

ECE 2400 Computer Systems Programming

Fall 2017

T01 Reading C Programs

School of Electrical and Computer Engineering
Cornell University

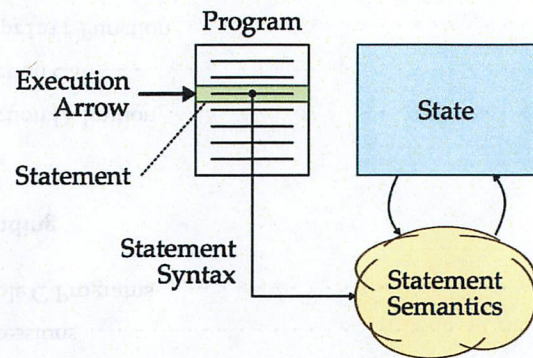
revision: 2017-09-03-20-55

1	Statements, Syntax, Semantics, State	3
2	Variables, Operators, Expressions	4
2.1.	Variables	5
2.2.	Operators	5
2.3.	Expressions	6
2.4.	Simple C Programs	7
3	Name Binding	8
4	Functions	10
4.1.	Function Definition	10
4.2.	Function Call	11
4.3.	The printf Function	14
5	Conditional Statements	15
5.1.	Boolean Operators	15
5.2.	if/else Conditional Statements	17

5.3.	switch/case Conditional Statements	20
6	Iteration Statements	24
6.1.	while Loops	24
6.2.	for Loops	26
7	Syntactic Sugar	28

Before you can learn to write, you must learn to read!
This is true for foreign languages and programming languages.

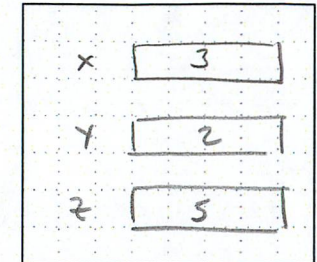
1. Statements, Syntax, Semantics, State



Program	Sequence of statements	It is raining outside. Should I use an umbrella?
Statement	Sentence	It is raining outside.
Syntax	Sentence grammar	punctuation; "I" is a pronoun; "is" uses present tense
Semantics	Sentence meaning	rain is water condensed from the atmosphere, outside means in the outdoors
State	Memory of prior statements	remember that it is raining outside when considering umbrella

An example "English" program

- 1 Create box named x.
- 2 Put value 3 into box named x.
- 3 Create box named y.
- 4 Put value 2 into box named y.
- 5 Create box named z.
- 6 Put $x + y$ into box named z.



stmt	x	y	z
1	—	—	—
2	?		
3	3		
4		?	
5		2	
6			?
			5

row is state before
corresponding statement
number

2. Variables, Operators, Expressions

- A **variable** is a box (in the computer's memory) which stores a value; variables are used for "state"
- An **operator** is a symbol with special semantics to "operate" on variables and values
- An **expression** is a combination of variables, values, and operators which evaluates to a new value

2.1. Variables

- A **variable** is a box (in the computer's memory) which stores a value
- An **identifier** is used to name a variable
- A **type** specifies the kind of values that can be stored in a variable
- A variable declaration statement creates a new variable

TYPE IDENTIFICATION

```

1 int my_variable;
2 int MY_VARIABLE;
3 int variable_0;
4 int 0_variable;
5 int variable$1;

```

semicolon!

illegal

2.2. Operators

- The **assignment operator** (=) "assigns" a new value to a variable
- An **assignment statement** combines the assignment operator with a left-hand side (LHS) and a right-hand side (RHS)
- The **LHS** specifies the variable to change
- The **RHS** specifies the new value

1 int my_variable;

declaration statement

2 my_variable = 42;

assignment statement

- A variable declaration statement and an assignment statement can be combined into a single **initialization statement**

1 int my_variable = 42;

- Other operators are provided for arithmetic functions such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%)

2.3. Expressions

- An **expression** is a combination of variables, values, and operators which evaluates to a new value

```

1 3 + 4 = 7
2 3 + 4 * 2 + 7
3 3 * 4 / 2 * 6

```

$((3+4)*2)+7$ OR $(3+4)*(2+7)$ OR $3+(4*2)+7$
 $(7*2)+7$ $7*9$ $3+8+7$
 $14+7$ 63 19

$((3*4)/2)*6$ OR $(3*4)/(2*6)$ OR $3*(4/2)*6$
 $(12/2)*6$ $12/12$ $3*2*6$
 $6*6$ 1 36

- Operator precedence** is a set of rules describing in what order we should apply a sequence of operators in an expression

Category	Operator	Associativity
Multiplicative	* / %	left to right
Additive	+ -	left to right
Assignment	=	right to left

Be explicit – use parenthesis!

