Lecture 3: Expressions

Expressions

- An expression is a sequence of operators and operands that specifies a computation.

- What does an expression do?
  - Produces a result

```
2 + 2    // result: 4
```

  - May cause side effects (state change, I/O, etc.)

```
std::printf("%d", 4); // prints '4' to standard output
```

Arithmetic Operators + - * / %

```
int a = 15, b = 4;

cout << a + b;  // 19  (addition)
cout << a - b;  // 11  (subtraction)
cout << a * b;  // 60  (multiplication)
cout << a / b;  // 3   (integer division)
cout << a % b;  // 3   (remainder)
```

- % works only with integers
- / between integers gives integer result

[source]

Relational (Comparison) == != < > <= >=

```
int x = 10, y = 20;

cout << (x == y); // 0 (false)
cout << (x != y); // 1 (true)
cout << (x < y);  // 1 (true)
cout << (x > y);  // 0 (false)
cout << (x <= 10); // 1 (true)
cout << (y >= 30); // 0 (false)
```

- Result type is `bool`

[source]

Logical && || !

```
bool p = true, q = false;

cout << (p && q); // 0 (AND → both must be true)
cout << (p || q); // 1 (OR → one true enough)
cout << (!p);     // 0 (NOT → reverse)
```

[source]

Assignment = and Compound += -= *= /= %=

```
int a = 5;
a += 3; // a = 8
a -= 2; // a = 6
a *= 4; // a = 24
a /= 6; // a = 4
a %= 3; // a = 1
```

`x += y` means `x = x + y`

[source]

Increment / Decrement ++ --

```
int x = 5;

cout << ++x; // 6  (prefix: increment → use)
cout << x++; // 6  (postfix: use → increment)
cout << x;   // 7

cout << --x; // 6
cout << x--; // 6
cout << x;   // 5
```

[source]

Ternary Operator ?:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
cout << max;  // 20
```

[source]

Comma Operator ,

```
int a = 1, b = 2;
int c = (a++, b++, a + b);
cout << a; // 2
cout << b; // 3
cout << c; // 5  (last expression result is returned)
```

Executes left → right, returns last value

[source]

Bitwise & | ^ ~

```
int a = 6;  // 110
int b = 3;  // 011
cout << (a & b); // 2  (010) AND
cout << (a | b); // 7  (111) OR
cout << (a ^ b); // 5  (101) XOR
cout << (~a);    // bitwise NOT
```

[source]

Bit Shift << >>

```
int x = 5;  // 000101
cout << (x << 1); // 10  (shift bits left ×2)
cout << (x << 2); // 20  (×4)
cout << (x >> 1); // 2   (÷2)
```

- Left shift multiplies by 2
- Right shift divides by 2

[source]

sizeof Operator

```
int x;
double d;
cout << sizeof(x); // 4 (usually)
cout << sizeof(d); // 8 (usually)
cout << sizeof(x + d); // 8 (result of expression is
double)
```

[source]

Address & Dereference & *

