostream>

int main() {
    float x = 3.1;
    // UB
    double* y = (double*) &x;
    return 0;
}
```

static_cast

```
#include <iostream>

int main() {
    float x = 3.1;
    // CE
    double* y = static_cast<double*>(&x);
    return 0;
}
```

Summary

Use static_cast instead of C-cast. static_cast checks type compatibility.

Control flow statements

if

```
int main() {
    int x = 10;
    if (x) {
        int y = 20;
    }
    int z = 10;
    return 0;
}
```

if, else

```
int main() {
    int x = 10;
    if (x) {
        int y = 20;
    } else {
        int y = 5;
    }
    int z = 10;
    return 0;
}
```

Dangling else

```cpp
#include <iostream>

int main() {
    int x = 0;
    if (1)
        if (1)
            x = 1;
    else
        x = 2;
    return 0;
}
```

```cpp
#include <iostream>

int main() {
    int x = 0;
    if (0)
        if (0)
            x = 1;
    else
        x = 2;
    return 0;
}
```

```cpp
#include <iostream>

int main() {
    int x = 0;
    if (1)
        if (0)
            x = 1;
    else
        x = 2;
    return 0;
}
```

Summary

Use braces and write explicitly

while

```
int main() {
    int x = 0;
    while(x < 1) {x++;}
    return 0;
}
```

do-while

```
int main() {
    int j = 0;
    do {
        j++;
    } while (j < 2);
    return 0;
}
```

for

```cpp
int main() {
    for(size_t x = 0; x < 1; x++) {}
    return 0;
}
```

switch

```
int main() {
    int x = 0;
    switch(x) {
        case 0 : {
            int y = 1;
            break;
        }
        default: {
            int y = 2;
        }
    }
    return 0;
}
```

switch

```
    int main() {
        int x = 0;
        switch(x) {
            case 0 : {
                int y = 1;
            }
            default: {
                int y = 2;
            }
        }
        return 0;
    }
```

break

break

Jump to label of the end of loop

```
int main() {
    int x = 10;
    while( x < 10) {
        break;
    }
    return 0;
}
```

continue

continue

Jump to label of the beginning of the loop

```
int main() {
    int x = 10;
    while( x < 10) {
        continue;
    }
    return 0;
}
```

Return

```
int main() {
    return 0;
}
```