SAME
precedence
LEFT to RIGHTmultiplication
has higher
precedence
than
addition

ASIDE: / : what about 3/2?

?: 4%2 = 0 6%5 = 1

2.4. Simple C Programs

We can compose assignment and initialization statements which use variables, operators, and expressions to create a simple C program.

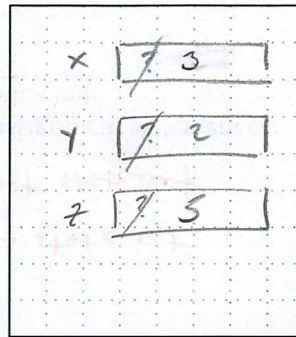
Translating our "English" program into a C program

```

1 int x;
2 x = 3;
3 int y;
4 y = 2;
5 int z;
6 z = x + y;

```

Use ? to indicate undefined value in stack frame diagram



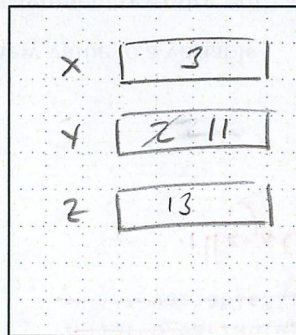
Draw stack frame diagram corresponding to the execution of this program

```

1 int x = 3;
2 int y = 2;
3 int z = x + (y * 5);
4 y = x + (y * x) + y;

```

NOT DECLARING A VARIABLE
JUST AN ASSIGNMENT
STATEMENT.



3. Name Binding

- So far we have only had one variable with a given name

```

1 int x = 1;
2 int x = 2;
3 int y = x;

```

illegal!

- Scope** of a variable is the region of code where it is accessible
- C allows using **blocks** to create new **local scopes**
- Can declare new variables that are only in scope (locally) in the block
- Can declare new variables in the local scope with same name as a variable declared in the parent scope
- Curly braces are used to open and close a block ({})
- Blocks are critical for defining functions, conditional statements, and iteration statements

```

1 int x = 1;
2 {
3   int x = 2;
4   int y = x;
5 }
6 int y = x;

```

Scope

Closing of block ends scope for local variables

- Key Question:** When we use a variable name, what variable declaration is it referring to?
- Name binding** is a set of rules to answer this question by associating a specific variable name to a specific in-scope variable declaration
- C uses **static (lexical) scoping** meaning the name binding happens statically at compile time

Which
Declaration
of x?

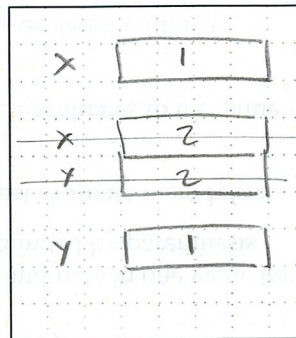
Steps for name binding

1. Draw circle in source code around use of a variable name
2. Determine which variables with that name are in scope
3. Draw line to variable declaration in the inner most enclosing block
4. Draw circle in source code around variable declaration

DO NAME BINDING FIRST!

```

1 int x = 1;
2 {
3   int x = 2;
4   int y = x;
5 }
6 int y = x;
  
```

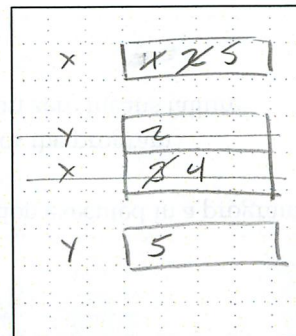


* ACTIVITY *

```

1 int x = 1;
2 {
3   x = 2;
4   {
5     int y = x;
6     int x = 3;
7     x = 4;
8   }
9   x = 5;
10 }
11 int y = x;
  
```

DO NAME BINDING FIRST!



