Programming Language Concepts
Control Flow

Janyl Jumadinova
2 February, 2017
What is the value of the variable i after executing the following code (in either Java or C)?

```java
i = 10;
i = i++;
```
Expression evaluation

What is the value of the variable i after executing the following code (in either Java or C)?

```
i = 10;
i = i++;
```

Same Question

```
i = 10;
i = ++i;
```
Expression Evaluation

Sequential execution–expression evaluation:

```
i = 10;
i = i++;
```

Answer: \( i = 10 \)

Problem: \( i++ \) is an expression, but it has a side effect: it adds one to \( i \). The convention is that the value of the expression \( i++ \) is the original value of \( i \). (This is called the “post-increment” operator.)

```
i = 10;
i = ++i;
```

Answer: \( i = 11 \)

Why the “\( \times \)”? This is VERY BAD CODE! (But it’s legal)
Expression Evaluation

```c
int i=6, j=3, k=2;
int m = i/j*k;
```

So since $6/6 = 1$, $m = 1$, right?
int i=6, j=3, k=2;
int m = i/j*k;

So since 6/6 = 1, m = 1, right?

Problem: / and * have equal precedence and they are left-associative: i/j*k = (i/j)*k = (6/3)*2 = 4.
The assignment operator “=” produces a value, just like other operators.

The value of the expression “i = 10” is 10.
The assignment operator “=” produces a value, just like other operators.

The value of the expression “i = 10” is 10.

This is a right-associative operator:

“i = j = k = 10” means “i = (j = (k = 10))” and has the effect of setting all three variables to the same value, 10.

In C, this can cause serious program bugs!
The following is legal in C:
\[ i = 0; \text{if (} i = 10 \text{) printf("i is 10");} \]

The effect is to assign 10 to the variable \( i \), then see if the resulting value (namely 10) is non-zero (in C, non-zero values represent “true”). This will always evaluate to true!
The following is legal in C:
```
i = 0; if (i = 10) printf("i is 10");
```

The effect is to assign 10 to the variable `i`, then see if the resulting value (namely 10) is non-zero (in C, non-zero values represent “true”). This will always evaluate to true!

The programmer probably meant to write:
```
if (i == 10) printf(‘‘i is 10’’);
```
Operators

**Infix:** operator goes between operands
- \(a+b\), \(3*x\), \(m \% 5\), etc.

**Prefix:** operator, then operands. Used in, e.g., LISP
- \((* (+ 2 4) (/ 1.2 4))\)

**Postfix:** operands, then operator. Used in PostScript:
- `/w 100 def`
- `/h 200 def`
- `100 100 moveto w 0 rlineto 0 h rlineto`
A Very Unusual Operator: ?

- Most operators are either binary (+, −, *, <, ==, &&, etc.) or unary (“plus sign” +, “minus sign” -, ++, !, etc.).
- However, C and Java also have a ternary operator (takes 3 arguments).
A Very Unusual Operator: ?

- Most operators are either binary (+, −, *, <, ==, &&, etc.) or unary ("plus sign" +, "minus sign" -, ++, !, etc.).
- However, C and Java also have a ternary operator (takes 3 arguments).

**Conditional operator “?”**

boolean-expression ? expression1 : expression2

- The boolean-expression is evaluated. If it is true, the value is expression1, otherwise it is expression2.
- It has very low precedence, just above assig
A Very Unusual Operator: ?

- Most operators are either binary (+, −, *, <, ==, &&, etc.) or unary ("plus sign" +, "minus sign" -, ++, !, etc.).
- However, C and Java also have a ternary operator (takes 3 arguments).

Conditional operator "?"

boolean-expression ? expression1 : expression2

- The boolean-expression is evaluated. If it is true, the value is expression1, otherwise it is expression2.
- It has very low precedence, just above assig For example:
  5 < 10?70 : −3 is 70, while 5 > 10?70 : −3 is −3
Evaluate these Java expressions

```java
int i = 10;
int a,b,c,d,e;
a = b = c = ++i; // value1: _____
i = a==b ?20:30; // value2: _____
!(10==20) && 5 < 3 || 2 < 1 // value3:____
i = 0;
d = (d=++i)+d; // value4:_____  
i = 0;
e = e+(e=++i); // value5:_____  
```
Many Other Operators

Bitwise Operators
- $10 \mid 7 = 15$ (bitwise “or”)
- $10 \& 7 = 2$ (bitwise “and”)
- $10 << 3 = 80$ (left shift)
- $10 >> 1 = 5$ (right shift)
Many Other Operators

**Bitwise Operators**
- $10|7 = 15$ (bitwise “or”)
- $10&7 = 2$ (bitwise “and”)
- $10 << 3 = 80$ (left shift)
- $10 >> 1 = 5$ (right shift)

**String operators:**
- “Hello” + “world”
Many Other Operators

Bitwise Operators
- $10 | 7 = 15$ (bitwise “or”)
- $10 & 7 = 2$ (bitwise “and”)
- $10 << 3 = 80$ (left shift)
- $10 >> 1 = 5$ (right shift)

String operators:
- “Hello” + “world”

Referencing/dereferencing operators (C):
- &,
- *
- →
Operators in Other Language

Exponentiation (raising to a power):
In Python, “**” is the exponentiation operator.

```
$ python
>>> 100**2
10000
>>> 1000**(1./3.)
9.999999999999998
>>> 4**3**2
262144
>>> (4**3)**2
4096
```

NOTE: exponentiation is usually right-associative, since we normally interpret

\[ 4^3^2 \] as \[ (4^{3^2}) \]
Conditional Branches

Familiar to most novice programmers:

- “if” and “if-else” statements
- “switch” statements
- Basic idea: if (condition) then ... else ...
- It wasn’t always quite this easy, though
if (i+j-k)10,20,30
10 print *,"i+j-k is negative"
go to 40
20 print *,"i+j-k is zero"
go to 40
30 print *,"i+j-k is positive"
40 stop
end

Evaluate i+j-k and take one of three branches:
statement 10 if i+j-k < 0,
statement 20 if i+j-k = 0,
statement 30 if i+j-k > 0

(You can run this in the lab -- look for file “arith-if.for” in the repository and follow instructions in comments.)
The “go to” Statement

- “go to” is an UNCONDITIONAL branch.
- Most early programming languages had go to statements.
- Later languages like C also adopted them.
- But, they were easy to misuse.
(Contrived) Example (in C):

```c
for (i = 0; i < 5; i++) {
    if (i == 3) goto OUTSIDE;
    INSIDE: printf("inside\n");
}

goto FINISH;
OUTSIDE: printf("outside\n");
goto INSIDE;
FINISH: ...
```

OUTPUT:

```
inside
inside
outside
inside
inside
```
Edsger W. Dijkstra (world famous computer scientist -- “Dijkstra’s Algorithm”, etc.) wrote a letter to the *Communications of the ACM* in 1968:

**Go To Statement Considered Harmful**

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

*CR Categories:* 4.22, 5.23, 5.24

**EDITOR:**

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More
HUGE response. Letter is now famous; many imitations. “Considered harmful” essays appear about almost every topic in computer science:

- XMLHttpRequest Considered Harmful
- Csh Programming Considered Harmful
- Turing Test Considered Harmful
- Considered Harmful Essays Considered Harmful
- ... etc ...
The “go to” Statement

But why?

- We can “break out of scope” with a goto (the for-loop block might have its own local variables)
- We can write incomprehensible code (“spaghetti code”)

IN-CLASS EXERCISE: write some spaghetti code - get it out of your system!