```
int x = 10;
int *p = &x;  // store memory address
cout << p;    // prints address
cout << *p;   // 10 (dereference value)
```

[source]

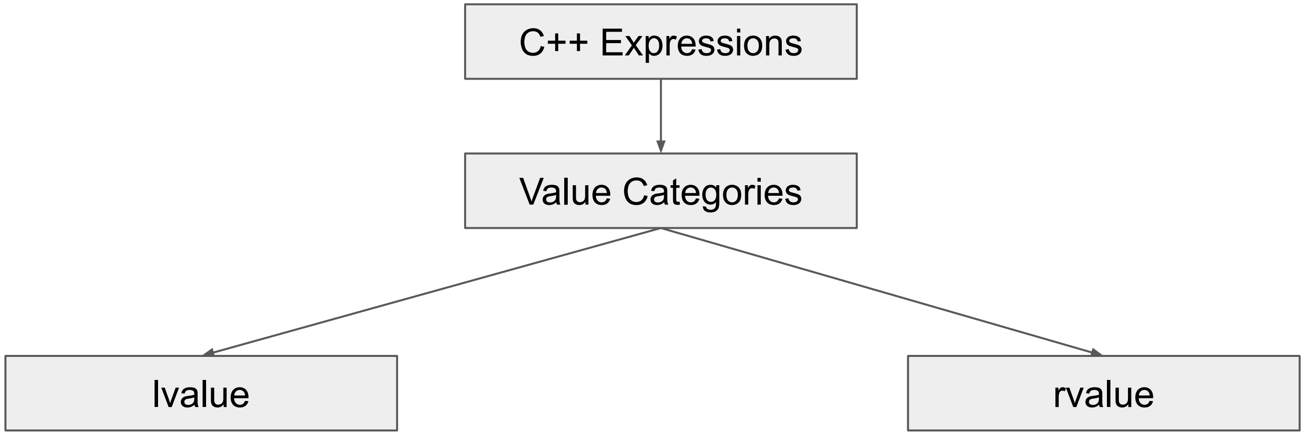
Value Categories in C++

Every C++ expression has:

- **A type** (`int, double`, etc.)
- **A value category** (relation to memory and lifetime)

[source]

Value Categories in C++

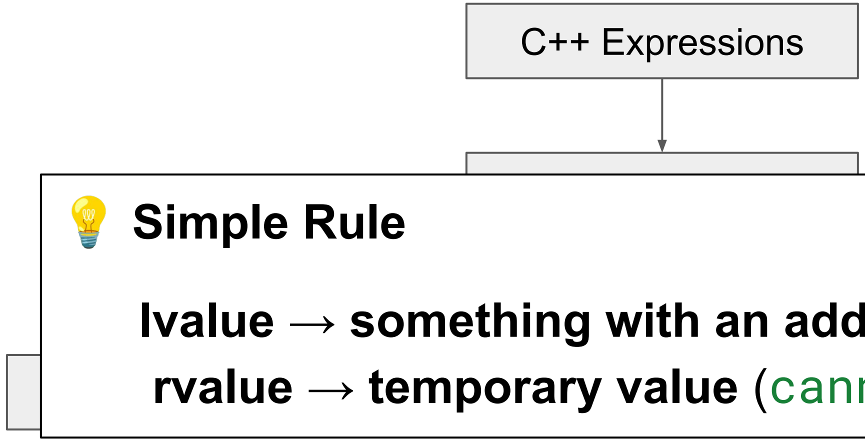


has address, persistent
lvalues can be assigned to

rvalue → **temporary, no address**
rvalues cannot appear on left of =

[source]

Value Categories in C++



💡 **Simple Rule**

lvalue → something with an address (`can use &`)

rvalue → temporary value (`cannot use &`)

has address, persistent
lvalues can be assigned to

rvalue → **temporary, no address**
rvalues cannot appear on left of `=`

[source]

Value Categories in C++

```
int x = 10;             // x is lvalue
int y = x + 5;          // (x + 5) is rvalue

int* p = &x;            // ✅ OK, x is lvalue
// int* q = &(x+5); // ❌ Error, rvalue has no address

x = 20;                 // ✅ allowed (lvalue on left of =)
// (x + 5) = 30;    // ❌ not allowed (rvalue cannot be
assigned)
```

[source]

Statements

- *Statements* are fragments of the C++ program that are executed in sequence.
- The body of any function is a sequence of statements.

[source]

Declaration Statements

```
int x = 10;
const double PI = 3.14;
std::string name = "C++";
```

- Introduces variables, objects, functions
- Best Practice:
  - Initialize immediately to avoid garbage values
  - Prefer **`const`** when value should not change
  - Use meaningful names (`int age` ✅, `int a` ❌)

[source]

Expression Statements

```
x = x + 5;
foo();
std::cout << "Hi";
```

- Performs computation or function call
- Ends with ;
- Avoid
  - if (x = 5) ❌ // Bug! Assignment instead of comparison
- ✅ Use == in conditions

[source]

Compound (Block) Statements

```
{
  int a = 5;
  a++;
}
```

- Groups multiple statements
- Limits variable scope (good practice)
- Keep blocks small and readable

[source]

Selection Statements (if, switch)

```
if (x > 0) {
    std::cout << "Positive";
} else {
    std::cout << "Non-positive";
}
```

- Use {} always

[source]

Selection Statements (if, switch)

```
switch(day) {
  case 1: std::cout<<"Mon"; break;
  default: std::cout<<"Other";
}
```

- Use break in switch to prevent fall-through

[source]

Iteration (Loop) Statements

```
for(int i = 0; i < 3; i++)
    std::cout << i;

while(x--) { }

for(auto v : vec)  // Best practice
    std::cout << v;
```

Prefer range-based for when possible

[source]

Jump Statements

```
break;         // exit loop/switch
continue;      // skip iteration
return 0;      // exit function
goto end;      // jump (avoid!)
end:
```

goto is discouraged — breaks readability

Prefer structured flow

[source]

Try (Exception Handling)

```
try {
  risky();
} catch(const std::exception& e) {
  std::cout << e.what();
}
```

✅ Handles runtime failures safely

✔ Catch by reference

✖ Avoid catching everything silently (catch(...) {}

[source]