4. Functions

- A **function** gives a name to a parameterized sequence of statements
- A **function definition** describes how a function behaves
- A **function call** is a new kind of expression to execute a function
- All code in this course will be inside functions

4.1. Function Definition

```

1 rtype function_name( ptype0 pname0, ptype1 pname1, ... )
2 {
3   function_body;
4 }
  
```

- A **function name** is a unique identifier for the function
- The **function body** is the parameterized sequence of statements
- The **parameter list** is a list of parameter types and names
- The **return type** is the type of the value returned by the function

```

1 int avg( int x, int y )
2 {
3   int sum = x + y;
4   return sum / 2;
5 }
  
```

```

1 int main()
2 {
3     int a = 10;
4     int b = 20;
5     int c = ( a + b ) / 2;
6     return 0;
7 }

```

Why not return into
A function?

Success!

- Main is special: it is always the first function executed in a program
- Main returns its "value" to the "system"
- The return value is called the **exit status** for the program
- Returning zero means success, greater than zero means failure

4.2. Function Call

```

1 function_name( pvalue0, pvalue1, ... )

```

- To call a function we simply use its name and pass in one value for each parameter in the parameter list surrounded by parenthesis
- If parameters are expressions, then we must evaluate them *before* calling the function
- A function call is itself an expression which evaluates to the value returned by the function
- Function parameters and "local" variables declared within a function are effectively in a new block which is called the function's **stack frame**

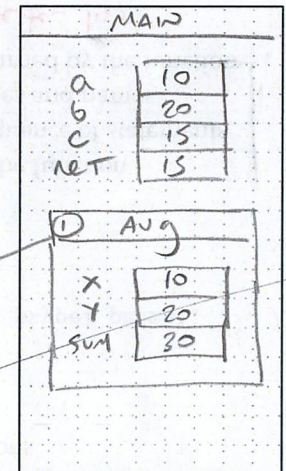
Steps for calling a function

0. Allocate variable for return value on caller's stack frame
1. Draw called function's stack frame w/ parameter boxes
2. Initialize parameters by evaluating expressions in function call
3. Record location of function call
4. Move execution arrow to first statement in called function
5. Evaluate statements inside the called function
6. At return statement, evaluate its argument, update variable in caller
7. Return execution arrow back to where the function was called
8. Erase the called function's frame
9. Use function's return value as value of function call

```

1 int avg( int x, int y )
2 {
3     int sum = x + y;
4     return sum / 2;
5 }
6
7 int main()
8 {
9     int a = 10;
10    int b = 20;
11    int c = avg( a, b );
12    return 0;
13 }

```



* Scoping *

Draw stack frame diagram corresponding to the execution of this program

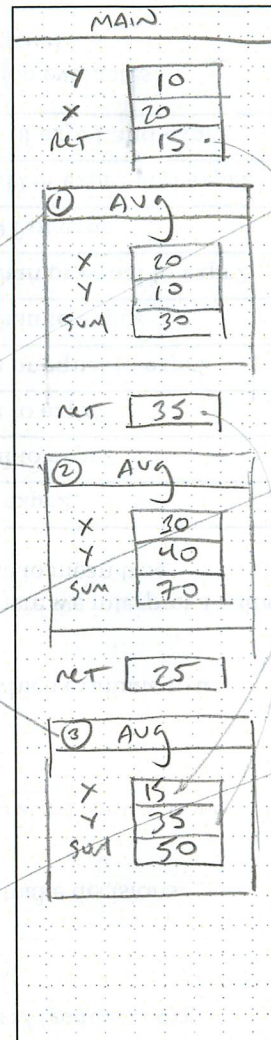
Scope

```

1  int avg( int x, int y )
2  {
3      int sum = x + y;
4      return sum / 2;
5  }
6
7  int main()
8  {
9      int y = 10;
10     int x = 20;
11     x = avg( avg(y,x), avg(30,40) );
12     return 0;
13 }

```

Scoping rules make it clear
that x,y in main have
nothing to do with x,y
in avg!



4.3. The printf Function

The printf function is provided by the C standard library and can be used to print values to the screen. Here is pseudocode for the printf function definition.

```

1  printf( format_string, value0, value1, ... )
2  {
3      substitute value0 into format_string
4      substitute value1 into format_string
5      ...
6      display final format_string on the screen
7  }

```

Here is an example of calling printf.

```

1  #include <stdio.h>
2
3  int avg( int x, int y )
4  {
5      int sum = x + y;
6      return sum / 2;
7  }
8
9  int main()
10 {
11     int a = 10;
12     int b = 20;
13     int c = avg( a, b );
14     printf( "average of %d and %d is %d\n", a, b, c );
15     return 0;
16 }

```

format specifier for int (pointing to %d)
escape sequence for newline (pointing to \n)

- Execute this code via <http://cpp.sh>
- Examine the machine instructions via <https://godbolt.org>

5. Conditional Statements

- **Conditional statements** enable programs to make decisions based on the values of their variables
- Conditional statements enable **non-linear forward control flow**

5.1. Boolean Operators

- **Boolean operators** are used in expressions which evaluate to a "boolean" value (i.e., true or false)
- In C, a "boolean" value is just an integer, where we interpret a value of zero to mean false and any non-zero value to mean true

<code>expr1 == expr2</code>	tests if expr1 is equal to expr2
<code>expr1 != expr2</code>	tests if expr1 is not equal to expr2
<code>expr1 < expr2</code>	tests if expr1 is less than to expr2
<code>expr1 <= expr2</code>	tests if expr1 is less than or equal to expr2
<code>expr1 > expr2</code>	tests if expr1 is greater than to expr2
<code>expr1 >= expr2</code>	tests if expr1 is greater than or equal to expr2
<code>!expr</code>	computes the logical NOT of expr
<code>expr1 && expr2</code>	computes the logical AND of expr1 and expr2
<code>expr1 expr2</code>	computes the logical OR of expr1 and expr2

Using these operators in an expression evaluates to either zero (false) or one (true)

Category	Operator	Associativity
* Unary	!	right to left
Multiplicative	* / %	left to right
Additive	+ -	left to right
* Relational	< <= > >=	left to right
* Equality	== !=	left to right
* Logical AND	&&	left to right
* Logical OR		left to right
Assignment	=	right to left

Mixing boolean operators to create a complex expression

1 7 < 6 && 3 > 1 || !0

Handwritten evaluation:
 ((7 < 6) && (3 > 1)) || (!0)
 (0 && 1) || 1
 0 || 1
 1

Experiment with <http://cpp.sh>:

```

1 #include <stdio.h>
2 int main()
3 {
4     int x = 7 < 6 && 3 > 1 || !0;
5     printf("%d\n", x);
6 }
```

5.2. if/else Conditional Statements

```

1 if ( conditional_expression ) {
2   then_block;
3 }
4 else {
5   else_block;
6 }

```

- A **conditional expression** is an expression which returns a boolean
- The **then block** is executed if the conditional expression is true
- The **else block** is executed if the conditional expression is false

```

1 if ( conditional_expression0 ) {
2   then_block0;
3 }
4 else if ( conditional_expression1 ) {
5   then_block1;
6 }
7 else {
8   else_block;
9 }

```

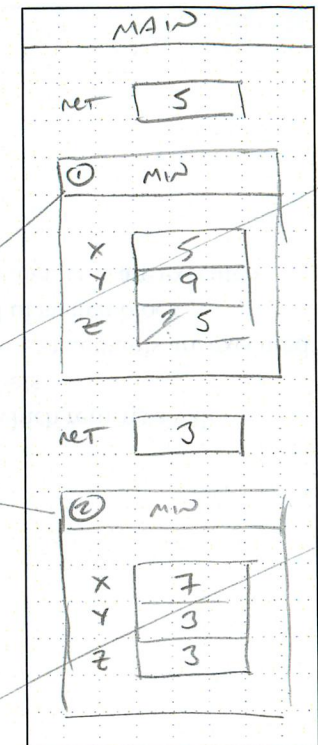
- If the first cond expression is true, execute first then block
- If the first cond expression is false, evaluate second cond expression
- If second cond expression is true, execute second then block
- If second cond expression is false, execute else block

```

1 int min( int x, int y )
2 {
3   int z;
4   if ( x < y ) {
5     z = x;
6   }
7   else {
8     z = y;
9   }
10  return z;
11 }
12
13 int main()
14 {
15   min( 5, 9 );
16   min( 7, 3 );
17   return 0;
18 }

```

non linear forward control flow
conditional expr
then
else

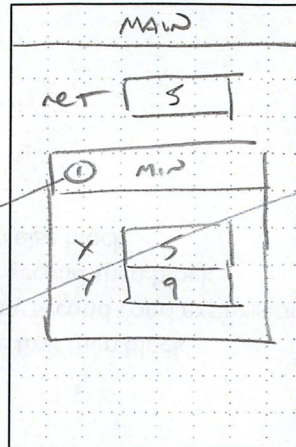


```

1 int min( int x, int y )
2 {
3     if ( x < y ) {
4         return x;
5     }
6     return y;
7 }
8
9 int main()
10 {
11     min( 5, 9 );
12     return 0;
13 }

```

Handwritten notes:
 - Red arrow from line 4 to line 5: "return needs NOT be at end"
 - Red arrow from line 6 to line 5: "else NOT required"

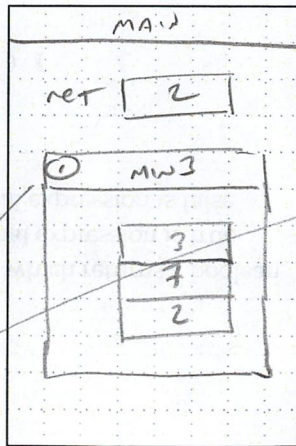


```

1 int min3( int x, int y, int z )
2 {
3     if ( x < y ) {
4         if ( x < z )
5             return x;
6     }
7     else if ( y < z ) {
8         return y;
9     }
10    return z;
11 }
12
13 int main()
14 {
15     min3( 3, 7, 2 );
16     return 0;
17 }

```

Handwritten notes:
 - Red arrow from line 15 to line 16: "return z"



5.3. switch/case Conditional Statements

```

1 switch ( selection_expression ) {
2
3     case case_label0:
4         case_statements0;
5         break;
6
7     case case_label1:
8         case_statements1;
9         break;
10
11    case case_label2:
12        case_statements3;
13        break;
14
15    default:
16        default_statements;
17
18 }

```

- A **selection expression** is an expression which returns a value
- The value is matched against the **case labels**
- If there is a match, then corresponding **case statements** are executed
- A **break** statement means to jump to end of switch block
- If no case labels match then the **default statements** are executed

```

1 int days_in_month( int month )
2 {
3     int x;
4     switch ( month )
5     {
6         case 1: x = 31; break;
7         case 2: x = 28; break;
8         case 3: x = 31; break;
9         case 4: x = 30; break;
10        case 5: x = 31; break;
11        case 6: x = 30; break;
12        case 7: x = 31; break;
13        case 8: x = 31; break;
14        case 9: x = 30; break;
15        case 10: x = 31; break;
16        case 11: x = 30; break;
17        case 12: x = 31; break;
18        default: x = -1;
19    }
20    return x;
21 }
22
23 int main()
24 {
25     int result = days_in_month( 7 );
26
27     // Indicate error to the system
28     if ( result == -1 )
29         return 1;
30
31     // Indicate success to the system
32     return 0;
33 }

```

Scope!

selects expr

case label

case statements

break - got end of switch block

error checking

default case

tells system there was an error

comment

```

1 int days_in_month( int month )
2 {
3     int x;
4     if ( month == 2 ) {
5         x = 28;
6     }
7     else {
8         switch ( month )
9         {
10            case 1:
11            case 3:
12            case 5:
13            case 7:
14            case 8:
15            case 10:
16            case 12:
17                x = 31;
18                break;
19
20            case 4:
21            case 6:
22            case 9:
23            case 11:
24                x = 30;
25                break;
26
27            default:
28                x = -1;
29        }
30    }
31    return x;
32 }

```

fall through

Identifying Primes

Write a C function that takes one integer input (x) that is between 0 and 9 (inclusive) and returns a boolean output. The function should return true if the input is prime (i.e., 2,3,5,7) and return false if the input is not prime. Use a case/switch conditional statement to explicitly check for primes.

```
int is_prime( int x ) {
```

```
    SWITCH (x) {
```

```
        case 0 : return 0;
```

```
        case 1 : return 0;
```

```
        case 2 : return 1;
```

```
        case 3 : return 1;
```

```
        case 4 : return 0;
```

```
        case 5 : return 1;
```

```
        case 6 : return 0;
```

```
        case 7 : return 1;
```

```
        case 8 : return 0;
```

```
        case 9 : return 0;
```

```
        default: return -1;
```

```
    }
```

```
}
```

activity

6. Iteration Statements

- **Iteration statements** enable programs to repeat code multiple times based on a conditional expression
- Iteration statements enable **backward flow control**
- Two primary kinds of iteration statements: **while** and **for** loops

6.1. while Loops

```
1 while ( conditional_expression ) {
2     loop_body;
3 }
```

- A **conditional expression** is an expression which returns a boolean
- The **loop body** is executed as long as conditional expression is true
- An **infinite loop** is when conditional expression is never true

```

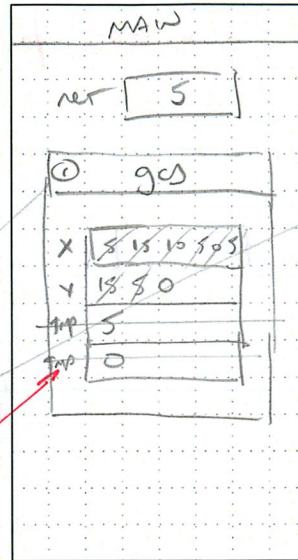
1  int gcd( int x, int y )
2  {
3      while ( y != 0 ) {
4          if ( x < y ) {
5              int temp = x;
6              x = y;
7              y = temp;
8          }
9          else {
10             x = x - y;
11          }
12      }
13      return x;
14  }
15
16  int main()
17  {
18      gcd(5,15);
19      return 0;
20  }

```

loop body

conditional expression

Allocate new temp because first temp goes away at end of scope!



stmt	x	y
4	5	15
4	15	5
4	10	5
4	5	5
4	0	5
4	5	0

6.2. for Loops

```

1  for ( initialization_stmt; cond_expr; increment_stmt ) {
2      loop_body;
3  }

```

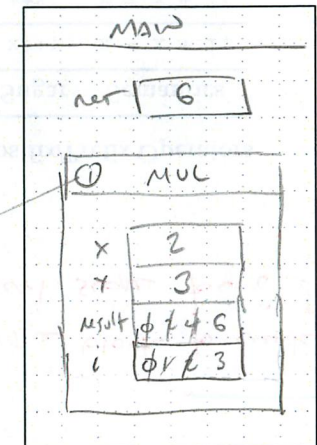
- The **initialization statement** is executed once before loop executes
- The **loop body** is executed as long as **conditional expression** is true
- The **increment statement** is executed at the end of each iteration

```

1  int mul( int x, int y )
2  {
3      int result = 0;
4      for ( int i=0; i<y; i=i+1 ) {
5          result = result + x;
6      }
7      return result;
8  }
9
10 int main()
11 {
12     mul(2,3);
13     return 0;
14 }

```

loop body



Output a sequence

** ACTIVITY **

Write a C function that takes one integer input (N) that is non-negative. The C function should output a sequence of integers according to the pattern on the left.

N	output
0:	0
1:	0 0
2:	0 0 0
3:	0 0 0 3
4:	0 0 0 3 4
5:	0 0 0 3 4 5
6:	0 0 0 3 4 5 6

```
void print_seq( int N ) {
```

```
    for ( int i=0; i < N+1; i = i+1 ) {
        if ( i < 3 )
            printf( "0 " );
        else
            printf( "%d ", i );
    }
```

Try of cpp.sh ?

7. Syntactic Sugar

- Syntactic sugar adds new syntax but not new semantics
- Syntactic sugar makes it easier to write certain programming patterns
- Syntactic sugar does not introduce any fundamentally new behavior

for loops are syntactic sugar

```
1 for ( int i = 0; i < y; i = i+1 ) {
2     result = result + x;
3 }

1 {
2     int i = 0;
3     while ( i < y ) {
4         result = result + x;
5         i = i + 1;
6     }
7 }
```

*no new block to create
local scope for i !*

Assignment Operators

Sugar	Semantics
$x += y;$	$x = x + y;$
$x -= y;$	$x = x - y;$
$x *= y;$	$x = x * y;$
$x /= y;$	$x = x / y;$

Postfix/Prefix Operators

Sugar	Semantics
$x++;$	$x = x + 1;$
$++x;$	$x = x + 1;$
$x--;$	$x = x - 1;$
$--x;$	$x = x - 1;$

Be careful, the value of $++x$ is $x + 1$, but the value of $x++$ is x .

```
1 int i = 1;
2 int j = ++i; // i == 2; j == 2
3 int k = i++; // i == 3; k == 2
```